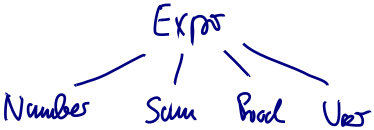
Pattern Matching

Reminder: Decomposition

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



eval
show
simplify

Attempts seen previously:

- *Classification and access methods*: quadratic explosion 40
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: does not always work, need to touch all classes to add a new method.

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.

new Sum(e_1, e_2)

Case Classes

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

```
trait 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`.

Case Classes (2)

It also implicitly defines companion objects with `apply` methods.

```
object Number {
  def apply(n: Int) = new Number(n)
}
object Sum {
  def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

Number (2)

↓

Number.apply(

so you can write `Number(1)` instead of `new Number(1)`.

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

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

Match Syntax

Rules:

- `match` 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.

```
e match {
  case pat1 => expr1
           ⋮
  case patn => exprn
}
```

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`.

Number (n)

Number (_)

1, true, "abc", N

val N = 2

Sum (Number (1), Var (x)) =>

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.

Sum (x, y)

Variable n

Constant N

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

$$\texttt{e match } \{ \texttt{ case } \texttt{p}_1 \Rightarrow \texttt{e}_1 \ \ldots \ \texttt{case } \texttt{p}_n \Rightarrow \texttt{e}_n \ \}$$

matches the value of the selector `e` with the patterns $\texttt{p}_1, \ldots, \texttt{p}_n$ 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 $C(p_1, \ldots, p_n)$ 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)))
```

→

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

→

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

Example (2)

→

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

→

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

↠

```
3
```

Pattern Matching and Methods

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

Example

```
trait Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

eval(e)

Expression Problem "

Expr
  def eval

Sum
  def eval = ...

Number
  def 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 (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.

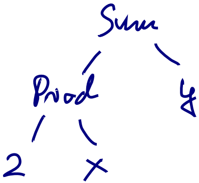
Example

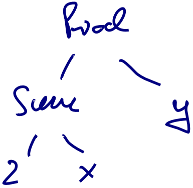
```
Sum(Prod(2, Var(”x”)), Var(”y”))
```

should print as “`2 * x + y`”. But

```
Prod(Sum(2, Var(”x”)), Var(”y”))
```

should print as “`(2 + x) * y`”.





2 +(x * y)

(2 + x) * y