Formally describe an artwork

For this activity, you will be describing two artworks, computationally.

The first artwork is "Homage to the Square: Warming" by Josef Albers (1959).

The second artwork is your choice. Your reasons for choosing an artwork are your own. We will also not engage in any critical interpretation. Instead, we limit ourselves to a "mechanical description" for each artwork.

Do the following:

1. Snap a photo with your smartphone.
2. Try to determine the set of words that can be used to describe the artwork, drawn from two categories:
   (a) primitives, and
   (b) combining forms.

   Be sure to define the words you use before you use them. Try to be precise; if you can be mathematical in your precision, even better. Ideally, you will produce a small set of F# `type`s that describe the things you see in an artwork.
3. Describe the artwork as completely as you can. If you find that your set of words is missing something, or if another word could be used to better describe the artwork, go back to step 2 and modify your set of words.

You may be wondering what is meant by primitive and combining form, so here are some definitions.

Primitives

A primitive is a type of data drawn from a (possibly infinite) set. It is atomic in the sense that it has no obvious constituent parts, or it can be defined in a way that it has no constituent parts. For example, an `int` is often taken to be a primitive in a programming language. Does it have constituent parts? Well, yes, in a way; from the machine's vantage point it can see and access an `int`'s individual bits. However, many programming languages (like Java) treat them as indivisible.

Combining Form

A combining form is a language element that describes some kind of composition. In typical programming languages, we often have two kinds of combining forms: those for data, and those for operations.

A combining form for data might be a `struct` in C, a `class` in Java, or a `type` in F#. A `class`, for instance, allows a user to combine multiple pieces of data (i.e., the class's `fields`) into a single piece of data. Many combining forms for data are recursive in the sense that they themselves are data, so that combining forms can combine other combining forms. For example, a class can have instances of classes in their fields.

There are two common combining forms for operations. An *instruction sequence* means "do this step and then the next step." You probably have not thought about the fact that a computer knows that it should move on to the next line in a program after it finishes running the current line, but a programming language actually has to tell the computer to do that. We often refer to this operation as `seq`. If you want a computer to perform a long list of operations, the effect is achieved by nesting `seq`s. And, of course, a *function* is a combining form that allows us to abstract over the data (i.e., the "formal parameters") used in an operation. Finally, combining forms for operations are usually recursive in the sense that they themselves are operations.

Example

As an example for this exercise, suppose that we want to describe a line. Is a line a primitive or a combining form? It depends on how we want to describe it. Suppose we decide that a line is going to be a combining form. How can we define it? Let's pick apart the idea of a line.

What do we need to describe a line? I learned in geometry class that a line is defined by its endpoints. What is an endpoint? One way of describing an endpoint is as a pair numbers, i.e., a coordinate. That means that we need numbers. Perhaps `Number` should be a primitive value in our language?

Let `Number` be a real number from $-\infty$ to $+\infty$, represented by the F# data type:

```
type Number = float
```

Let `Coordinate` be a combining form consisting of a pair of `Number`s, represented by the F# type:

```
type Coordinate = { x: Number; y: Number }
```

Let `Line` be a combining form consisting of a pair of `Coordinate`s, represented by the F# type:

```
type Line = { start: Coordinate; finish: Coordinate }
```

At some point, all of these pieces of data might be tied together using something like a `Canvas` type:

```
type Canvas =
| Lines of Line list
| // other things that can be on the canvas
```

All of the above types comprise the nodes needed to define an AST.

Focus on defining some terms to describe what you see. Once you have what you think comes close to defining all of the pieces of an artwork, try writing some F# code that generates an AST. No need to use a computer; just do this exercise on paper.

```
Lines (
  [
    {
      start  = { x = 3.0; y = 4.0 };
      finish = { x = 5.0; y = 5.0 }
    }
  ]
)
```

We will explore how parsers do their magic of converting strings into ASTs in an upcoming lecture. For example, a parser may allow us to write a sentence like the one below to generate an AST like the one above.

```
line starting at 3,4 and ending at 5,5
```