香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 4)

COMP 3031, Fall 2025

Lionel Parreaux

Working with Class Hierarchies

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

```
abstract class Expr:
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr

class Number(n: Int) extends Expr:
  def isNumber = true
  def isSum = false
  def numValue = n
  def leftOp = throw Error(”Number.leftOp”)
  def rightOp = throw Error(”Number.rightOp”)
```

Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr:
  def isNumber = false
  def isSum = true
  def numValue = throw Error(”Sum.numValue”)
  def leftOp = e1
  def rightOp = e2
```

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int =
  if e.isNumber then e.numValue
  else if e.isSum then eval(e.leftOp) + eval(e.rightOp)
  else throw Error("Unknown expression " + e)
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

Problem: There's no static guarantee you use the right accessor functions. You might hit an `Error` case if you are not careful.

Problem: There's no static guarantee we have not forgotten some cases.

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(name: String)        extends Expr   // Variable ‘name’
```

You need to add methods for classification and access to all classes defined above.

```
abstract class Expr:
  ...
  def isProd: Boolean
  def isVar: Boolean
  def nameValue: String
```

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(name: String)        extends Expr   // Variable ‘name’
```

You need to add methods for classification and access to all classes defined above.

```
abstract class Expr:
  ...
  def isProd: Boolean
  def isVar: Boolean
  def nameValue: String
```

Non-Solution: Type Tests and Type Casts

A “hacky” solution could use type tests and type casts.

Scala let’s you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object’s type conforms to
def asInstanceOf[T]: T        // treats this object as an instance of type ‘T‘
                              // throws ‘ClassCastException‘ if it isn’t.
```

These correspond to Java’s type tests and casts

```
Scala                   Java

x.isInstanceOf[T]       x instanceof T
x.asInstanceOf[T]       (T) x
```

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if e.isInstanceOf[Number] then
    e.asInstanceOf[Number].numValue
  else if e.isInstanceOf[Sum] then
    eval(e.asInstanceOf[Sum].leftOp)
    + eval(e.asInstanceOf[Sum].rightOp)
  else throw Error(”Unknown expression ” + e)
```

This is ugly and still generally unsafe.

Eval with Type Tests and Type Casts

Here's a formulation of the eval method using type tests and casts:

```
def eval(e: Expr): Int =
  if e.isInstanceOf[Number] then
    e.asInstanceOf[Number].numValue
  else if e.isInstanceOf[Sum] then
    eval(e.asInstanceOf[Sum].leftOp)
    + eval(e.asInstanceOf[Sum].rightOp)
  else throw Error(”Unknown expression ” + e)
```

This is ugly and still generally unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
abstract class Expr:
  def eval: Int

class Number(n: Int) extends Expr:
  def eval: Int = n

class Sum(e1: Expr, e2: Expr) extends Expr:
  def eval: Int = e1.eval + e2.eval
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Assessment of OO Decomposition

- OO decomposition mixes *data* with *operations* on the data.
- This can be the right thing if there's a need for encapsulation and data abstraction.
- On the other hand, it increases complexity(*) and adds new dependencies to classes.
- It makes it easy to add new kinds of data but hard to add new kinds of operations.

(*) In the literal sense of the word:

complex = *plaited, woven together*

Thus, complexity arises from mixing several things together.

Limitations of OO Decomposition

OO decomposition only works well if operations are on a *single* object.

What if you want to simplify expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

You are back to square one; you need test and access methods for all the different subclasses.

Pattern Matching

Reminder: Decomposition

The task we are trying to solve is find a general and convenient way to access heterogeneous data in a class hierarchy.

Attempts seen previously:

- *Classification and access methods*: quadratic explosion
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: causes coupling between data and operations; need to modify all classes to add a new method; cannot inspect arguments.

Solution 2: Functional Decomposition with Pattern Matching

Observation: the sole purpose of test and accessor functions is to *reverse* the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

This situation is so common that many functional languages, Scala included, automate it.

Case Classes

A *case class* definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Like before, this defines a trait `Expr`, and two concrete subclasses `Number` and `Sum`.

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of `switch` from C/Java to class hierarchies.

It's expressed in Scala using the keyword `match`.

Example

```scala
def eval(e: Expr): Int = e match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

Match Syntax

Rules:

- `match` is preceded by a selector (or scrutinee) expression and is followed by a sequence of *cases*, `pat => expr`.
- Each case associates an *expression* `expr` with a *pattern* `pat`.
- A `MatchError` exception is thrown if no pattern matches the value of the selector.

Forms of Patterns

Patterns are constructed from:

- *constructors*, e.g. `Number`, `Sum`,
- *variables*, e.g. `n`, `e1`, `e2`,
- *wildcard patterns* `_`,
- *constants*, e.g. `1`, `true`.
- *type tests*, e.g. `n: Number`

Variables always begin with a lowercase letter.

The same variable name can only appear once in a pattern. So, `Sum(x, x)` is not a legal pattern.

Names of constants begin with a capital letter, with the exception of the reserved words `null`, `true`, `false`.

Evaluating Match Expressions

An expression of the form

$$\texttt{e match \{ case } p_1 \Rightarrow e_1 \ \dots \ \texttt{case } p_n \Rightarrow e_n \ \texttt{\}}$$

matches the value of the selector `e` with the patterns $p_1, \dots, p_n$ in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

What Do Patterns Match?

- A constructor pattern $\mathtt{C}(p_1, \ldots, p_n)$ matches all the values of type `C` (or a subtype) that have been constructed with arguments matching the patterns $p_1, \ldots, p_n$.
- A variable pattern `x` matches any value, and *binds* the name of the variable to this value.
- A constant pattern `c` matches values that are equal to `c` (in the sense of ==)

Example

Example

```
eval(Sum(Number(1), Number(2)))
```

$\rightarrow$

```
Sum(Number(1), Number(2)) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

$\rightarrow$

```
eval(Number(1)) + eval(Number(2))
```

Example (2)

$\rightarrow$

```
(Number(1) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
) + eval(Number(2))
```

$\rightarrow$

```
1 + eval(Number(2))
```

$\twoheadrightarrow$

```
3
```

Exhaustiveness of Pattern Matching

Problem: There's no static guarantee we have not forgotten some cases.

The code below currently compiles!

```
def eval(e: Expr): Int = e match
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

But this usage throws an exception:

```
eval(Number(2))  // scala.MatchError: Number(2) (of class ppl4.Number)
```

Solution: ***seal*** the data structure, prohibiting the above definition

```
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Exhaustiveness of Pattern Matching

Problem: There's no static guarantee we have not forgotten some cases.

The code below currently compiles!

```
def eval(e: Expr): Int = e match
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

But this usage throws an exception:

```
eval(Number(2))  // scala.MatchError: Number(2) (of class ppl4.Number)
```

Solution: ***seal*** the data structure, prohibiting the above definition

```
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Exhaustiveness of Pattern Matching

Problem: There's no static guarantee we have not forgotten some cases.

The code below currently compiles!

```
def eval(e: Expr): Int = e match
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

But this usage throws an exception:

```
eval(Number(2))  // scala.MatchError: Number(2) (of class ppl4.Number)
```

Solution: ***seal*** the data structure, prohibiting the above definition

```
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the base trait.

Example

```
sealed abstract class Expr:
  def eval: Int = this match
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
```

Exercise

Write a function `show` that uses pattern matching to return the representation of a given expressions as a string.

```
def show(e: Expr): String = ???
```

Exercise (Solution)

Write a function show that uses pattern matching to return the representation of a given expressions as a string.

```
def show(e: Expr): String = e match
  case Number(n) => n.toString
  case Sum(e1, e2) =>
    show(e1) + ” + ” + show(e2)
    // or, equivalently:
    s”${ show(e1) } + ${ show(e2) }”
```

Exercise (Optional, Harder)

Add case classes `Var` for variables `x` and `Prod` for products `x * y` as discussed previously.

Change your `show` function so that it also deals with products.

Pay attention you get operator precedence right but to use as few parentheses as possible.

Example

```
Sum(Prod(Number(2), Var("x")), Var("y"))
```

should print as "`2 * x + y`". But

```
Prod(Sum(Number(2), Var("x")), Var("y"))
```

should print as "`(2 + x) * y`".

Exercise (Solution)

```scala
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Product(e1: Expr, e2: Expr) extends Expr
case class Var(name: String) extends Expr
```

Exercise (Solution)

```scala
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Product(e1: Expr, e2: Expr) extends Expr
case class Var(name: String) extends Expr

def show(e: Expr): String = e match
  case Number(n) => n.toString
  case Sum(e1, e2) => s"${ show(e1) } + ${ show(e2) }"
  case Product(e1, e2) => s"${ showFactor(e1) } * ${ showFactor(e2) }"
  case Var(n) => n

def showFactor(e: Expr): String = e match
  case e: Sum => s"(${ show(e) })"
  case _ => show(this)
```

Lists

Lists

The list is a fundamental data structure in functional programming.

A list having $x_1, \ldots, x_n$ as elements is written `List(`$x_1, \ldots, x_n$`)`

Example

```
val fruits = List("apples", "oranges", "pears")
val nums   = List(1, 2, 3, 4)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty  = List()
```

There are two important differences between lists and arrays.

- Lists are immutable — the elements of a list cannot be changed.
- Lists are recursive (i.e., nested), while arrays are flat.

The List Type

Like arrays, lists are homogeneous: the elements of a list must all have the same type.

The type of a list with elements of type `T` is written `scala.List[T]` or shorter just `List[T]`

Example

```
val fruits: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]        = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Nothing]   = List()
```

Constructors of Lists

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruits = "apples" :: ("oranges" :: ("pears" :: Nil))
nums   = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

Right Associativity

Convention: Operators ending in ":" associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

We can thus omit the parentheses in the definition above.

Example

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

Operations on Lists

All operations on lists can be expressed in terms of the following three:

`head`	the first element of the list
`tail`	the list composed of all the elements except the first.
`isEmpty`	'true' if the list is empty, 'false' otherwise.

These operations are defined as methods of objects of type `List`. For example:

```
fruits.head      == "apples"
fruits.tail.head == "oranges"
diag3.head       == List(1, 0, 0)
empty.head       == throw NoSuchElementException("head of empty list")
```

List Patterns

It is also possible to decompose lists with pattern matching.

`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a `head` matching `p` and a `tail` matching `ps`.
`List(p1, ..., pn)`	same as `p1 :: ... :: pn :: Nil`

Example

`1 :: 2 :: xs`	Lists of that start with `1` and then `2`
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	The empty list, same as `Nil`
`List(2 :: xs)`	A list that contains as only element another list that starts with `2`.

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
O	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
X	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Sorting Lists

Suppose we want to sort a list of numbers in ascending order:

- One way to sort the list `List(7, 3, 9, 2)` is to sort the tail `List(3, 9, 2)` to obtain `List(2, 3, 9)`.
- The next step is to insert the head `7` in the right place to obtain the result `List(2, 3, 7, 9)`.

This idea describes *Insertion Sort* :

```
def isort(xs: List[Int]): List[Int] = xs match
  case Nil  => Nil
  case y :: ys => insert(y, isort(ys))
```

Exercise

Complete the definition insertion sort by filling in the `???`s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case Nil => ???
  case y :: ys => ???
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case Nil => List(x)
  case y :: ys =>
    if x < y then x :: xs else y :: insert(x, ys)
```

What is the worst-case complexity of insertion sort relative to the length of the input list N?

```
O      the sort takes constant time
O      proportional to N
O      proportional to N log(N)
O      proportional to N * N
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case Nil => List(x)
  case y :: ys =>
    if x < y then x :: xs else y :: insert(x, ys)
```

What is the worst-case complexity of insertion sort relative to the length of the input list `N`?

```
O     the sort takes constant time
O     proportional to N
O     proportional to N * log(N)
X     proportional to N * N
```

Pure Data

Pure Data

In the previous sessions, you have learned how to model data with class hierarchies.

Classes are essentially bundles of functions operating on some common values represented as fields.

They are a very useful abstraction, since they allow encapsulation of data.

But sometimes we just need to compose and decompose *pure data* without any associated functions.

Case classes and pattern matching work well for this task.

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
sealed abstract class Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

This time we have put all case classes in the `Expr` companion object, in order not to pollute the global namespace.

So it's `Expr.Number(1)` instead of `Number(1)`, for example.

One can still “pull out” all the cases using an import.

```
import Expr.*
```

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
sealed abstract class Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

Pure data definitions like these are called *algebraic data types*, or ADTs for short.

They are very common in functional programming.

To make them even more convenient, Scala offers some special syntax.

Enums for ADTs

An *enum* enumerates all the cases of an ADT *and nothing else*.

Example

```
enum Expr:
  case Var(s: String)
  case Number(n: Int)
  case Sum(e1: Expr, e2: Expr)
  case Prod(e1: Expr, e2: Expr)
```

This enum is equivalent to the case class hierarchy on the previous slide, but is shorter, since it avoids the repetitive `class ... extends Expr` notation.

Pattern Matching on ADTs

Match expressions can be used on enums as usual.

For instance, to print expressions with proper parameterization:

```
def show(e: Expr): String = e match
  case Expr.Var(x) => x
  case Expr.Number(n) => n.toString
  case Expr.Sum(a, b) => s"${show(a)} + ${show(a)}}"
  case Expr.Prod(a, b) => s"${showFactor(a)} * ${showFactor(a)}"

import Expr.*

def showFactor(e: Expr): String = e match
  case e: Sum => s"(${show(expr)})"
  case _ => show(expr)
```

Simple Enums

Cases of an `enum` can also be simple values, without any parameters.

Example

Define a `Color` type with values `Red`, `Green`, and `Blue`:

```
enum Color:
  case Red
  case Green
  case Blue
```

We can also combine several simple cases in one list:

```
enum Color:
  case Red, Green, Blue
```

Pattern Matching on Simple Enums

For pattern matching, simple cases count as constants:

```
enum DayOfWeek:
  case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

import DayOfWeek.*

def isWeekend(day: DayOfWeek) = day match
  case Saturday | Sunday => true
  case _ => false
```

More Fun With Enums

Enumerations can take parameters and can define methods.

Example:

```
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1,  0)
  case Up    extends Direction( 0,  1)
  case Left  extends Direction(-1,  0)
  case Down  extends Direction( 0, -1)
  def leftTurn = Direction.values((ordinal + 1) % 4)
end Direction

val r = Direction.Right
val u = r.leftTurn       // u = Up
val v = (u.dx, u.dy)     // v = (1, 0)
```

More Fun With Enums

Notes:

- Enumeration cases that pass parameters have to use an explicit `extends` clause
- The expression `e.ordinal` gives the ordinal value of the enum case `e`. Cases start with zero and are numbered consecutively.
- `values` is an immutable array in the companion object of an enum that contains all enum values.
- Only simple cases have `ordinal` numbers and show up in `values`, parameterized cases do not.

Enumerations Are Shorthands for Classes and Objects

The `Direction` enum is expanded by the Scala compiler to roughly the following structure:

```scala
abstract class Direction(val dx: Int, val dy: Int):
  def leftTurn = Direction.values((ordinal - 1) % 4)
object Direction:
  val Right = new Direction( 1,  0) { }
  val Up    = new Direction( 0,  1) { }
  val Left  = new Direction(-1,  0) { }
  val Down  = new Direction( 0, -1) { }
end Direction
```

There are also compiler-defined helper methods `ordinal` in the class and `values` and `valueOf` in the companion object.

Domain Modeling

ADTs and enums are particularly useful for domain modelling tasks where one needs to define a large number of data types without attaching operations.

Example: Modelling payment methods.

```
enum PaymentMethod:
  case CreditCard(kind: CardKind, holder: String, number: Long, expires: Date)
  case PayPal(email: String)
  case Cash

enum CardKind:
  case Visa, Mastercard, Amex
```

Summary

In this unit, we covered two uses of `enum` definitions:

- as a shorthand for hierarchies of case classes
- as a way to define data types accepting a finite set of values

The two cases can be combined: an enum can comprise parameterized and simple cases at the same time.

Enums are typically used for pure data, where all operations on such data are defined elsewhere.

How Other Languages Do It

Functional Languages

The `Expr` algebraic data type example in OCaml.

```
type expr = Number of int | Sum of expr * expr

let rec eval x =
  match x with
    Number n -> n | Sum (e1, e2) -> eval e1 + eval e2
```

In Haskell:

```
data Expr = Number Int | Sum Expr Expr

eval :: Expr -> Int
eval (Number n) = n
eval (Sum e1 e2) = eval e1 + eval e2
```

Functional Languages

The `Expr` algebraic data type example in OCaml.

```
type expr = Number of int | Sum of expr * expr

let rec eval x =
  match x with
    Number n -> n | Sum (e1, e2) -> eval e1 + eval e2
```

In Haskell:

```
data Expr = Number Int | Sum Expr Expr

eval :: Expr -> Int
eval (Number n) = n
eval (Sum e1 e2) = eval e1 + eval e2
```

Functional Languages

Commentary:

- ADTs in functional languages are often slightly more concise than Scala, due to more type inference and more lightweight syntax.
- On the other hand, Scala's class hierarchies are more expressive in various ways, such as allowing
  - members (methods, fields, etc.) in the base and derived classes
  - multi-level hierarchies with multiple sealed trait inheritance

    ```scala
    sealed trait EitherOrBoth[A, B]
    case class Both[A, B](a: A, b: B) extends EitherOrBoth[A, B]
    sealed trait Either[A, B] extends EitherOrBoth[A, B]
    case class Left[A, B](a: A) extends Either[A, B]
    case class Right[A, B](b: B) extends Either[A, B]
    ```
  - non-sealed (i.e., *open-ended*) hierarchies
  - existentials (similar to *Generalized* ADTs)

Languages Inspired by Functional Languages

The `Expr` algebraic data type example in Rust.

```
enum Expr {
  Number(i32),
  Sum(Box<Expr>, Box<Expr>),
}
fn eval(e: &Expr) -> i32 {
  match e {
    Expr::Number(n) => *n,
    Expr::Sum(e1, e2) => eval(e1) + eval(e2),
  }
}
```

Notice low-level details related to memory management (`Box`).

Languages Inspired by Functional Languages

The `Expr` algebraic data type example in Rust.

```
enum Expr {
  Number(i32),
  Sum(Box<Expr>, Box<Expr>),
}
fn eval(e: &Expr) -> i32 {
  match e {
    Expr::Number(n) => *n,
    Expr::Sum(e1, e2) => eval(e1) + eval(e2),
  }
}
```

Notice low-level details related to memory management (`Box`).

Imperative Languages

The `Expr` algebraic data type example in C.

```
typedef enum { NUMBER, SUM } ExprKind;
typedef struct Expr {
  ExprKind kind;
  union {
    struct { int n; } number;
    struct { struct Expr* e1; struct Expr* e2; } sum; } u;
} Expr;
int eval(Expr* e) {
  switch (e->kind) {
    case NUMBER: return e->u.number.n;
    case SUM: return eval(e->u.sum.e1) + eval(e->u.sum.e2); } }
```

Ugly *and* unsafe!

Imperative Languages

The `Expr` algebraic data type example in C.

```
typedef enum { NUMBER, SUM } ExprKind;
typedef struct Expr {
  ExprKind kind;
  union {
    struct { int n; } number;
    struct { struct Expr* e1; struct Expr* e2; } sum; } u;
} Expr;
int eval(Expr* e) {
  switch (e->kind) {
    case NUMBER: return e->u.number.n;
    case SUM: return eval(e->u.sum.e1) + eval(e->u.sum.e2); } }
```

Ugly *and* unsafe!

Imperative Languages

The `Expr` algebraic data type example in C++

We will not talk about C++ in this course.

Imperative Languages

The `Expr` algebraic data type example in C++

We will not talk about C++ in this course.

Languages With Structural Typing

The `Expr` algebraic data type example in TypeScript.

```
type Expr = Num | Sum;
type Num = { kind: "number"; n: number; }
type Sum = { kind: "sum"; e1: Expr; e2: Expr; }

function evalExpr(e: Expr): number {
  switch (e.kind) {
    case "number": return e.n;
    case "sum": return evalExpr(e.e1) + evalExpr(e.e2);
    default: return assertNever(e); // For exhaustiveness checking
  }
}
function assertNever(x: never): never { return x; }
```

Languages Inspired by Scala

Nowadays, many languages have adopted Scala-like pattern matching on sealed class hierarchies, including

- Kotlin
- Swift
- etc.
- and even modern/future Java versions