You’re Just My Type

Introductory diversion that may (or may not) have anything to do with this topic

Assume that I only allowed you to use **multiplication** and **addition**.

You are asked to create an arithmetic expression that is equivalent to some chosen number **k**, where **k** > 3.

You are only permitted to use the number 1 in your calculations.

How will you *most efficiently* arrive at **k**? Can you think of an algorithm that uses only multiplication, addition, and comparison operations?

Why do we use types?

In programming:

- Impose some kind of structure on “raw bytes” of memory
- Optimise data access by using different data structures
  - e.g. a list to store a lot of things of the same type, when you’re not sure how many things there will be
  - e.g. an array to store a lot of things of the same type, when you know how many things there are
- Create abstractions that are useful to work with
  - e.g. a dictionary to have have a “key” linked to some value
  - e.g. a queue where you can “push” from one side and “pop” off the other side
  - e.g. a stack where you can “push” and “pop” items from the top
  - etc

Why do we use types?

In mathematics:

- ~~Impose some kind of structure on “raw bytes” of memory~~
- ~~Optimise data access by using different data structures~~
  - ~~e.g. a list to store a lot of things of the same type, when you’re not sure how many things there will be~~
  - ~~e.g. an array to store a lot of things of the same type, when you know how many things there are~~
- Create abstractions that are useful to work with
  - e.g. a dictionary to have have a “key” linked to some value
  - e.g. a queue where you can “push” from one side and “pop” off the other side
  - e.g. a stack where you can “push” and “pop” items from the top
  - etc

When is an abstraction *useful*?

In a *mathematical* sense, what makes one abstraction more “useful” than another?

1. It is more useful if it can be **reused** for different problems
    a. Example: graphs, lists, trees, …
2. It is more useful if it has a **simple** definition
    a. Example: lists
3. It is more useful if we can **build** other abstractions from it
    a. Example: lists can be used to represent graphs or trees

We prefer having a **few** powerful abstractions, instead of many ineffective ones!

Fundamental idea: **range**

Every type has a **range**: a *number of representable values*.

Some types have a particular set of symbols that “work well” with that type, but this isn’t important: if two types *have the same range*, then we can perfectly represent one of them by using the other one.

Examples:

- Signed byte (values: -127 to 128)
- Byte (values: 0 to 255)
- 8-bit character (values: NUL to ÿ)

Unit type

Type name	**Range (*number of representable values*)**	**Examples**
Unit	1	()
Boolean	2	true, false
Byte	256	0, …, 125, …, 255
Integer	2^{32}	-2147483648, …, 2147483647
String	∞	…, “hello world!”, “”, …

Algebraic data types

The custom data-types that you can create in a functional language are called *algebraic data types*.

Algebraic data types come in two varieties:

- **Product** types
  - These let you *group values together*.
- **Sum** types
  - These let you create a *limited set of values*.

Why do these have such odd names?

Product Types

How many different values can you represent using...

- A unit?
- A boolean?
- A byte?
- A 32-bit integer?
- A string?

How many different values can you represent using…

- A unit?
- A boolean?
- A byte?
- A 32-bit integer?
- A string?

…if you had **two** of them?

How many different values can you represent using…

- A unit?
- A boolean?
- A byte?
- A 32-bit integer?
- A string?

…if you had ***n*** of them?

What if you had *different* types together?

How many different values can you represent using…

When we have a *group*, we multiply the ranges to get the number of representable values.

This is why data-types that *group* values are called **product** types.

In mathematics, we could write this as $(4, 8, 2.6)$

In functional programming, we could write this as **(4, 8, 2.6)**

You can also have named groups (similar to “structs” and “objects” in other languages). In F#, we call these “records”.

Record syntax

Record types are defined as follows:

```
type MyRecord =
  { SomeField : string
    Whatever: int
    Yolo : bool * int
  }
```

A value of a particular record-type is created by giving **every** field a value:

```
{ SomeField = "Hello"
  Whatever = 32
  Yolo = (true, 99)
}
```

Tip: Instead of whitespace and indentation, you can also use semicolons, if you would like to.

Sum Types

Primitive values as sets

Set name	Cardinality	Examples
Unit	1	()
Boolean	2	true, false
Byte	256	0, …, 125, …, 255
Integer	2^{32}	-2147483648, …, 2147483647
String	∞	…, “hello world!”, “”, …

How many values can you represent using...

- A set of {stapler, pencil, orange}?
- A set of {headphones, book}?
- A set of {paper}?

If you had to represent a traffic light with red, orange, and green lights, how would you do that?

Data types and sets

If you had to represent a traffic light with red, orange, and green lights,

1. How would you do that in Python?
2. How would you do that in Java?
3. What difficulties might you run into?

Sum type syntax

Sum types in F# are called “discriminated unions”. They are defined as follows:

```
type OfficeSupplies
  = Stapler
  | Pencil of lengthInCm : int
  | Paper
```

Each of the alternatives is called a **case**. You create an **OfficeSupplies** value by writing the case name and, if necessary, its arguments.

Examples:

- Paper
- Pencil 15

Designing Types

(and the utility of parametric polymorphism)

An inventor's problem

Imagine that you're part of a group of inventors.

Obviously, figuring out what it's possible to create is a core activity of inventors! But inventors might disagree about this. For example, some might think that we can make interstellar spaceships, and others might say that we can't.

To solve disagreements, they decide that if you say that something can be created, then you should also provide a **blueprint**—a plan for how to create it. If everyone sees what's needed to create it, and everyone agrees that the process of creating it works, then everyone will agree about what can be created.

Blueprints

What is a *workable blueprint*? Is it just a bunch of scribbles on paper? Surely not!

Blueprints have to list down

- What is needed for an invention;
- What process you need to perform with those things;
- What you expect to be the outcome.

Examples:

- steel AND leather ⇒ knife
- paper AND ink AND glue AND card ⇒ book

Blueprints

steel AND leather ⇒ knife

- You need **both** of these things; you can't make a knife with just one!
  - We'll use AND to show when we need everything
- The "⇒" is critically important. This is a description of what needs to be done with the steel and leather, to turn it into a knife!
  - Heating the metal, hammering it, forging it, cooling it, grinding it.
  - Then wrapping the handle with leather properly, so that the knife has a good grip and won't come loose

If everyone can see the process, and everyone can see the components, then everyone can see whether it can actually be done—and the inventors can start agreeing with each other!

Blueprint components

- Some components are basic and everyone knows about them and agrees about them
  - Steel, wood, helium, etc…
- Some components need to be made from other components
  - Hammer, wallet, lightbulb, etc…

You *should* be able to follow a component down to its basic elements, and thus agree about everything.

Substitute components

Sometimes we can substitute components.

- steel AND leather ⇒ knife
- steel AND wood ⇒ knife
- steel AND plastic ⇒ knife

We can say: steel AND (leather | wood | plastic) ⇒ knife

- We will use "|" to show this kind of alternation
- For convenience, we might want to say:

  handle-material = leather | wood | plastic

  steel AND handle-material ⇒ knife

Different technique → different outcome

If you apply a different process to the same components, you can get a different outcome!

- steel AND leather ⇒ knife
- steel AND leather ⇒ hammer
- steel AND leather ⇒ axe

Outcomes can vary too!

One process can produce many outcomes:

- steel ⇒ nails AND screws
- steel ⇒ screws AND metal-filings

One process can produce variations:

- paper AND ink AND (glue | thread) AND (card | board) ⇒ (hardcover-book | softcover-book)

Summary

- Blueprints describe how to *construct* something—and if we agree that the blueprint works, then the thing can be constructed!
- A blueprint can use many different components.
  - Components can be basic things e.g. wood or steel
  - Components can be groups of things
  - Components can be sets of substitutes
- Components can be transformed into different components by using processes.
- If all the processes work, then we will be able to construct the thing that the blueprint says we can construct!

Functionally speaking…

- **Programs** describe how to *construct* **a system**—and if we agree that the **program** works, then the **system** can be constructed!
- A **program** can use many different **types**.
  - **Types** can be basic things e.g. **int** or **string**
  - **Types** can be groups of things
  - **Types** can be sets of **cases**
- **Types** can be transformed into different **types** by using **functions**.
- If all the **functions** work, then we will be able to construct the **system** that the **program** says we can construct!

Notation for groups and sets

Groups:

- (3, "hello", 7.4, 9)
  - This is a “tuple”, of course
  - Each has an *arity*: a number of elements
  - Constructed by naming all the values

Sets:

- type Material = Wood | Leather | Plastic
  - This is a “discriminated union”—but you can call it a sum type
  - Constructed by naming the case (and argument(s), if necessary)
  - Here, **Material** is the name of the type, and **Wood**, **Leather**, and **Plastic** are the names of cases

Arbitrary combinations are possible! 🎉

- Tuples (“product types”) that include sum types
- Discriminated unions (“sum types”) linked to product types
- Type notation
  - Tuples: read the “*****” as “AND” *functional programmers say “by”, not “AND”, but they mean the same thing*
  - A tuple value is named by arity and type of its contents, from left to right
    - e.g. **("test", 100, 0.53)** is a “3-tuple of string and int and float”
  - A discriminated union value is named by the set name and *sometimes* the case name
    - e.g. **Wood** from **type Material = Wood | Leather | Plastic** is a “Material” or “Wood Material”
  - Functions: read the “->” as “to”
  - A function is named by its types
    - e.g. **(fun x -> x % 2 = 0)** is a “int to bool”

Back to inventors

When specifying a blueprint, we know that it is useful to have groups, e.g. **steel AND leather**.

It is also useful to have sets of substitutes, e.g. **leather | plastic | wood**.

And it is useful to have groups that include substitutes: **steel AND (leather | plastic | wood)**

Functionally speaking…

When specifying a **program**, we know that it is useful to have **product types**

- e.g. **int * string**.

It is also useful to have sets of **cases**

- e.g. **type HandleMaterial = Leather | Plastic | Wood**.
- (in our language, case-names and sum types must always start with a capital letter.)

And it is useful to have **product types** that include **sum types**:

- **int * HandleMaterial**

Back to inventors

It is useful to have substitutes that include properties and even groups of properties:
(wood *of* hardness AND color | leather *of* animal | plastic)

We can combine these in any ways that we want to!
steel AND (wood *of* hardness AND color | leather *of* animal | plastic)

Functionally speaking…

It is useful to have **sum types** that include ***basic types*** and even ***product types***:

- **type HandleMaterial =**
  | Wood *of* hardness:int * color:string
  | Leather *of* animal:string
  | Plastic
- If we define a case that has some property attached to it, then we must give it that property

We can combine these in any ways that we want to!
e.g. **(8, Wood (7, "dark brown"), 99.0, Leather ("cow"))**

Summary

- **Algebraic Data Types** are *product types* and *sum types*
- **Tuples** are **product types**, and have an **arity**.
  - Programming language notation: **(t_0 * t_1 * …)**
- **Discriminated unions** are **sum types**.
  - They are a **set** of **cases**. Each **case** is a member of the **set**.

(What about functions?)

How do we use Algebraic Data Types in our functions?

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

You have already been using patterns all along—you just didn't know that it was called a "pattern"!

```
fun (tax, price) -> price * tax
```

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

You have already been using patterns all along—you just didn't know that it was called a "pattern"!

```
fun (tax, price) -> price * tax
```

This is called a **tuple pattern**. It *verifies that the tuple is a 2-tuple*; and it binds the values in a tuple to different names.

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

You have already been using patterns all along—you just didn't know that it was called a "pattern"!

```
fun (tax, price) -> price * tax
```

This is called a **tuple pattern**. It *verifies that the tuple is a 2-tuple*; and it binds the values in a tuple to different names.

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

Let's say that we want a function that takes in a 2-tuple, and just gives us back first item.

```
fun (a, b) -> a
```

This is called a **tuple pattern**. It *verifies that the tuple is a 2-tuple*; and it binds the values in a tuple to different names.

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

Let's say that we want a function that takes in a 2-tuple, and just gives us back first item. We don't actually need a name for the second item—in fact, we can ignore it totally, while still verifying the "2-tuple" *structure*!

```
fun (a, _) -> a
```

This is called a **tuple pattern**. It *verifies that the tuple is a 2-tuple*; and it chooses which values we want to bind; and it binds a value in a tuple to a name.

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

Let's say that we want a function that takes in a 2-tuple, and just gives us back first item. We don't actually need a name for the second item—in fact, we can ignore it totally, while still verifying the "2-tuple" *structure*!

```
fun (a, _) -> a
```

Using the _ pattern is *optional*. If you want to bind names and never use them … go ahead 🤷‍♀️!

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

```
type HandleMaterial =
| Wood of hardness:int * color:string
| Leather of animal:string
| Plastic
```

What about sum types?

Every case in a sum type could require different handling. How can we handle each case?

Answer: using *functions*! There is a special syntax for functions that do exactly this.

Patterns 🤝 Algebraic Data Types (ADTs)

A **pattern** is a way of **binding** the values of an ADT to **symbols**. It lets us:

1. Verify that a type has a particular structure, and
2. Choose *which values* we want to bind, and
3. Give each of those values a name

```
type HandleMaterial =
| Wood of hardness:int * color:string
| Leather of animal:string
| Plastic
```

Every case in a sum type could require different handling. How can we handle each case?

Answer: using *functions*! There is a special syntax for functions that do exactly this.

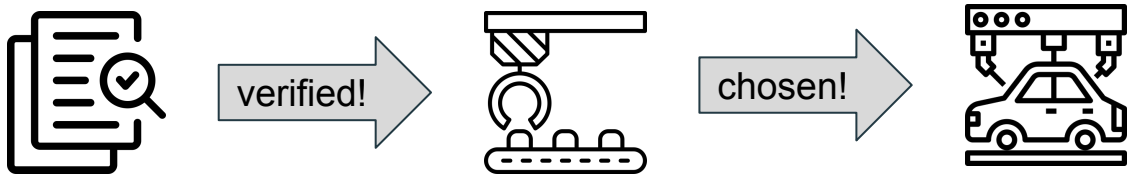
```
function
| Wood (h, c) -> "using wood"
| Leather (what) -> "leather from " + what
| Plastic -> "using plastic"
```

Manufacturers 🤝 Blueprints

A **manufacturer** must **link** physical **items** to **component parts**. It must:

1. Verify that parts needed for a component are available

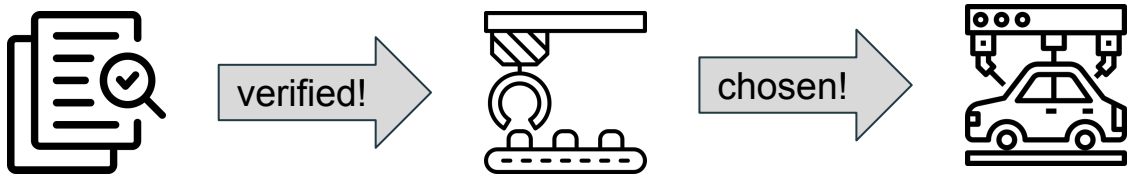
2. Decide which physical items are needed in a particular step
   a. If a particular physical item isn't needed right now, then we can ignore it
   b. It is *still in the group*, but we don't need it right now
3. Choose only those items to work with



Manufacturers 🤝 Blueprints

A **pattern** will **bind** runtime **values** to **symbols**. It must:

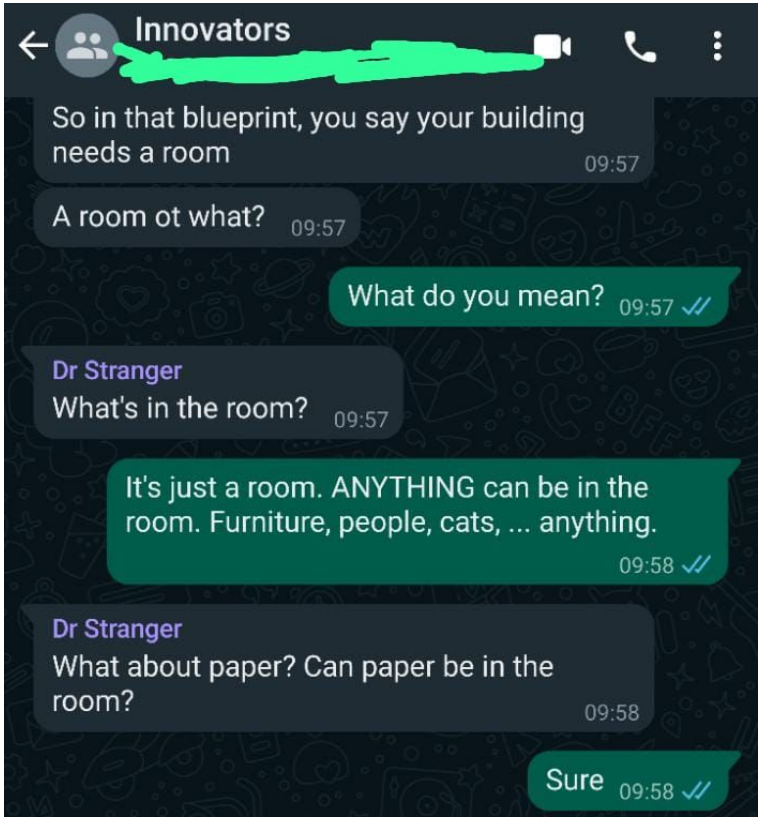
1. Verify that this values has the structure specified by the type
   a. If this is a *product type* (e.g. “tuple”), then check the types, arity, etc.
   b. If this is a *sum type*, then check the cases, number of linked values, etc
2. Select only the values which are needed right now
   a. If a particular value isn’t needed right now, then we can ignore it
   b. It is *still in the group*, but we don’t need it right now
3. Bind only those values to work with

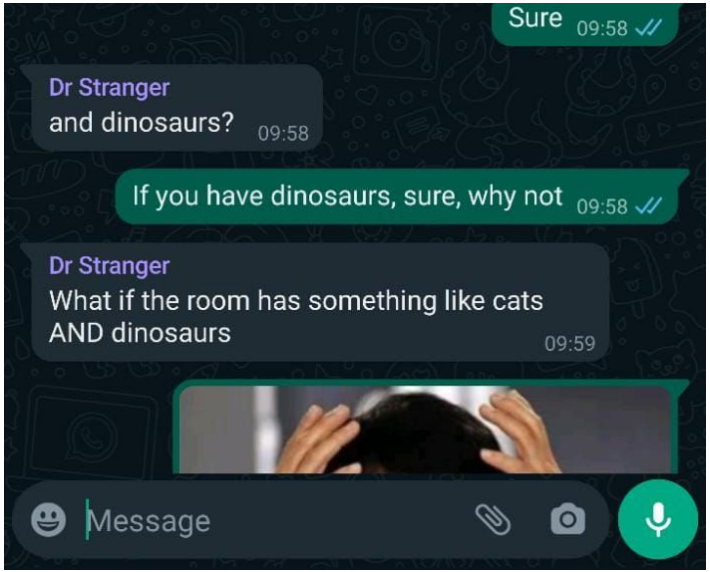


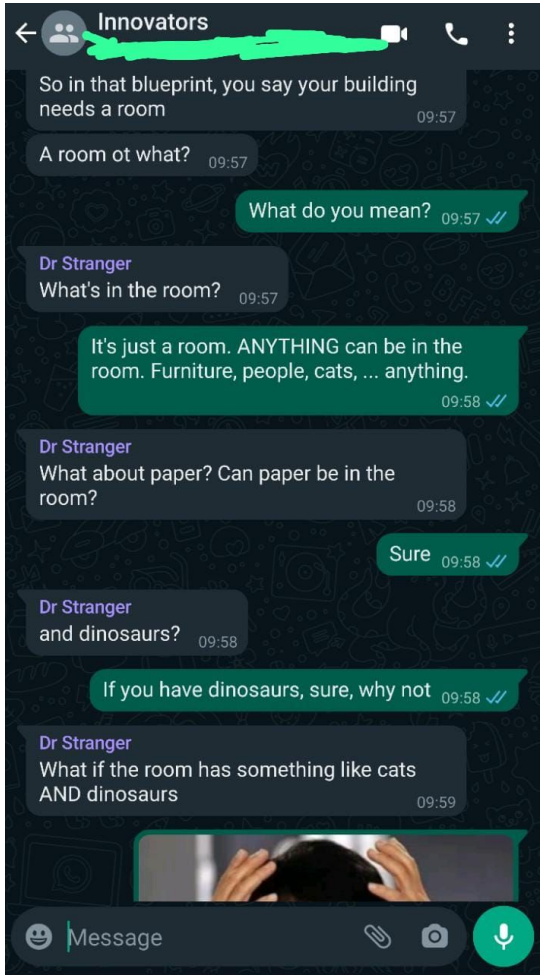
Invent=Develop, Manufacture=Execute

- The inventor creates blueprints
- The developer creates a program
- This 👆🏾 is what happens during **development**!
- The inventor gets their plans approved
- The developer compiles their program
- This 👆🏾 is what happens at **compile time**!
- The manufacturer takes the *blueprints* and *manufactures* them using *physical items*
- The machine takes the *program* and *executes* it using *actual values*
- This 👆🏾 is what happens at **runtime**!

Dealing with anything


Innovators
So in that blueprint, you say your building needs a room 09:57
A room ot what? 09:57
What do you mean? 09:57
Dr Stranger
What's in the room? 09:57
It's just a room. ANYTHING can be in the room. Furniture, people, cats, ... anything. 09:58
Dr Stranger
What about paper? Can paper be in the room? 09:58
Sure 09:58


Sure 09:58
Dr Stranger
and dinosaurs? 09:58
If you have dinosaurs, sure, why not 09:58
Dr Stranger
What if the room has something like cats AND dinosaurs 09:59
Message



Inventing polymorphism

Even the best of blueprints can be misunderstood!

How do we solve Dr Stranger's problem?

Polymorphism: “anything goes here!”

“Polymorphism” is the ability to specify *anything* as part of a component.

Important: **anything** cannot be **nothing**.

You have to have *something* in the “room”. The only thing you can choose is what that “something” will be.

Polymorphism: “anything goes here!”

“Polymorphism” is the ability to specify *anything* as part of a component.

Advantages:

1. You can put *anything* there!
   - “A room of anything”, “A container of anything”, etc
2. A manufacturer can use *different anythings*—and they *can both be the same*!
   - “Furniture is a cupboard of anything1, and a drawer of anything2”
   - If anything1 = clothes and anything2 = clothes, that’s okay
   - If anything1 = tools and anything2 = papers, that’s okay too!
3. You can use “anything” *and* specific things, in the same component
   - “A dressing table has a mirror, AND drawers containing anything”

Polymorphism: “anything goes here!”

“Polymorphism” is the ability to specify *anything* as part of a component.

Disadvantages:

1. Anything means *anything*: you give up the ability to make **any assumptions** (e.g. weight, color, …)
2. *After* something has been put there, you cannot act as though something else has been put there
    a. A cupboard of clothes will always remain a cupboard of clothes—it cannot magically change into a cupboard of tools.

Polymorphism: “anything goes here!”

“Polymorphism” is the ability to specify *anything* as part of a type.

Advantages:

1. You can put *anything* there!
   - “A list of anything”, “A stack of anything”, etc
2. At runtime, one can have *different anythings*—or they *can both be the same*!
   - “A pair has a first-item of anything1, and a second-item of anything2”
   - If anything1 = int and anything2 = string, that’s okay
   - If anything1 = float and anything2 = float, that’s okay too!
3. You can use “anything” *and* specific things, in the same type
   - “An error has a string description, AND error data which can be anything”

Polymorphism: “anything goes here!”

“Polymorphism” is the ability to specify *anything* as part of a type.

Disadvantages:

1. Anything means *anything*: you give up the ability to make **any assumptions** (e.g. size, operations, …)
2. *After* something has been put there, you cannot act as though something else has been put there
    a. A list of string will always remain a list of string—it cannot magically change into a list of bool.

Polymorphism, formally

A polymorphic type is any type that includes a **type variable**.

In F#, all type variables are written as either **'something** or (*much more rarely*) as ^something .

In type names, type variables are included in <> characters as part of the type name.

```
type TwoThings<'a, 'n> =
  { oneThing : 'n
  , anotherThing : 'a * int
  }
```

Everywhere else, a type variable is written without <> characters.

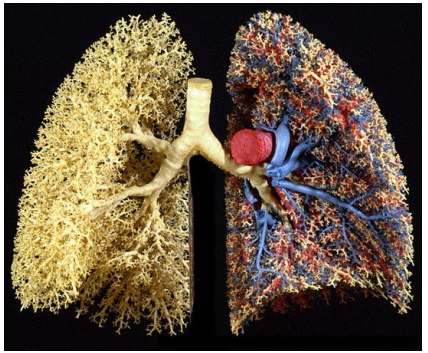
```
fun x y -> (y, 8, x)
```

has the type

```
'a -> 'b -> 'b * int * 'a
```

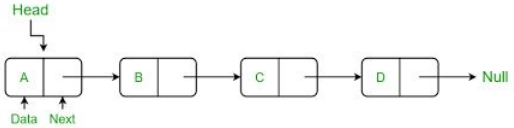
Recursive data types

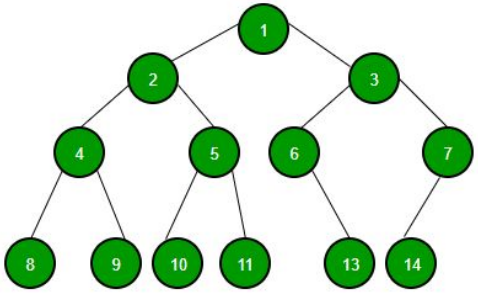
Self-similarity is everywhere ❤️



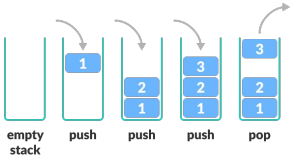
Self-similarity is everywhere ❤️

- Trees
- Linked-lists
- Stacks
- Programming languages
- …



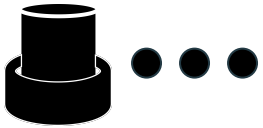


Recursive data structures are simple and powerful.



Our First Data Structure: An Integer List

```
type IntList =
| Empty
| Data of int * IntList
```

- A list is either empty… 
- …or it has some data, followed by the rest of the list 

Our First Data Structure: An Integer List

```
type IntList =
| Empty
| Data of int * IntList
```

- Here is an empty list:

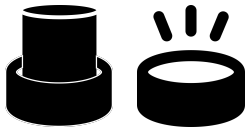
  Empty

Our First Data Structure: An Integer List

```
type IntList =
| Empty
| Data of int * IntList
```

- Here is a list with one item

  Data (900, Empty)

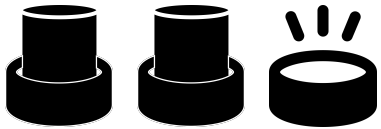


Our First Data Structure: An Integer List

```
type IntList =
| Empty
| Data of int * IntList
```

- Here is a list with two items

  Data (400, Data (900, Empty))

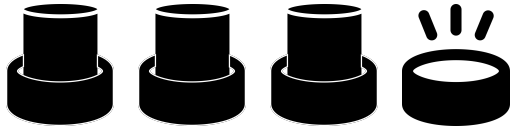


Our First Data Structure: An Integer List

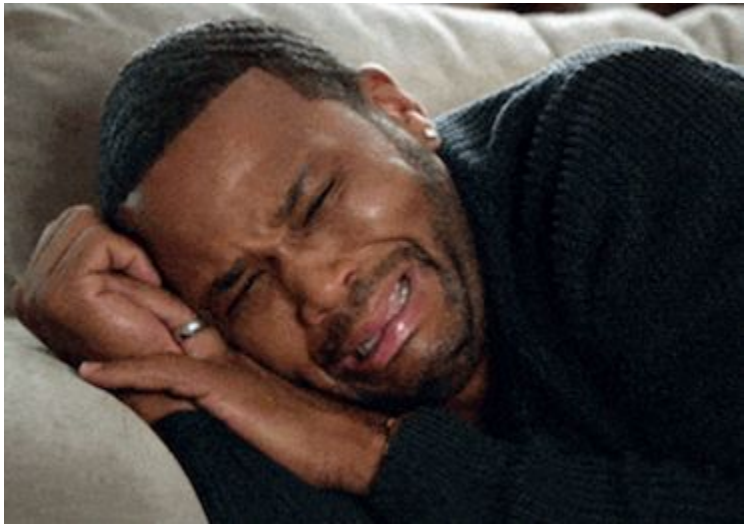
```
type IntList =
| Empty
| Data of int * IntList
```

- Here is a list with three items

Data (103, Data (400, Data (900, Empty)))



How do we look at all the items??



How do we look at all the items??

If only there was a way to look at one thing…

…and then look at the next…

…and then look at the next…

…until we get to the end…

How do we look at all the items??

Recursion + Patterns = 

```
let rec sumItems items =
  (function
    | Empty -> 0
    | Data (item, nested) -> item + sumItems nested
  ) items
```

Parametric Polymorphism

```
type MyList<'anything> =
| Empty
| Data of 'anything * MyList<'anything>
```

Parametric Polymorphism

```
type MyList<'anything> =
| Empty
| Data of 'anything * MyList<'anything>
```

Parametrically polymorphic types usually work very well with complementary functions. In most functional languages, for example, one will find the function called “cons”:

let cons = fun element list -> Data (element, list)

One also finds “head”, which gives you the first element in a list, and “tail”, which gives you all the other elements.

Usually, there is also a function called “concat” or “join”, which joins two lists together.

Built-in List<'T> type

```
type MyList<'anything> =
| Empty
| Data of 'anything * MyList<'anything>
```

The built-in List<'T> type is defined in *exactly the same way* as our custom-built MyList<'T> 👆.

However, because lists are so commonly used, there is *special syntax* for deconstructing and constructing lists.

You can construct a list by placing elements, separated by a semicolon, in [ ] brackets:

- [] creates an empty list
- [9] is a list with 1 element
- ["hi"; "there"] is a 2-element list

You can construct a list by “cons”ing an element to another list via the “cons” operator (two colons):

- 7 :: [] is a 1-element list
- 3 :: [9] is a 2-element list

You can join two lists with the “concat” operator (an @ sign):

- [9; 2] @ [1; 0] = [9; 2; 1; 0]

Built-in List<'T> type

```
type MyList<'anything> =
| Empty
| Data of 'anything * MyList<'anything>
```

The built-in List<'T> type is defined in *exactly the same way* as our custom-built MyList<'T> 👆.

However, because lists are so commonly used, there is *special syntax* for deconstructing and constructing lists.

List-patterns are:

- [] matches an empty list
- [x] matches a 1-element list
- [a; b] matches a 2-element list
- a::rest matches the **head** and **tail** of a list
- a::b::rest matches two elements at the start of a list, and the tail after them.