Programming Language Design

2015

Week #8: Type Inference &
The Expression Problem

Instructor: Hidehiko Masuhara

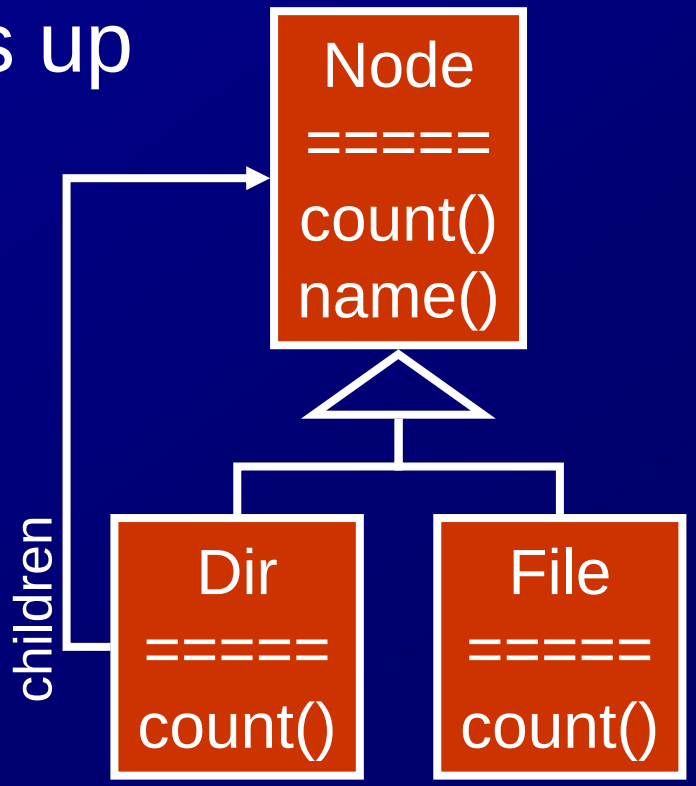
Quiz 1/3 (5 min.)

1. Enumerate the ***advantages*** of dynamically-typed languages over statically-typed languages
2. Enumerate the ***disadvantages*** of dynamically-typed languages over statically-typed languages

Quiz 2/3 (5 min.)
Criticize a library for file systems

- count() recursively sums up the total number of files

(Note: library users can't modify the definitions of the library itself)



Quiz 3/3 (10 min.)
Design a library for file systems

- A ***File*** has a name and a size
- A ***Dir*** has children (of ***File***s and ***Dir***s)
- Library users want to (all recursively)
  - count total number of files in a directory*
  - know total size of files in a directory*
  - find a file in a directory* and to know its size
  - etc.

(*files in a directory include those in subdirectories)

Note: users can't modify the library definition

Note: traversing directories should be implemented inside of the library

Type Inference

- a technique to determine types of expressions in untyped programs
  - eg. given:

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

  infers: "sunOf : ((int→int) × int)→int"

- successful in functional languages (eg. ML, Haskell)

Method of type inference

Classical type-checking

"expression means express and statemer

- Type-checking rules
  - Rule: (for each type of expression) permitted combinations of types of subexpressions and a type of result
    - eg: if e1 then e2 else e3 ➔ e1 is bool, e2 and e3 have the same result type
  - Implementation: 1. typing each expression, 2. check again the rule
- Typing (giving a type to an expression)
  - ~~Types of variables & functions are declared in a program~~
  - ~~Interpret the type-checking rule as "deriving a result type from types of subexpressions"~~
    ~~eg. if e1 then e2 else e3 ➔ result type = result type of e2~~

types of constants and built-in functions are predetermined

search for a type assignments that satisfies the rules

- Search problem; hence more than one solution
- Choose the most general solution
  - if there is only one most general solution, it is called a principal type
  - often requires generic types

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

Type inference in OOPL

- usu. requires some type information from programmers
- eg. Scala: infers only local variables

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

(from scala-lang.org)

Type inference in OOPL
Why Scala infers only local variable types?

- because objects are evil !

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

  - what's the type of obj?
  - then what's the return type of obj.name?

Can dynamically-typed languages have type inference?

- Dynamically-typed languages
  - Lisp, Smalltalk, Python, Ruby, etc.
  - ***Type-checks at runtime***
    - by attaching type information to values
    - by selecting appropriate operations based on the attached types
      eg. def double(x) { return x+x; }
      → check type of x upon +, and do iadd, fadd, or error
- If we can infer types, we can eliminate runtime type checking!
  - because correct programs do not "get wrong"

Can dynamically-typed languages have type inference?

- We already know full type inference for OOP is difficult (if not impossible)
- What other difficulties?
  - Implicit union types:

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

  - Recursive functions:

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

Type-declarations in dynamically-typed languages: "declare" in Common Lisp

- Role: a hint to a compiler
- eg.

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

- Consequence: a compiler can generate code with assuming that x always has a fixnum
  - No guarantee if the assumption is wrong

Type-checks at runtime
- by attaching type information to values
- by selecting appropriate operations based on the attached types
  eg. def double(x) { return x+x; }
  → check type of x upon +, and do iadd, fadd, or error

Type inference in dynamically-typed languages: soft typing [CF91]

- a system that infers parts of a program
- Purpose: statically detecting type errors, eliminating redundant runtime type checking
- Method:
  - mostly the same as in statically typed languages
  - use a union type when it cannot determine to one type

    eg.
    ```
    def sq(x) {
        if (x<0) return "NG"; else return sqrt(x); }
    ```
    → sq : int -> string + float
  - leave runtime type checking if for undecidable expressions

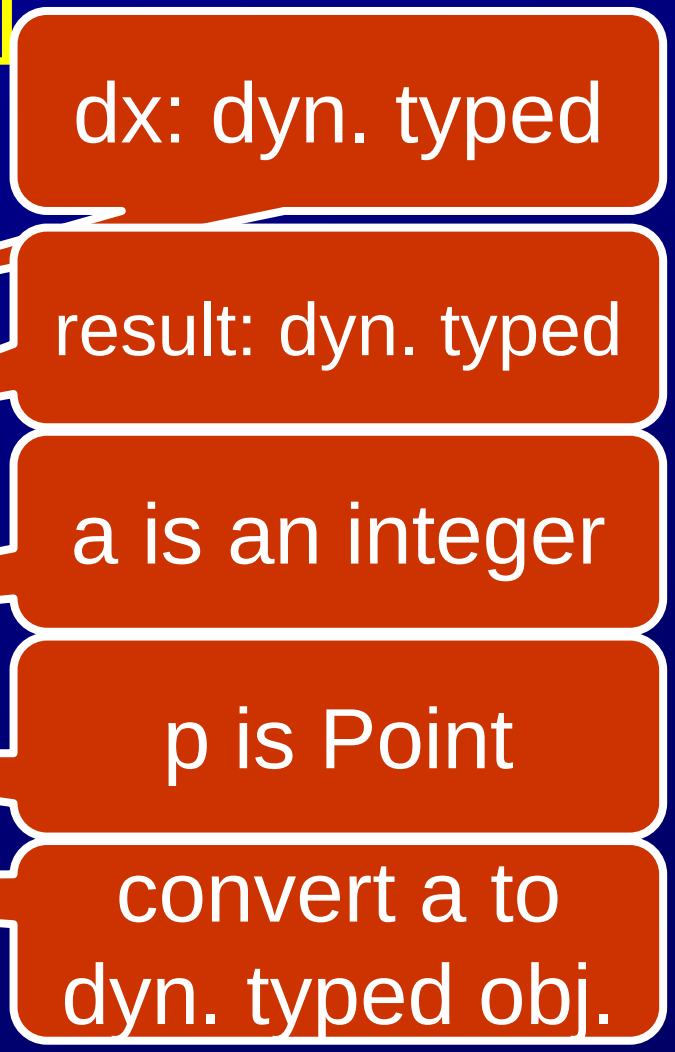
    eg. sq(123)*2

Gradual typing [ST07]

- Motivation: dynamically-typed languages are better for quick prototyping as we don't need to think about types
- Statically-typed languages are better in finding type errors before execution
- Start development in a DTL, and then migrate into a STL
  - but it is hard to migrate all at once
  - so, migrate gradually → gradual typing
  - the programmer can choose typed and untype definition on a per-module basis
    - challenges: passing an untyped function to the typed world, and vice versa

Example of gradual typing [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)
```



Example of gradual typing [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 has at least a field x of int

Point is infered as [x:int, equal: [x:int]->bool]

statically safe

Example of gradual typing [ST07]

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

(statically)
type error

Languages with gradual types

- Newspeak [Bracha08]: a la Smalltalk (called ***optional typing***)
- TypedClojure: extension to Clojure
- TypeScript: extension to Javascript
- Dart: a la Java

Implicit type conversion

- how "int i=...; float f=...; return i + f;" returns float?
  - due to a rule "int + float → float", or
  - implicitly converted to "return itof(i)+f;"
    (so called upcasting: conversion to more general type)
- (Java)Person x=...; println("x=" + x);
  - implicitly converted to println("x=" + x.toString());
  - NB. no subtyping between Person & String
  - but only between specific types

User-defined implicit type conversion

(Scala) quoted from [OSV08]

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

printWithSpaces("xyz")
```

converted by using stringWrapper

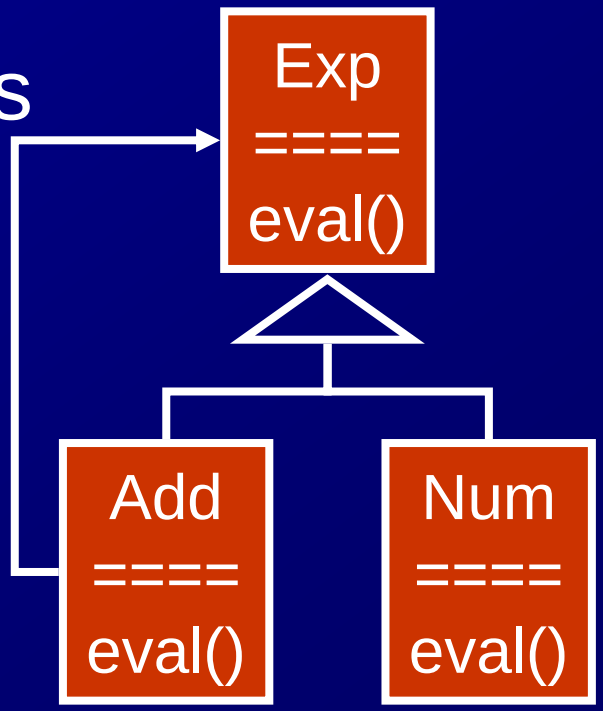
The Expression Problem
[Wadler98]

- A problem of extending data structure and its operations in a type-safe way
  - known for a long time (named by [Wadler98])
- Extending = without changing existing definitions, realizing
  - addition of new variant to the data structure
  - addition of new operations

EP: a simple interpreter

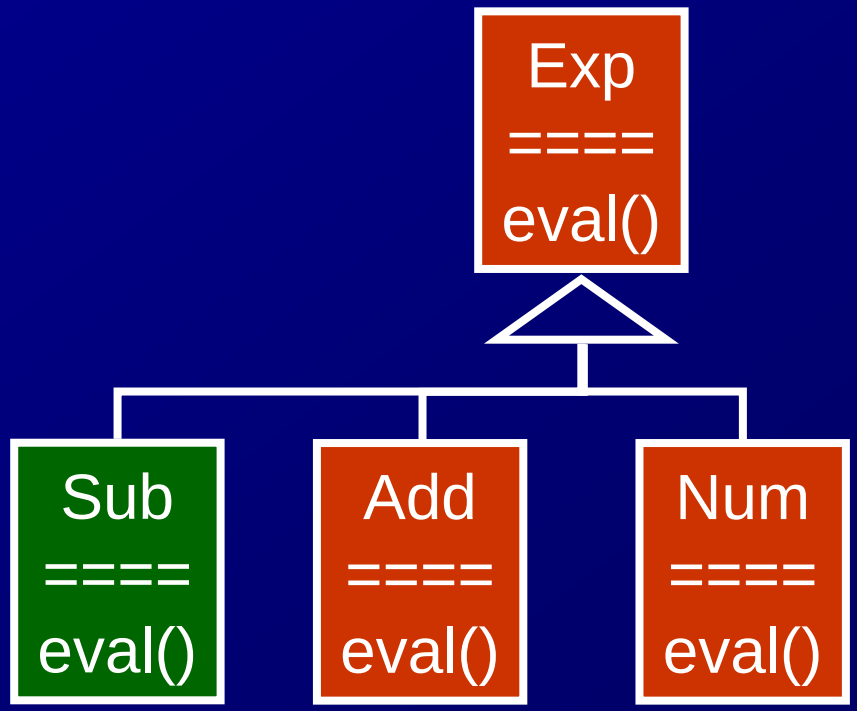
(single inheritance)

- Exp represents AST nodes
  eval() computes a value
  of an expression
- new Add(new Num(1),
  new Num(2)).eval()
  ➔ 3



EP: addition of a data variant

- to support subtraction
  - by adding Sub with eval
- without changing existing classes!

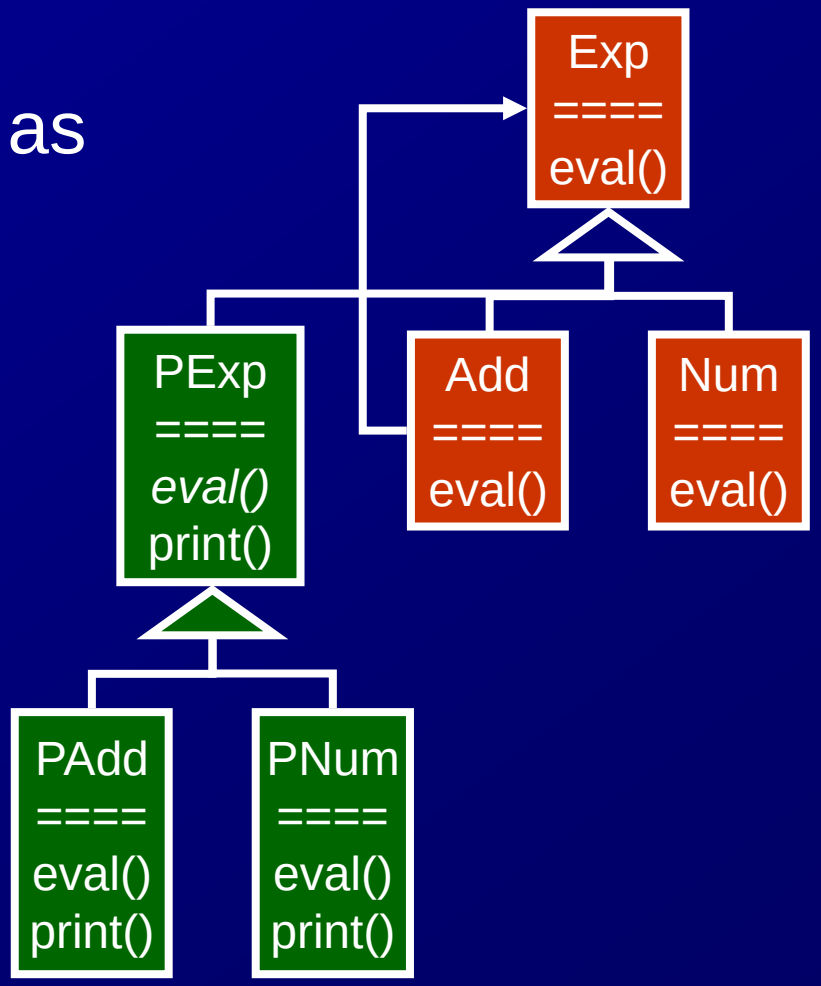


EP: addition of operations

- to print an expression as a string

1. NG: adding print() to Exp ← existing class!
2. Define PExp as a subclasses of Exp → OK?

   cannot inherit definition of eval

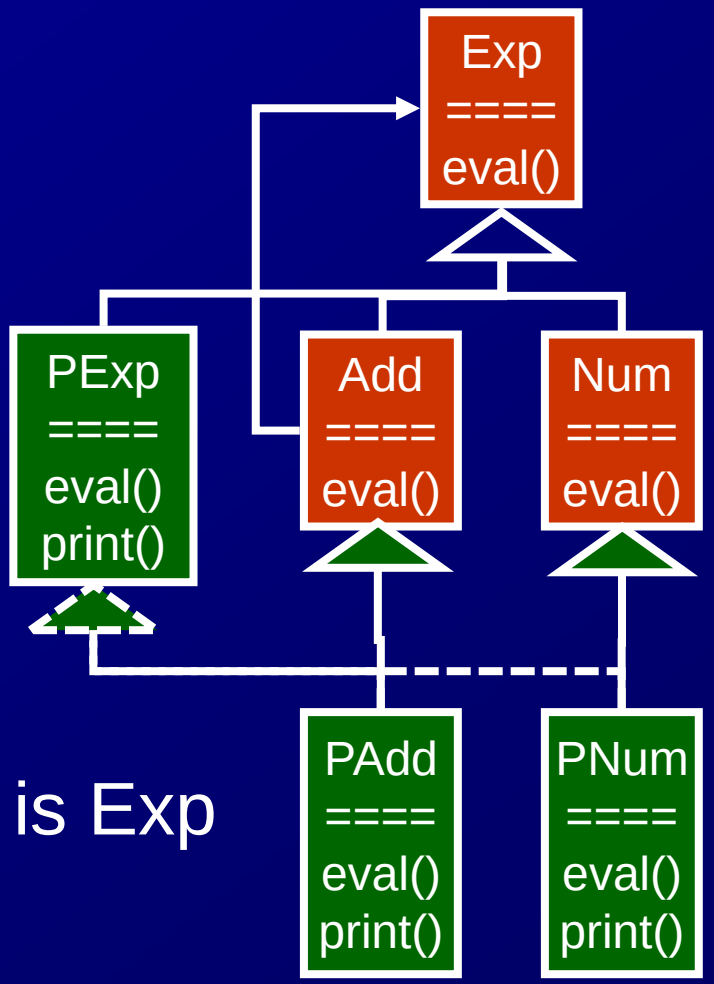


EP: addition of operations

- to print an expression as a string

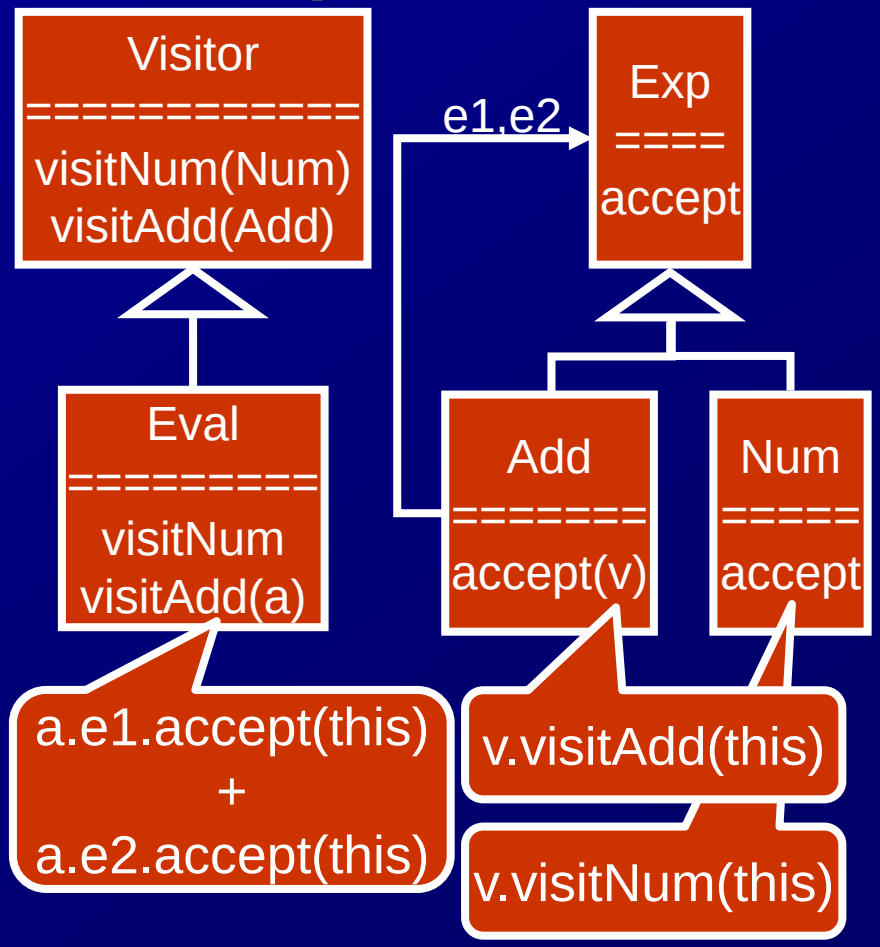
1. NG: adding print() to Exp ← existing class!
2. Define PExp as a subclasses of Exp
   → OK?

   subexpression of PAdd is Exp
   → no print method



EP: using the Visitor pattern

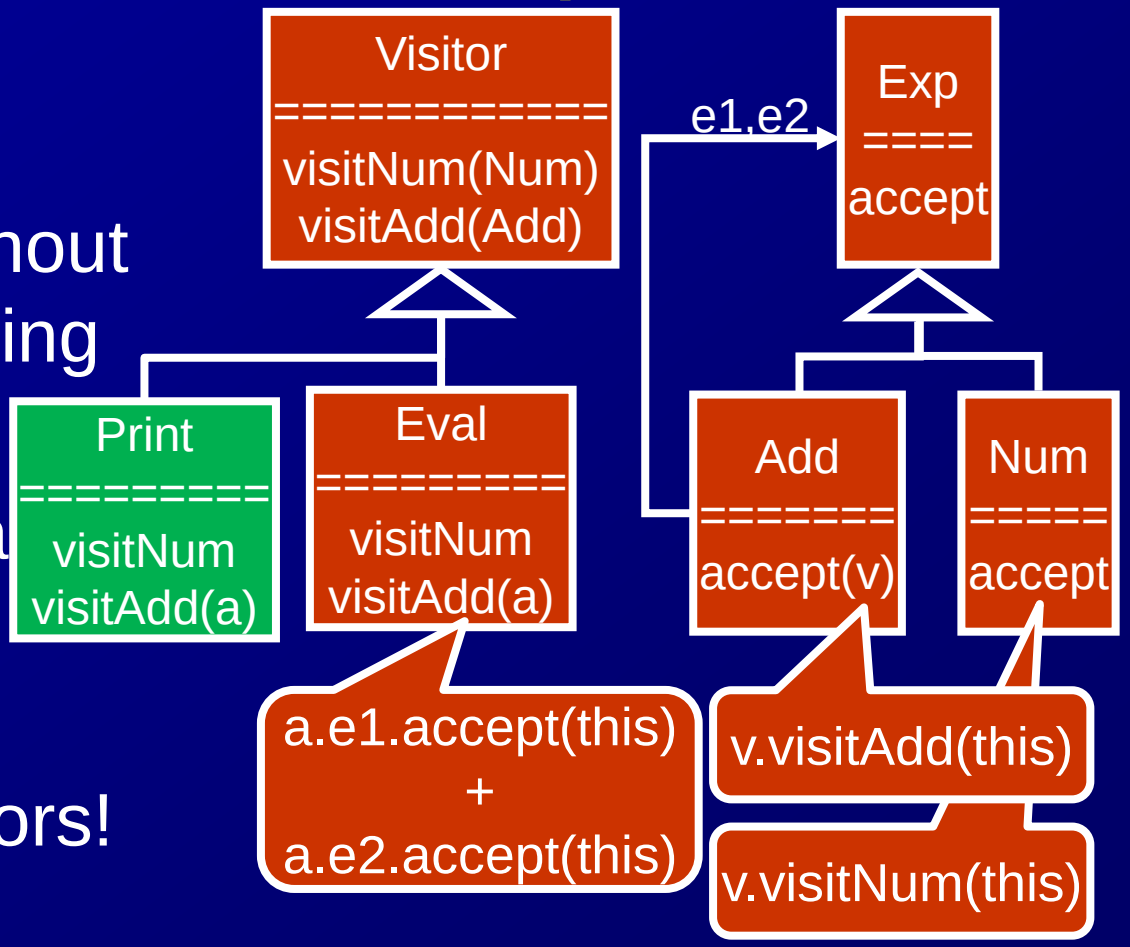
- one of the GoF design patterns
- to separate data structure from its operations
- additional operations without changing existing code!



EP: using the Visitor pattern

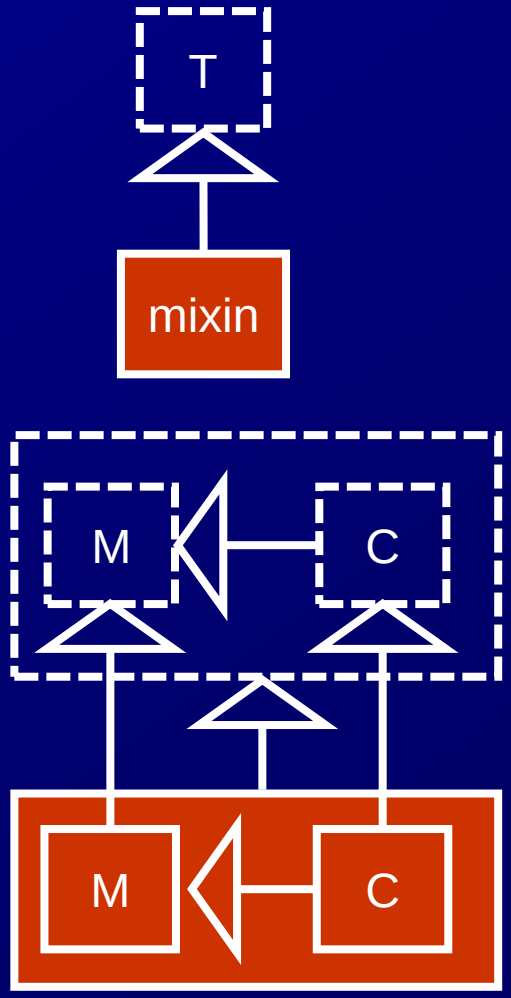
- additional operations without changing existing code!
- additional data variant?

→need to change existing visitors!

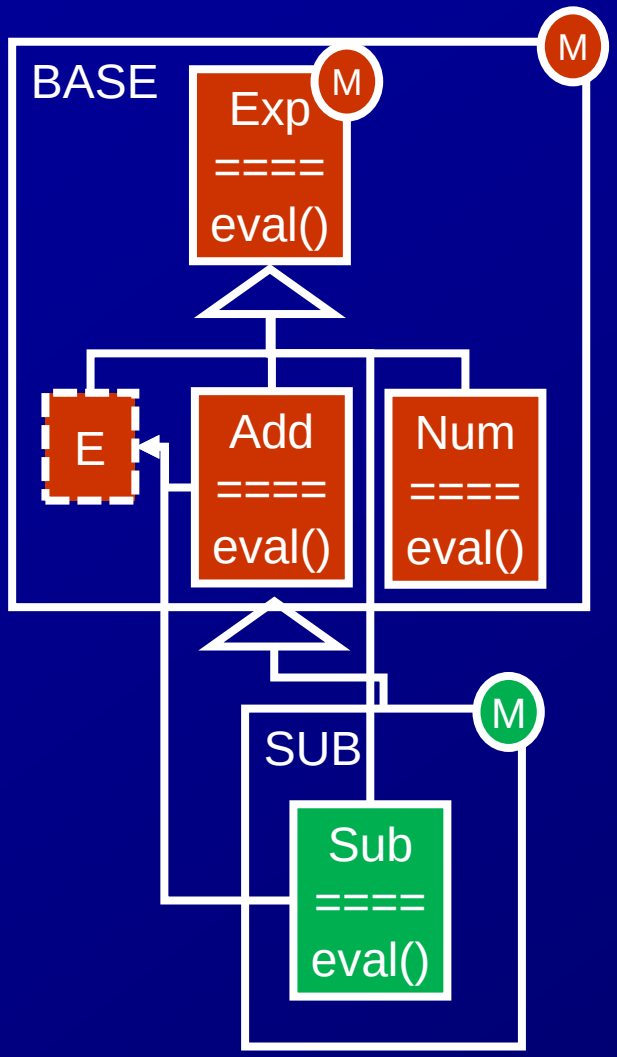


A solution with Mixin layers

- (review)mixin: a class parameterized with its superclass
- Mixin layers: nested mixins [SB98]



A solution of EP with Mixin layers(and type variables) [ZO04]



- M: define as mixin
- E: a type variable constrained as subtype of Exp
  - subexpression in Add is of type E
- adding data variant: straightforward

A solution of EP with Mixin layers: addition of operation

BASE

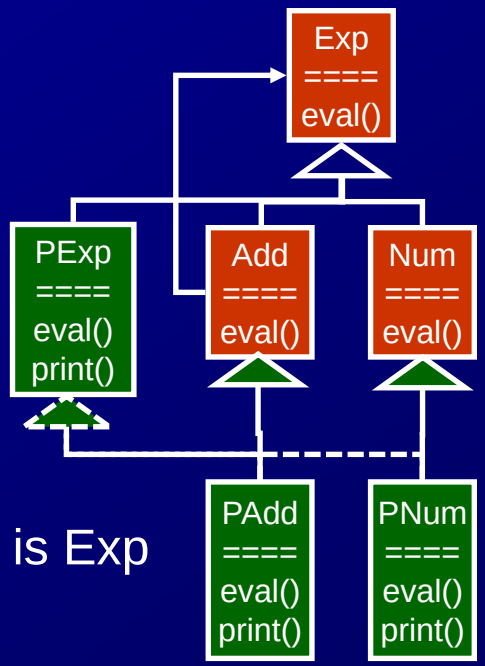
E

EP: addition of operations

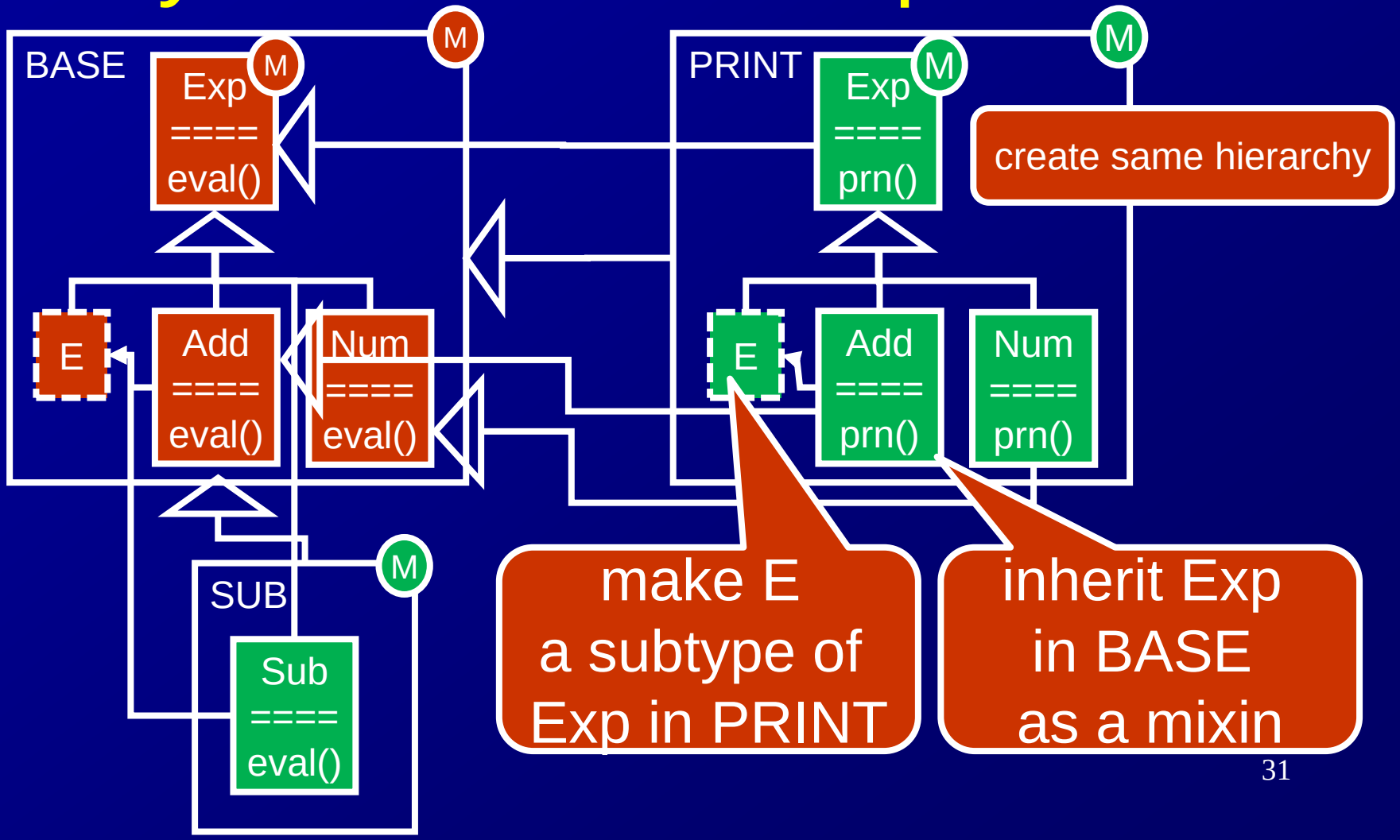
- to print an expression as a string

1. NG: adding print() to Exp ← existing class!
2. Define PExp as a subclasses of Exp → OK?

   subexpression of PAdd is Exp → no print method



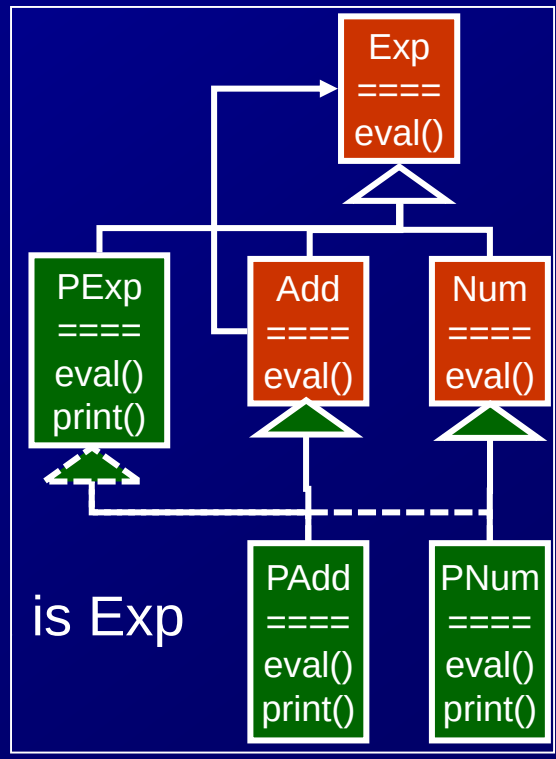
A solution of EP with Mixin layers: addition of operation



A solution of EP with generics and constrained type parameters

```
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() ...} }
```



References

[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.