This is just like procedural languages, is there a more elegant way?

Blue -> [255, 0, 0]
Green -> [0, [255, 0, 0, 0, 0]
Red -> [255, 0, 0]

Let us define a function that maps color names to integer RGB codes:

```javascript
function colorToRGB(color) {
  switch (color) {
    case 'blue':
      return [255, 0, 0];
    case 'green':
      return [0, 255, 0];
    case 'red':
      return [255, 0, 0];
    default:
      return [0, 0, 0];
  }
}
```

For example:

```javascript
colorToRGB('blue') // [255, 0, 0]
```

Defining Your Own Types

General types are pronounced as "this type 1.

square :: Integer -> Integer
square n = n * n

You can also specify the type of a definition, while it is in a separate line:

```
square n = n * n
```

Since each expression has a type, each definition also has a type.

Types
A More Interesting Type

This is how you should write a function or read one:

```python
def function_name(parameters):
    # function body
```

or

```python
function_name(parameters)
```

Here's an example:

```python
def add_two_numbers(a, b):
    return a + b

result = add_two_numbers(3, 4)
print(result)  # Output: 7
```

More Examples of Functions

**More Examples of Functions (cont.)**

```python
# Python example

def multiply_numbers(a, b):
    return a * b

# Call the function
result = multiply_numbers(5, 3)
print(result)  # Output: 15
```

**Writing Functions for Your Types (cont.)**

```python
# Python example

def get_area(shape):
    if isinstance(shape, Ellipse):
        width, height = shape.width, shape.height
        area = pi * width * height
    elif isinstance(shape, Rectangle):
        width, height = shape.width, shape.height
        area = width * height
    else:
        raise ValueError("Unknown shape type")
    return area

# Example usage
ellipse = Ellipse(10, 5)
rectangle = Rectangle(8, 6)
area_ellipse = get_area(ellipse)  # Output: 157.08
area_rectangle = get_area(rectangle)  # Output: 48
```

**Smart Fraction**

```python
# Python example

def smart_fraction(numerator, denominator):
    if denominator == 0:
        raise ValueError("Denominator cannot be zero")
    return Fraction(numerator, denominator)

# Example usage
fraction = smart_fraction(1, 2)  # Output: Fraction(1, 2)
```

**Smart Fraction (cont.)**

```python
# Python example

def simplify_fraction(fraction):
    for factor in range(2, fraction.numerator + 1):
        if fraction.numerator % factor == 0 and fraction.denominator % factor == 0:
            fraction = Fraction(fraction.numerator // factor, fraction.denominator // factor)
    return fraction

# Example usage
simplified_fraction = simplify_fraction(Fraction(10, 20))  # Output: Fraction(1, 2)
```
in pattern matching.

But constructors and functions are different. E.g., constructors can use functions.

So real, green, blue are like functions requiring no parameters.

In fact you can use it as such.

rectangle Float Float Shape

The constructor is acting like a function.

rectangle 1.0 2.0 Shape

Consider again:

Constructor vs Function

A More Interesting Function

And the computer will display it.

rectangle 1.0 2.0

Now you can enter

This tells the computer, "Just display data of this type named."

def print(Shape)

data Shape = rectangle Float Float

Add a line, "display Shape at the end of the type definition."

A More Interesting Type (cont.)

...
A More Interesting Function of List

\[ \text{Area} = \text{sum of the areas in}\ x. \]

1. If the list is empty, return 0.
2. If the list is not empty, then it is like [x::xs], where x is the first element.

\[ \text{Area:xs} = \text{Area:xs} + \text{Area:xs}. \]

A Function of List

\[ \text{Accumulate} (x::xs) = \text{Accumulate} (x::xs) + x. \]

1. The sum of xs is of course, accumulator xs.
2. And xs is the rest of the list, where x is the sum of xs.

\[ \text{Accumulate} (x::xs) = \text{Accumulate} (x::xs) + x. \]

A Function of List

\[ \text{Accumulate} (x::xs) = \text{Accumulate} (x::xs) + x. \]

1. The empty list is, of course, accumulator xs.
2. And xs is the rest of the list, where x is the first element.

\[ \text{Accumulate} (x::xs) = \text{Accumulate} (x::xs) + x. \]