Computer Science is not

- Soldering tiny things together.
- Programming for heros.
- Becoming a better gamer.
- Revitalizing a bricked phone.

... but it helps for those too.

Computer Science is

Philosophy / Logic?
The Discipline

Computer Science

is

Philosophy / Logic?

In a way, with a strong emphasis on the way how logic can be computed and how it can be used to prove that something will compute. ... so it is certainly related.

... but it is related.

Computer Science

is

Mathematics?

In a way, yet computer scientists emphasize on "computing" their theories and are less concerned about e.g. infinity or the infinitesimal. Some areas of discrete mathematics are directly employed though. ... but it is related.

Computer Science

is

Engineering?
**The Discipline**

**Computer Science**

is

**Science?**

Not really, as computer science is mostly neither “approximate” nor based on differential calculus. ... but it is related.

Directly relevant science faculties: Physics, Biology, Psychology, Sociology, and others
Solving simple to highly complex problems practically as well as theoretically sound.

- Can we reason about our systems (in an unambiguous/logical sense)?
- Can we build on a solid foundation?

Levels of abstractions

Logic – Mathematics – Computer Science – Hardware

Abstraction

Implementation

Design, Verification, Development, Experimentation
Problem solving

Say: … addition

\( a + b \), where \( a \) and \( b \) are:

- Values as natural numbers, integers, real numbers, vectors, matrices?
- Physical objects, conceptual objects, abstract symbols?
- Electrical charges or currents?
- Databases?
- Processors?
- … a few million other things which may need adding.

[Page 19]

Problem solving

Define a specific ‘+’ by means of an algebra

Symbols (defining base objects and notations):

- e.g. Vectors over real numbers, written as:
  \[
  \begin{pmatrix}
  1.0 \\
  0.0 \\
  1.0
  \end{pmatrix},
  \begin{pmatrix}
  0.1 \\
  0.2 \\
  0.1
  \end{pmatrix}
  \]

Operators (generating new objects):

- e.g. \( a_1 + b_1 = a_1 + b_1 \)
  \[
  \begin{pmatrix}
  a_1 \\
  b_1
  \end{pmatrix},
  \begin{pmatrix}
  a_2 \\
  b_2
  \end{pmatrix}
  \]

[Page 18]

Problem solving

Things in computer science

Data – Programs

Data (hardware view):
- Bits.
- Words (collections of bits).
- Many words.
- Many, many, many words.

Programs (hardware view):
- CPU instructions.
- Lists of CPU instructions.
- Long lists of CPU instructions.
- Loooooong lists of CPU instructions.

While we can define and use ‘+’ for bits and words, it is very hard to design anything of noteworthy complexity on this abstraction level.

[Page 20]
Data abstraction types

By grouping bits and words while following a certain underlying structure (“type”), we arrive at structured data which can then be treated as a new entity.

These new entities are then used themselves to be grouped into further, even more interesting data structures.

Example:

We group 64 bits by following this structure:

$$\text{Sign} \begin{cases} 0 & \text{for positive} \\ 1 & \text{for negative} \end{cases} \text{ Exponent} + \frac{2}{1} \cdot \text{Fraction}$$

and interpret those bits as:

$$(-1)^\text{Sign} \left( \frac{2^{\text{Exponent}} + 1}{2} + \sum_{i=1}^{52} 2^{-i} \cdot \text{Fraction} \right)$$

We call it a Long_Float:

and spend the rest of the day to define a ‘+’ on this new type Long_Float.

Example:

We group 64 bits by following this structure:

$$\text{Sign} \begin{cases} 0 & \text{for positive} \\ 1 & \text{for negative} \end{cases} \text{ Exponent} + \frac{2}{1} \cdot \text{Fraction}$$

and interpret those bits as:

$$(-1)^\text{Sign} \left( \frac{2^{\text{Exponent}} + 1}{2} + \sum_{i=1}^{52} 2^{-i} \cdot \text{Fraction} \right)$$

We can now refer to the whole vector with one symbol, yet for even better readability in some cases we will also name the three parts of a Long_Float_Vector in a meaningful way, i.e., x, y, z.

We can now swiftly define a ‘+’ on this new type Long_Float_Vector:

$$x + y = \begin{cases} x_1 + y_1 & \text{if } x_1 \geq y_1 \\ y_1 + x_1 & \text{otherwise} \end{cases}$$

while the ‘i’ on the right are the ones which we defined on the Long_Floats before.
Example:

We then group three of these Long_Floats to form a 3-dimensional vector:

\[ \begin{align*}
&v_1 + v_2 = \left\{ \frac{x_1 + y_1}{x_1 + z_1} \right\} + \left\{ \frac{x_2 + y_2}{x_2 + z_2} \right\} = \left\{ \frac{x_1 + x_2}{x_1 + z_1} + \frac{y_1 + x_2}{x_2 + z_2} \right\} \\
&v_3 + v_4 = \left\{ \frac{x_3 + y_3}{x_3 + z_3} \right\} + \left\{ \frac{x_4 + y_4}{x_4 + z_4} \right\} = \left\{ \frac{x_3 + x_4}{x_3 + z_3} + \frac{y_3 + x_4}{x_4 + z_4} \right\}
\end{align*} \]

while the '4's on the right are the ones which we defined on the Long_Floats before.

We can now refer to the whole vector with one symbol, yet for even better readability in some cases we will also name the three parts of a Long_Float_Vector \( v \) in a meaningful way, i.e.: \( x, y, z \).

We can now swiftly define a '+' on this new type Long_Float_Vector:

\[ v_1 + v_2 = \left\{ \frac{x_1 + y_1}{x_1 + z_1} \right\} + \left\{ \frac{x_2 + y_2}{x_2 + z_2} \right\} = \left\{ \frac{x_1 + x_2}{x_1 + z_1} + \frac{y_1 + x_2}{x_2 + z_2} \right\} \]

while the '4's on the right are the ones which we defined on the Long_Floats before.

There are three types each with value range of \( 2^{64} \).

Example:

Another data structure example: A mixture of base colours (Red, Green, Blue) is represented by the set of base colours which is switched on.

There are three valid values for the basic colours themselves (Red, Green, Blue), and each colour can be either 'on' or 'off' (two valid values):

```
Red | Base colours
---|----------------
Red | Green
Red | Blue
Green| Blue
```

Definied as a data structure: \( \text{type Mixture} ::= \text{array} \ (\text{Base_Colours}) \rightarrow \text{On_Off} \)

Definied as a function: \( \text{type Mixture} ::= \text{function} \ (\text{Base_Colours}) \rightarrow \text{On_Off} \)
Algebras in Computer Science

Next abstraction level

Algebra on types

So far we can add two (same type) values out of those:
- Real numbers of certain precision.
- Vectors of a those.
- Base colours.

While it seems silly to add a colour value to a vector value:

Could we instead add the types vector and colour?

If we can define operations on types themselves, they will become algebraic entities themselves!

Adding two types?

What would you expect for the number of possible values if we “add” the type `Long_Float_Vector` and the type `Mixture`?

In practice we would probably use the type `Boolean` here as we can then employ all predefined operations from the Boolean algebra.

In practice we would probably use the type `Boolean` here as we can then employ all predefined operations from the Boolean algebra.
Algebras in Computer Science

Next abstraction level

Multiplying two types?

What would you expect for the number of possible values if we “multiply” the type `Long_Float_Vector` and the type `Mixture`?

\[
\text{Long_Float} \times \text{Long_Float} = 2^{64} \times 2^3 = 2^{67} \]

… we are getting ahead of ourselves: Let's start again somewhat simpler.

Algebras in Computer Science

Algebraic types foundations

A ‘One’ type

```haskell
data One = Unit
```

This type has exactly one value which is called “Unit” here.

(Haskell has also a built-in type called `()` with the only value of `()`.)

How can we now define ‘+’ (addition) on types, such that:
- the number of possible values which the resulting type can hold is exactly:
- the sum of the possible values of each type?)
Algebraic types foundations

Let's make a 'Two' type

type Two = One :+ One
using the definition of '+', Two translates into:
Left_Alternative One | Right_Alternative One

Unsurprisingly the type Two can hold two different values.

Classical definition with meaningful names:
data Boolean = True | False

Adding two types: type a '+' type b

data a :+: b = Left_Alternative a | Right_Alternative b

The addition of two types results in a type which can hold
either values of the first type or values of the second type (but not both at the same time).

Let's add one to a type

type Add_One a = a :+ One
using the definition of '+', Add_One a translates into:
Left_Alternative a | Right_Alternative One

Unsurprisingly the resulting type can hold one more value.

Classical Haskell definition with meaningful names:
data Maybe a = Nothing | Just a

This is highly useful if you need to define a variable which holds values of a
type (like say: a vector) but occasionally there is no numeric value available
and we need to express that this vector is currently "not available".

Adding one more possible value to a existing type gives you just this option.
Let's add one to a type

type Add_One a = a :+ One

using the definition of '+', Add_One a translates into:

Left_Alternative a | Right_Alternative One

Unsurprisingly the resulting type can hold one more value.

Classical Haskell definition with meaningful names:

data Maybe a = Nothing | Just a

This is highly useful if you need to define a variable of a type (like a vector), but occasionally there is no numeric value available and we need to express that this vector is currently "not available".

Adding one more possible value to a existing type gives you just this option.

The Haskell constructor name Nothing is a somewhat unfortunate choice and should not be confused with an actual "void" type.

Maybe rather memorize this constructor as a "special value" to lessen the confusion.

How can we now define '*' (multiplication) on types, such that:

- the number of possible values which the resulting type can hold is exactly:
- the product of the possible values of each type?

Multiplying two types: type a '*' type b

data a :* b = Product a b

The product of two types is the type which can hold values of the first type and values of the second type (at the same time).

Unsurprisingly the resulting type can hold the product of the possible values of each type.

type Tuple_2 a = a :* a
-- a * a

type Tuple_3 a = a :* a :* a
-- a * a * a

For readability and ease of access, the individual components are commonly given names:

data Vector_3D a = Coordinates {x, y, z :: a}
-- a * a * a

instead of

data Vector_3D a = Coordinates a a a
-- a * a * a

How can we now define '^' (exponentiation) on types, such that:

- the number of possible values which the resulting type can hold is exactly:
- the possible values of one type to the power of the possible values of other type?
Algebraic types foundations

$(Type \ b) \ (Type \ a)$

\[
data b :^a a = \text{Function} \ (a \rightarrow b)
\]

How many side-effect-free functions with domain $a$ and range $b$ are there?

\[
data b :^a a = \text{Array} \ a \ b
\]

read: “array (a) of b”

How many side-effect-free functions with domain $a$ and range $b$ are there?

\[
x : \text{Boolean} \rightarrow \text{Boolean}
\]

$x, y : \text{Boolean} \rightarrow \text{Boolean}$

$2^2 = 4$ functions: $\{0, 1, x, \neg x\}$
Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Function (a -> b)
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

We could now also define the type `Mixtures` simply as:

```
Mixtures = Boolean :^ Base_Colours
```

---

Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Array a b
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean ⇒ functions:\{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top, \bot, 0\}
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

---

Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Array a b
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean ⇒ functions:\{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top, \bot, 0\}
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

---

Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Array a b
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean ⇒ functions:\{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top, \bot, 0\}
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

---

Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Array a b
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean ⇒ functions:\{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top, \bot, 0\}
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

---

Algebraic types foundations

(Type b) (Type a)

```
data b :^ a = Array a b
```

How many side-effect-free functions with domain `a` and range `b` are there?

```
x : Boolean -> Boolean ⇒ functions: {0,1,x,\top}
x,y: Boolean -> Boolean ⇒ functions:
\{0,1,x,y,\top,\bot\} \land y, x \land \top, x \lor \top, x \lor y, x \lor \bot, x \lor \top, x = y, x \land y\}
Base_Colours -> Boolean ⇒ functions:\{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top\}, \{\bot\}, \{\top, \bot\}, \{\top, \bot, 0\}
```

There are \( | \{ | \) side-effect-free functions with domain `a` and range `b`.

\( | \{ | \) denotes the cardinality or "number of possible values" of types `a` and `b`.

---
Algebraic type symbols and operators so far:

- **data** `One = Unit` -- also predefined in Haskell as `()`
- **data** `a :+: b = Left_Alternative a | Right_Alternative b`
- **data** `a :+: b = Product a b`
- **data** `b :^ a = Function (a -> b)`

Which directly produces most types used in practice – here some common basic types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Produced by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Two values</td>
<td><code>One :+: One</code></td>
</tr>
<tr>
<td>Enumerations</td>
<td>An enumeration of values</td>
<td><code>One :+: One :+: One :+: One ...</code></td>
</tr>
<tr>
<td>Range types</td>
<td><code>n</code> values</td>
<td>A function adding <code>One</code> for <code>n</code> times</td>
</tr>
<tr>
<td>Maybe types</td>
<td>Type with one special, extra value</td>
<td><code>a :+: One</code></td>
</tr>
<tr>
<td>Either types</td>
<td>Values out of multiple types</td>
<td><code>a :+: b :+: c ...</code></td>
</tr>
<tr>
<td>Tuples</td>
<td>Multiple types interpreted as one</td>
<td><code>a :+: a :+: a ...</code></td>
</tr>
<tr>
<td>Records</td>
<td>Different types interpreted as one</td>
<td><code>a :+: b :+: c ...</code></td>
</tr>
<tr>
<td>Functions</td>
<td>Side-effect-free functions</td>
<td><code>b :^ a</code></td>
</tr>
</tbody>
</table>

Your programming language will provide more convenient syntax to define these types.

Algebraic types foundations: Recursive types

**Using a type to define itself**

Let's play and add itself:

- **Self_Reference_1** `a = a :+: (Self_Reference_1 a)`
  
  \[ a :+: S(a) \]

  What does that mean?

- **Self_Reference_2** `a = a :+: (Self_Reference_2 a)`
  
  \[ a :+: S(a) \]
Algebraic types foundations: Recursive types

Using a type to define itself

Let's play and add itself:

\[
\text{Self}_{\text{Reference}_1} \ a = a \cdot (\text{Self}_{\text{Reference}_1} a) \Rightarrow a + S(a)
\]

or multiply with itself:

\[
\text{Self}_{\text{Reference}_2} \ a = a \cdot (\text{Self}_{\text{Reference}_2} a) \Rightarrow a \cdot S(a)
\]

And what happens if we add/multiply one while we are at it:

\[
\begin{align*}
\text{Self}_{\text{Reference}_3} \ a &= 1 \cdot (a \cdot (\text{Self}_{\text{Reference}_3} a)) \\
\text{Self}_{\text{Reference}_4} \ a &= 1 \cdot (a \cdot (\text{Self}_{\text{Reference}_4} a)) \\
\text{Self}_{\text{Reference}_5} \ a &= 1 \cdot (a \cdot (\text{Self}_{\text{Reference}_5} a)) \\
\text{Self}_{\text{Reference}_6} \ a &= 1 \cdot (a \cdot (\text{Self}_{\text{Reference}_6} a)) \\
\text{Self}_{\text{Reference}_7} &= 1 \cdot (\text{Self}_{\text{Reference}_7}) \\
\text{Self}_{\text{Reference}_8} &= 1 \cdot (\text{Self}_{\text{Reference}_8})
\end{align*}
\]

Legend:

\[
\begin{align*}
+: & \text{ add} \\
*: & \text{ multiply}
\end{align*}
\]

Which ones produce structures of practical value?

Lists

Verbal definition: A list is either empty or it holds an element followed by a list – or formally:

\[
\text{data} \ List \ a = \text{Alg}_{\text{List}} (\text{One} :+ (a :* (\text{List} a)))
\]

Unrolled and with some more meaningful constructor names:

\[
\text{data} \ List \ a = \text{Empty}_{\text{List}} | \text{Attach} \ a \ (\text{List} a)
\]
Algebraic types foundations: Recursive types

Lists

Verbal definition: A list is either empty or it holds an element followed by a list – or formally:

```haskell
data List a = Empty_List | Attach a (List a)
```

Unrolled and with some more meaningful constructor names:

```haskell
data List a = Empty_list a | Attach a (List a)
```

How many different values can this list hold?

\[ L(a) = 1 + a \cdot L(a) \quad \text{(literally from the type operators above)} \]

\[ L(a) = \frac{1}{1-a} \quad \text{(completely unjustified transformation ... we'll try it anyway)} \]

Reading out the Taylor series delivers exactly how many values a list with \( n \) elements can hold:

A list is either an empty list (one value), or is has one element (hence a possible values), or it has two elements (hence \( a^2 \) possible values), or it has three elements (hence \( a^3 \) possible values), ...

Binary Tree

Verbal definition: A binary tree is either empty or it holds an element followed by two trees – or:

```haskell
data Binary_Tree a = Alg_Tree (One :+: (a :+: (Binary_Tree a) :+: (Binary_Tree a)))
```

Unrolled and with some more meaningful constructor names:

```haskell
data Binary_Tree a = Empty_Tree a | Node a (Binary_Tree a) (Binary_Tree a)
```

How many different values can this binary tree hold?

\[ T(a) = 1 + a \cdot (T(a))^2 \quad \text{(literally from the type operators above)} \]

\[ T(a) = \frac{1}{1-4a} \quad \text{(another completely unjustified transformation ... we'll try it anyway)} \]
Binary Tree

Verbal definition: A binary tree is either empty or it holds an element followed by two trees – or:

\[
\text{data } \text{Binary\_Tree } a = \text{Alg\_Tree} \left( \text{One :+ (a :* (Binary\_Tree a) :* (Binary\_Tree a))} \right)
\]

Unrolled and with some more meaningful constructor names:

\[
\text{data } \text{Binary\_Tree } a = \text{Empty\_Tree} | \text{Node a (Binary\_Tree a) (Binary\_Tree a)}
\]

How many different values can this binary tree hold?

\[
T(a) = 1 + a \cdot (T(a))^2 \quad (\text{literally from the type operators above})
\]

\[
T(a) = \frac{1}{2} a + \frac{1}{4} a^2 + \frac{1}{8} a^3 + \frac{1}{16} a^4 + \frac{1}{32} a^5 + \ldots
\]

(Taylor series of above function)

Reading out the Taylor series delivers how many values a binary tree with \( n \) nodes can hold:

A binary tree with five nodes can hold \( 42^5 \) values, which also means it can appear in 42 different configurations!

© 2015 Uwe R. Zimmer, The Australian National University page 65 of 424 (chapter 1: “Algebras in Computer Science” up to page 71)

Binary Tree

Verbal definition: A binary tree is either empty or it holds an element followed by two trees – or:

\[
\text{data } \text{Binary\_Tree } a = \text{Alg\_Tree} \left( \text{One :+ (a :* (Binary\_Tree a) :* (Binary\_Tree a))} \right)
\]

Unrolled and with some more meaningful constructor names:

\[
\text{data } \text{Binary\_Tree } a = \text{Empty\_Tree} | \text{Node a (Binary\_Tree a) (Binary\_Tree a)}
\]

How many different values can this binary tree hold?

\[
T(a) = 1 + a \cdot (T(a))^2 \quad (\text{literally from the type operators above})
\]

\[
T(a) = \frac{1}{2} a + \frac{1}{4} a^2 + \frac{1}{8} a^3 + \frac{1}{16} a^4 + \frac{1}{32} a^5 + \ldots
\]

(Taylor series of above function)

Reading out the Taylor series delivers how many values a binary tree with \( n \) nodes can hold:

A binary tree with five nodes can hold \( 42^5 \) values, which also means it can appear in 42 different configurations!
Algebras in Computer Science

Algebraic types foundations: Recursive types

Binary Tree

**Verbal definition:** A binary tree is either empty or it holds an element followed by two trees – or:

\[ \text{data } \text{Binary}_\text{Tree} = \text{Alg}_\text{Tree} (\text{One} : + (\text{Binary}_\text{Tree} a) : * (\text{Binary}_\text{Tree} a)) \]

Unrolled and with some more meaningful constructor names:

\[ \text{data } \text{Binary}_\text{Tree} = \text{Empty}_\text{Tree} | \text{Node} a (\text{Binary}_\text{Tree} a) (\text{Binary}_\text{Tree} a) \]

How many different values can this binary tree hold?

\[ T(a) = 1 + a \cdot (T(a))^2 \]  
(literally from the type operators above)

\[ T(a) = \frac{1}{2} \left(1 + \sqrt{1 - 4a}\right) \]  
(another complete)

\[ T(a) = 1 + a + 2a^2 + 3a^3 + 4a^4 + 4a^5 + \ldots \]

Reading out the Taylor series delivers how to draw a binary tree with four nodes:

14 ways to draw a binary tree with four nodes:

- \( \text{null} \) configurations

A binary tree with five nodes can hold \( 42a^5 \) values, which also means it can appear in 42 different configurations:

42 ways to draw a binary tree with five nodes:

- \( \text{null} \) configurations

---

Algebras in Computer Science

Summary

Algebras in Computer Science

- **Algebra** (general notion)

- **Algebras for non-numeric entities in Computer Science (Algebraic Types)**
  - Sum types
  - Product types
  - Exponential types
  - Recursive types

- **Practical types derived:**
  - Boolean, Enumerations, Ranges
  - Variants (Either), Maybe
  - Tuples, Vectors, Records
  - Lists, Binary trees