THE COMPARISON

FP VS OOP

In FP (and procedural), programs break down into functions that perform some operations

- Functions may take one or more arguments

In OOP, programs break down into classes that give behavior to some kind of data

These two approaches are exactly opposite and provide complementary perspectives to the same problem

Which approach to take depends on how the software is planned to be extended

BASIC SET-UP

- Expressions for a small “language” such as for arithmetic
- Different variants of expressions, such as *integer expressions, negation expressions or addition expressions*
- Different operations over expressions, such as *evaluating them, converting them to strings*

	Evaluate	toString
Integer		
Negation		
Addition		

- Conceptual matrix must be populated indicating how the program would behave for each grid of the matrix

FUNCTIONAL APPROACH

	Evaluate	toString
Integer		
Negation		
Addition		

Define a *datatype for expressions, with one constructor for each variant*

- For languages with dynamic typing we don't need to name the data types

Define a *function for each operation*

In each function, have a branch (e.g., via pattern-matching) for each variant of data

- If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches

```
exception BadResult of string

datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

fun eval e =
    case e of
        Int _      => e
      | Negate e1  => (case eval e1 of
                           Int i => Int (~i)
                         | _ => raise BadResult "non-int in negation")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                           (Int i, Int j) => Int (i+j)
                         | _ => raise BadResult "non-ints in addition")

fun toString e =
    case e of
        Int i      => Int.toString i
      | Negate e1  => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2) => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"
```

	Evaluate	toString
Integer		
Negation		
Addition		

OBJECT ORIENTED APPROACH

Define a *class for expressions, with one abstract method for each operation*

Define a *subclass for each variant of data*

In each subclass, have a method definition for each operation

- If there is a default for many variants we can use a method definition in the superclass so that via inheritance we can avoid enumerating all the branches

This approach is data-oriented decomposition: breaking the problem down into classes corresponding to each data variant

```
abstract class Exp {
    abstract Value eval(); // no argument because no environment
    abstract String toStrng(); // renaming b/c toString in Object is public
}

abstract class Value extends Exp { }

class Int extends Value {
    public int i;
    Int(int i) {
        this.i = i; }
    Value eval() {
        return this; }
    String toStrng() {
        return "" + i; }
}

class Negate extends Exp {
    public Exp e;
    Negate(Exp e) {
        this.e = e; }
    Value eval() {
            return new Int(- ((Int)(e.eval())).i); }
    String toStrng() {
        return "-(" + e.toStrng() + ")"; } }
```

```
class Add extends Exp {
    Exp e1;
    Exp e2;
    Add(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    Value eval() {
        // we downcast from Exp to Int, which
will raise a run-time error
        // if either subexpression does not
evaluate to an Int
        return new Int(((Int)(e1.eval())).i +
((Int)(e2.eval())).i);
    }
    String toStrng() {
        return "(" + e1.toStrng() + " + " +
e2.toStrng() + ")"; }}
```

COMPARISON

	Evaluate	toString
Integer		
Negation		
Addition		

Functional decomposition breaks programs down into functions that perform some operation and object-oriented decomposition breaks programs down into classes that give behavior to some kind of data

Its about deciding whether to lay out a program “by column” or “by row”

This is needed in conceptualizing software or deciding how to decompose a problem

Various tools and IDEs help to view a program in a way different than how the code is decomposed

https://homes.cs.washington.edu/~djg/

	Evaluate	toString
Integer		
Negation		
Addition		

COMPARISON: CONTEXT OF THE EXAMPLE

It is “more natural” to have the cases for eval together rather than the operations for Negate together

For problems like implementing graphical user interfaces, the object-oriented approach is probably more popular

- It is “more natural” to have the operations for a kind of data (like a MenuBar) together (such as backgroundColor, height, and doIfMouseIsClicked rather than have the cases for doIfMouseIsClicked together (for MenuBar, TextBox, SliderBar, etc.)

EXTENSIBILITY

	Evaluate	toString	NoNegativeConstant
Integer			
Negation			
Addition			
Multiplication			

Adding a new operation is easy following functional approach

```
fun noNegConstants e =
    case e of
        Int i        => if i < 0 then Negate (Int(~i)) else e
      | Negate e1    => Negate(noNegConstants e1)
      | Add(e1,e2)   => Add(noNegConstants e1, noNegConstants e2)
```

Adding a new data variant, such as Mult of exp * exp is less pleasant

In OOP approach it is just the opposite

PLAN FOR UNPLANNED EXTENSIONS

Making software that is both robust and extensible is valuable but difficult

Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change(without breaking extensions)

	Evaluate	toString	NoNegativeConstant
Integer			
Negation			
Addition			
Multiplication			

New data types can be easily added through ??

New functions can be easily added through ??

- Higher Order functions-Other cases
- Visitor pattern

Extensibility-Double-edged sword

- In fact, languages often provide constructs exactly to *prevent extensibility*
- *Final* class cannot be extended

MORE EXTENSIONS TO THE PROBLEM

The methods may take two or more arguments or evaluating the expressions can be more complicated if different kinds of data are considered

Add can be modified as follows

- If the arguments are ints or rationals, do the appropriate arithmetic
- If either argument is a string, convert the other argument to a string (unless it already is one) and return the concatenation of the strings.

	Int	String	Rational
Int			
String			
Rational			

MORE EXTENSIONS TO THE PROBLEM

	Int	String	Rational
Int			
String			
Rational			

If many cases work the same way, we can use wildcard patterns and/or helper functions to avoid redundancy

One common source of redundancy is *commutativity*

- adding a rational and an int is the same as adding an int and a rational

```
Value eval() {
            return new Int(((Int)(e1.eval())).i + ((Int)(e2.eval())).i);
}
```

BINARY METHODS IN OOP

```
Value eval() {
        return e1.eval().add_values (e2.eval());
}
```

An Int, MyRational, or MyString should “know how to add itself to another value”

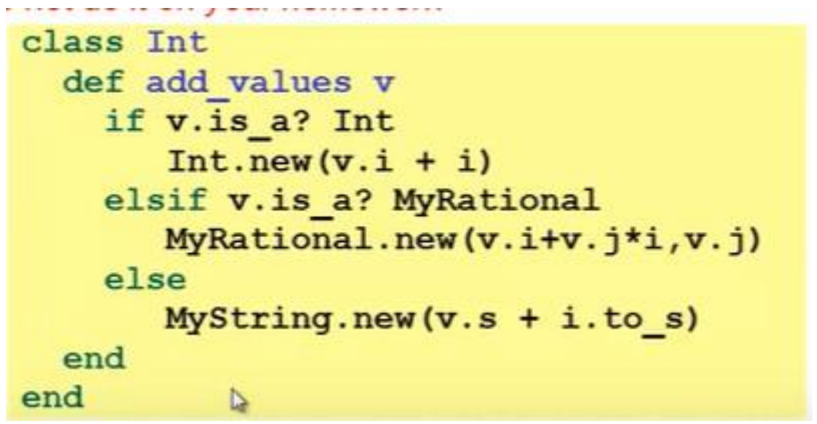
By putting add_values methods in the Int, MyString, and MyRational classes, the work is divided into three pieces using dynamic dispatch depending on the class of the object that e1.eval returns, i.e., the receiver of the add_values call in the eval method in Add

Double dispatch is the ability of dynamically selecting a method not only according to the run-time type of the receiver (single dispatch), but also to the run-time type of the argument (when all arguments are considered, we have multiple dispatch).

BINARY METHODS IN OOP

Adding new types, such as MyString and MyRational are quite easy in OOP

Call e1.eval().addValues(e2.eval())



```
class Int
  def add_values v
    if v.is_a? Int
       Int.new(v.i + i)
    elsif v.is_a? MyRational
       MyRational.new(v.i+v.j*i,v.j)
    else
       MyString.new(v.s + i.to_s)
  end
end
```

Combination of Object oriented and Racket style programming (conditional decomposition)

But then each of these three needs to handle three of the nine cases, based on the class of the second argument

- Any of addInt(..), addMyString(..), addMyRational(…) of Int class would be called if the first argument is Int

While this approach works, it is really not object-oriented programming. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using is_a? to figure out the cases in each method)

The extensibility advantage of OOP is lost!

DYNAMIC DISPATCH

```
e0.m(e1,…,en)
```

Dynamic dispatch – Also known as late binding or virtual methods –

- Call self.m2() in method m1 defined in class C can resolve to a method m2 defined in a subclass of C
- Most unique characteristic of OOP
- To implement dynamic dispatch, evaluate the method body with self mapping to the receiver (result of e0)
- That way, any self calls in body of m use the receiver's class
- Not necessarily the class that defined m

https://courses.cs.washington.edu/courses/cse341/19sp/lec21slides.pdf