# Formally describe an artwork

In our last visit, we determined the set of words we needed to describe an artwork. For this visit we will concretely focus on the data structures we need to capture those ideas. Specifically, we will spend this time designing abstract data types (ADTs) for a language's abstract syntax tree (AST). It is important to spend a little time thinking about ASTs because an AST is the output of a parser:

```
let parse (s: string) : AST option = ...
```

For this activity, you will devise ASTs that computationally describe two artworks.

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

The second artwork is your own choice. Critical interpretation of art is not a part of this exercise. Instead, our goal is to devise a "mechanical description" for each artwork.

## Refresher on Primitives and Combining Forms

In an AST, every element that you define is something that your language can do. For example, think about the lambda calculus. In that language we are able to represent data using variables, define functions, and call functions. It is helpful to think of every AST type as being paired with a function that says what computational steps that type does. We call that function `eval`.

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 can be defined in a way that it has no constituent parts. For example, an `integer` 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 `integer`'s individual bits. However, from the vantage point of a programming language (like Java), it is often taken to be indivisible. When evaluated using `eval`, a primitive usually just returns itself. It does not do anything fancy.

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.

For example, a combining form for data might be a `struct` in C, a `class` in Java, or a `record` in F#. A `class`, for instance, allows a user to combine multiple data elements (i.e., the class's `fields`) into a single unit. Many combining forms are recursive. For example, a `class` can be recursive (e.g., we can use one to define a linked list), and a combining form for data is data, then combining forms can combine other combining forms.

A combining form for operations in many languages is a function definition. Function definitions represent substitution over operations; they make it possible for those operations to be reused with different data. Function definitions are also recursive, as functions can be built from other functions.

What a combining form does when `eval` is called really depends on the combining form. Classes usually allocate memory and store a collection of references (e.g., pointers) to the memory locations of their data elements. Function definitions and function calls are distinct combining forms. A function definition allocates a data structure in memory to make substitution operations efficient. A function call does the substitution and performs the operations.

## Worked 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 a little on how we want to use it. Do we want users of our language to be able to inspect its parts? That sounds like a reasonable thing, so let's make a line a combining form. How can we define it? Let's pick apart the idea of 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 with 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 }
```

Now we have a line. That's clearly useful; we can imagine using the above information to draw a line. Specifically, if we call `line`'s `eval` function, it should draw itself.

Of course, we usually want to draw more than one line. Perhaps we can represent the space where we might draw the lines:

```
type Canvas =
| Lines of Line list
| // other elements we might want on our canvas
```

When we call `eval` for `Canvas`, it draws all the lines it stores. This seems like a reasonable start for an AST for a line language.

Now that we know what a computer needs to represent an idea, we can invent syntax for our language. At this stage, you can let your imagination run wild, but remember: it should be possible for a machine to mechanically translate your syntax into instances of your AST. For example,

```
line starting at 3,4 and ending at 5,5
line starting at 1,1 and ending at 9,2
line starting at 3,4 and ending at 20,1
```

Can you write an AST constructor expression that generates the data structure implied by the program above? We will write parsers that do this step in the next class.

## Activity

Now, turn your attention to your two artworks. What do you see? Once you have what you think comes close to defining the most important pieces of an artwork as F# types, try writing some sample programs. Imagine that you have a parser that creates instances of those types using the information provided in the program.

Procedure:

1. Snap a photo with your smartphone.
2. As before, try to determine the set of words that can be used to describe the artwork. If you have notes from the last time we did this, you may reuse them. Words should be drawn from two categories:
   (a) primitives, and
   (b) combining forms.
3. Attempt to define your words as precisely as you can. If you can be mathematical in your precision, even better. The goal is to produce a small set of F# `type`s that describe the things you see in an artwork.
4. 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.
5. Generate some sample programs.