

Functions and Symbols

(etc)

Pure computational functions

- Maps a **domain** to a **codomain**
 - We use “arrow syntax” to show domain and codomain
 - Tells us *what kinds of things* the input and output are
- Either terminates (i.e. provides an answer), or does not terminate (i.e. loops infinitely, or throws an exception)
 - If it *always* terminates properly, it is a **total function**. Otherwise, it is a **partial function**.
- Maps each element of the domain to an element of the codomain—and does nothing else (i.e. is “pure”)!
 - We use “barred arrow syntax” to show exactly how this is done
 - Tells us *what computation* generates the output

$$\mathbb{N} \rightarrow \mathbb{R}$$

$$x \mapsto x^2 + \frac{x}{3} + 1$$



Calculating VAT

$$x \mapsto 1.15x$$

- Calculates VAT on a sale (VAT is 15% of the price)
- How? Barred-arrow syntax tells us:
 - Take some *symbolic price* x
 - Perform the computation: multiply that *symbolic* price by **1.15**
- On the right side, we see the **body** of the function:
 - **$1.15x$**
- On the left side, we see the **input** of the function:
 - **x**
- **No value** can be output *until the input is bound!*

Calculating VAT

$$x \mapsto 1.15x$$

- Calculates VAT on a sale (VAT is 15% of the price)
- How? Barred-arrow syntax tells us:
 - Take some *symbolic price* x
 - Perform the computation: multiply that *symbolic* price by **1.15**
- On the right side, we see the **body** of the function:
 - **$1.15x$**
- On the left side, we see the **input** of the function:
 - **x**
- **No value** can be output *until the input is bound!*

NOTE that the function is fully specified: it does not need a name, or multiple parameters.

In our programming language (F#)

$$x \mapsto 1.15x$$

Syntax of mathematics

`fun x -> 1.15 * x`

Syntax of F#

- Prefix with “fun” to say it’s a function.
- Use “->” instead of “ \mapsto ” between input and body
- Use “*” to multiply

In our programming language (F#)

Let $x = 16.51$ in $x \mapsto 1.15x$

Syntax of mathematics (binding)

- “Let” sets the value of a symbol

(fun x -> 1.15 * x) 16.51

Syntax of F# (binding)

- Put the whole function in brackets
- Put the value after the brackets



Demo time

Specifying the VAT rate

What if the 15% VAT rate changes? A function that accepts a tax rate t would be more useful than a function which always assumes a 15% tax rate.

How do we write that?

- In an imperative programming language, it might look like:

- `def calculateVat(t, x):`
 `return t * x`

👉 Python

- `double calculateVat(double t, double x) {`
 `return t * x;`
 `}`

👉 Java

- What would it look like as a *function* in *mathematical* notation? Is it $(t, x) \mapsto t \cdot x$?
- What would it look like as a *function* in our *programming language*?

Binding

Binding involves three ideas:

- “Immutability”
- “Referential transparency”
- Scope

...which you have seen before, many times, but you just didn't have names for them 😊!

Immutability

Imagine a particular coin value that is rolling down this coin-sorting machine.

- It doesn't matter *which* coin you are imagining, as long as it has some defined and known value.

Three *positions* that the coin will be in are marked on the picture as “1”, “2”, and “3”.

Question: **does the coin change into a different coin at any point?** (e.g. a R2 becoming a R5)



Immutability

This is a (simple) algebraic problem, where x has been bound to the value 3.

Question: **do any of the values become a different value at any point in the problem?** (e.g. the value 4 becoming the value 8)

Solve $\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)}$, given $x = 3$.

$$\begin{aligned}\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)} &= \frac{x^2 + 4x + 5}{2x^3 + 3x^2 + 4x + 6} \\ &= \frac{3^2 + 4 \cdot 3 + 5}{2 \cdot 3^3 + 3 \cdot 3^2 + 4 \cdot 3 + 6} \\ &= \frac{9 + 12 + 5}{54 + 27 + 12 + 6} \\ &= \frac{26}{99}\end{aligned}$$

Immutability

The “coin” of the machine and the values in the equation are **immutable**: they never change what they are.

The same thing is true in functional programming.

- You can *create new values* in a computation
 - If you do “5+1”, you get the new value 6. But that doesn’t change the values of 5 or 1. If a new value is *based on* other values, that doesn’t mean that it *changes* the other values!
- But you can **never change** a value once it has been created.

All values in a functional language **never change** after they have been created, just like in Mathematics! This is called **immutability**.



$$\begin{aligned} \text{Solve } \frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)}, \text{ given } x = 3. \\ \frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)} &= \frac{x^2 + 4x + 5}{2x^3 + 3x^2 + 4x + 6} \\ &= \frac{3^2 + 4 \cdot 3 + 5}{2 \cdot 3^3 + 3 \cdot 3^2 + 4 \cdot 3 + 6} \\ &= \frac{9 + 12 + 5}{54 + 27 + 12 + 6} \\ &= \frac{26}{99} \end{aligned}$$

Referential transparency

Solve $\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)}$, given $x = 3$.

$$\begin{aligned}\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)} &= \frac{x^2 + 4x + 5}{2x^3 + 3x^2 + 4x + 6} \\ &= \frac{3^2 + 4 \cdot 3 + 5}{2 \cdot 3^3 + 3 \cdot 3^2 + 4 \cdot 3 + 6} \\ &= \frac{9 + 12 + 5}{54 + 27 + 12 + 6} \\ &= \frac{26}{99}\end{aligned}$$

Consider the computation on the left, where x has been bound to the value 3.

Question: **if I replace x with 3 at any point, will I get the same answer?**

Referential transparency

In functional programming, just like in Mathematics, once a symbol is bound to a value, it can always be replaced by that value.

If we say x or we say 3 , we mean exactly the same thing!

We can always replace a symbol with its value, or a value with its symbol, and nothing will change in the computation. This is called **referential transparency**.

Solve $\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)}$, given $x = 3$.

$$\begin{aligned}\frac{x^2 + 4x + 5}{(x^2 + 2)(2x + 3)} &= \frac{x^2 + 4x + 5}{2x^3 + 3x^2 + 4x + 6} \\ &= \frac{3^2 + 4 \cdot 3 + 5}{2 \cdot 3^3 + 3 \cdot 3^2 + 4 \cdot 3 + 6} \\ &= \frac{9 + 12 + 5}{54 + 27 + 12 + 6} \\ &= \frac{26}{99}\end{aligned}$$

Scope

Scope is about *where a binding exists* and *how long a binding exists*.

In “A” and “B”, the name of the symbol is the same.

Question: is the x in “A” bound to the same value as the x in “B”?

A Let $x = 5$ in $x \mapsto x + 5$.

\vdots *and later...*

B Let $x = 8$ in $x \mapsto \frac{3x}{2}$.

Scope

Scope is about *where a binding exists* and *how long a binding exists*.

“Where does a binding exist?”

- In any place where a symbol is bound to a value.
- In “A”, it exists just after we say $\text{Let } x = 5 \text{ in } x \mapsto$
- In “B”, it exists just after we say $\text{Let } x = 8 \text{ in } x \mapsto$

A $\text{Let } x = 5 \text{ in } x \mapsto x + 5.$

\vdots *and later...*

B $\text{Let } x = 8 \text{ in } x \mapsto \frac{3x}{2}.$

Scope

Scope is about *where a binding exists* and *how long a binding exists*.

“How long does a binding exist?”

- Until the end of the body.
- In “A”, it exists until the end of the **body** $x + 5$.
- In “B”, it exists until the end of the **body** $\frac{3x}{2}$.
- As soon as a function has ended, there is no **x** symbol.

A Let $x = 5$ in $x \mapsto x + 5$.

\vdots *and later...*

B Let $x = 8$ in $x \mapsto \frac{3x}{2}$.

Scope

Scope is about *where a binding exists* and *how long a binding exists*.

In our programming language, we have very similar rules:

- A binding exists in any place where a symbol is bound to a value.
- A binding exists until the end of the body (or other scope).

A Let $x = 5$ in $x \mapsto x + 5$.

⋮ and later...

B Let $x = 8$ in $x \mapsto \frac{3x}{2}$.

Scope

Scope is about *where a binding exists* and *how long a binding exists*.

In our programming language, we have very similar rules:

- A binding exists in any place where a symbol is bound to a value.
 - See **circles+arrow** on right: the 16.51 *value* is **bound** to the **x** symbol.
- A binding exists until the end of the body (or other scope).
 - See **underline** on right

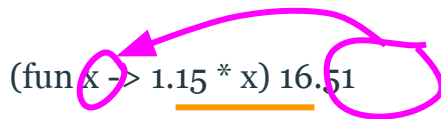
In our programming language, there are also other places where binding takes place. We will explain them, and their scope, as we encounter them.

A Let $x = 5$ in $x \mapsto x + 5$.

⋮ and later...

B Let $x = 8$ in $x \mapsto \frac{3x}{2}$.

(fun **x** -> 1.15 * x) **16.51**

A diagram illustrating variable binding in a lambda expression. The expression is (fun x -> 1.15 * x) 16.51. A pink arrow points from the argument 16.51 to the parameter x. Both the parameter x and the argument 16.51 are circled in pink. The body of the function, 1.15 * x, is underlined in orange.

“Binding” definition

A **referentially transparent** link between
a **symbol** and an **immutable value**
within a **scope**

Terminology: “apply” / “function application”

We **apply** a function to an input, and this generates some result.

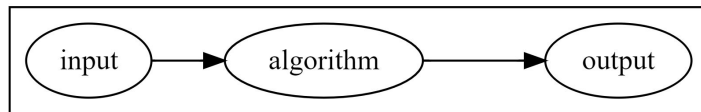
Function application is the process of applying a function to an input.

Synonym: “call”. We **call** a function by supplying it with an input.

What can a function do?

What sort of things do functions or methods in Java, Python, C, etc do?

Do they do only what the Theory of Computation says they do?



Do *all* functions only do this?

If not, what *else* do they do?

Our functions are **deterministic**

A function is **deterministic** if it will always return the same output value when given the same input value, *without* relying on anything other than its input to generate its output.

Examples:

$$x \mapsto x + 1$$

$$x \mapsto 5$$

$$(x, y, z) \mapsto (y + z)/x$$

A **non-deterministic** function might change its output depending on the time, or a randomly-picked number, or how much network traffic it has received, or the result received from a database, or...

Our functions have no side-effects

A **side-effect** is any **externally-visible change** that a function does, apart from generating a result.

Examples:

- Printing some text to the console
- Writing a file
- Sending data over the network
- Updating the data in a database

In more advanced functional programming, we will handle side-effects using a programming trick called “monads”. In this course, we will *isolate* side-effects strictly.

Deterministic + no side-effects = pure

A **pure** function is one which is **deterministic** and has **no side-effects**. All functions in functional programming are pure *by default*; in fact, some well-known functional languages *only* allow pure functions.

But why do we want this??

- Can apply *equational reasoning* (i.e. treat your program like a mathematical equation!)
- Hugely easier for a compiler to optimise
- Can *prove* correctness of programs
 - *Testing* correctness is much easier too
- Free, cheap, and easy parallel and distributed processing!
- Time-travel debugging
- No runtime errors, no exceptions, and no “state”-related bugs—ever!
- Use the computer’s spare time to precompute results

Making beautiful music

Think about a single sound, like a clap or a musical note.



It can be useful and nice to hear, just on its own. But when you can reuse it cheaply and easily, then you can take reusable sounds and turn them into actual music.



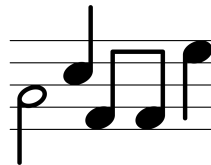
[!\[\]\(339a16584d5da0f0a3ca4e9ec17bf6a1_img.jpg\) Let's Make Beautiful Music Together !\[\]\(e06a1d39938b2f5d7a2c3618fea4f77f_img.jpg\)](#)

Making beautiful ~~math~~ programs

Think about a single sound, like a clap or a musical note.



It can be useful and nice to hear, just on its own. But when you can reuse it cheaply and easily, then you can take *reusable sounds* and turn them into actual music.



A mathematical function is like a single sound. Useful on its own, but when you can reuse it cheaply and easily, then you can take *reusable functions* and turn them into a mathematical argument.

When your mathematical functions are also *computational* functions—i.e. when they have an *algorithm* attached to them—you can turn your reusable functions into a program!

Let $calculateVat(x) = 1.15x$

Let $addBag(y) = y + 0.28$

Let $price(v) = addBag(calculateVat(v))$

Binding a function 🤔

Let *calculate* $Vat(x) = 1.15x$

In mathematics, we can bind a function to a symbol, if we want to. We can bind a function to a symbol in functional programming, too.

But remember that in general, a function does not need to have a name*.

(A “normal” function, i.e. a function without a name, is called a **lambda function**.)

* there is one exception**, and we’ll talk about it soon.

** it’s not *really* an exception, either. But talk to me about that in Honours.

In our programming language (F#)

Let *calculateVat*(x) = $1.15x$

Syntax of mathematics

let calculateVat x = $1.15 * x$

Syntax of F#

- No brackets are needed.
- “let” is in lowercase.

In our programming language (F#)

calculate Vat(16.51)

Syntax of mathematics

calculateVat 16.51

Syntax of F#

- No brackets are needed.

Recursion

Recursion is the basic idea of doing things repeatedly, **until** we reach a point where we don't need to do them any more.

(you may already be very familiar with this idea, because you've already done loops in imperative programming!!)

To do something repeatedly, we need to have:

- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
 - If we **didn't** have this, then our functions would not be total!
- Some result of all the repetition.
 - Otherwise, why are we bothering to do all of this repeating??

Recursion

Recursion is the basic idea of doing things repeatedly, **until** we reach a point where we don't need to do them any more.

To do something repeatedly, we need to have:

- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

$$\begin{aligned}x &\mapsto 1.15x \\ y &\mapsto y + 0.28\end{aligned}$$

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged? We will need to add *multiple* 28-cent charges.

(just assume that we can't use multiplication here, for whatever reason, so we want to loop instead 😊)

Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

```
double calculatePrice(int numBags, double x)
{
    double result = 0;
    while (numBags > 0) {
        result += 0.28;
        numBags--;
    }
    result += 1.15 * x;
    return result;
}
```

Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

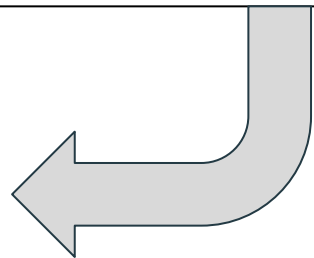
- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

Note the *reuse*, multiple times, of ***addBag*** and ***price***! We can only do this because we can bind a symbol to a function.

Let $calculateVat(x) = 1.15x$

Let $addBag(y) = y + 0.28$

Let $price(bags, x) = \begin{cases} calculateVat(x) & bags = 0 \\ addBag(price(bags - 1, x)) & \text{otherwise.} \end{cases}$



Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

Let $calculateVat(x) = 1.15x$

Let $addBag(y) = y + 0.28$

Let $price(bags, x) = \begin{cases} calculateVat(x) & bags = 0 \\ addBag(price(bags - 1, x)) & \text{otherwise.} \end{cases}$

Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

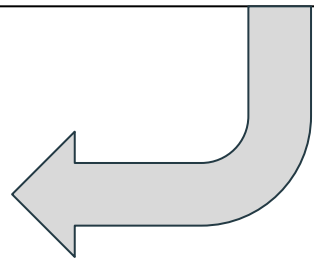
- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

What is the scope of the bindings in this function? How can you tell?

Let $calculateVat(x) = 1.15x$

Let $addBag(y) = y + 0.28$

Let $price(bags, x) = \begin{cases} calculateVat(x) & bags = 0 \\ addBag(price(bags - 1, x)) & \text{otherwise.} \end{cases}$



In our programming language (F#)

$$\text{Let } f(x) = \begin{cases} 0 & x \leq 1 \\ 1 + f(x - 1) & \text{otherwise.} \end{cases}$$

Syntax of mathematics

```
let rec f x =  
    if x <= 1 then  
        0  
    else  
        1 + f (x-1)  
Syntax of F#
```

- “rec” is necessary for all functions that call themselves.
- A conditional expression is needed to determine when the recursion stops.
- The if-expression...
 - **must** have a “then” branch and an “else” branch
 - **must** return the same type from both branches
 - Is written as: “if ... then ... else ...”

Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
- Some result of all the repetition.

```
let addBag y = y + 0.28
let calculateVat x = x * 1.15
let rec price (bags, x) =
    if bags = 0 then
        calculateVat(x)
    else
        addBag(price(bags-1, x))
```

- The programming language uses a single “=” to check for equality, unlike Java/Python/C/etc, which all use “==”.
- We use *indentation* in F# to define *scope*.



Recursion

Practical problem: Consider shopping that needs *many plastic bags*. How do we calculate the price that is charged?

To do something repeatedly, we need to have:

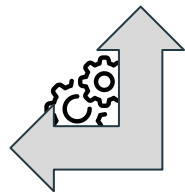
- Some point that we (re-)start repeating from.
- Some condition that tells us when we should stop repeating.
 - If the condition leads to a *non-recursive* branch, that branch is called the **base case** (see underlined section).
 - Otherwise, the branch is called the **recursive case** (see **bold** section)
- Some result of all the repetition.

Let $calculateVat(x) = 1.15x$

Let $addBag(y) = y + 0.28$

Let $price(bags, x) = \begin{cases} calculateVat(x) \\ addBag(price(bags - 1, x)) \end{cases}$

```
let addBag y = y + 0.28
let calculateVat x = x * 1.15
let rec price (bags, x) =
  if bags = 0 then
    calculateVat(x)
  else
    addBag(price(bags-1, x))
```



$bags = 0$
otherwise.

My Advice About Recursion

Technically speaking, some people say recursion must “solve smaller subproblems by calling the current function”. That’s *technically* true (given certain meanings of “smaller”, “subproblem”, and “current”). But it’s also a pointlessly complex statement that you should ignore—for now.

If your only way to “loop” is by using recursion, you’ll naturally figure out how to do it, just like you naturally figured out how to do a while-loop. This will give you an intuitive understanding of recursion. That intuitive understanding, right now, is much more valuable than a technical definition.

Then, later on, you’ll be able to analyze *what* you’ve done *after* you’ve done it, and it will be much easier to understand.

Build Your Own Recursive Function

$$\text{Let } f(x) = \begin{cases} 0 & x \leq 1 \\ 1 + f(x-1) & \text{otherwise.} \end{cases}$$

```
let rec f x =  
  if x <= 1 then  
    0  
  else  
    1 + f (x-1)
```

Can you create these recursive functions, using *mathematical syntax* and *programming language syntax*?

- A function to repeatedly divide a number by 3, until it is no longer divisible by 3? The non-divisible part is returned.
- A function to repeatedly divide a number by n , until it is no longer divisible by n ? The non-divisible part is returned.
- A function to repeat a string n times, returning that string?
 - In the programming language, “wh” + “oops” = “whoops”
 - Mathematics doesn’t have a consistent way of representing strings. Just use the quotation-mark programming-language syntax.

Interlude: “let” binding

We can also use “let” to bind any value to a symbol. Again, this is just like it is in mathematics.

Often, there’s *no need* to do this. Binding non-function values is boring and unnecessary!

But if you’re going to *reuse* the value, then it’s convenient to give it a name.

In our programming language (F#)

Let $x = 9$

Syntax of mathematics

let x = 9

Syntax of F#

- “let” is in lowercase

Summary

- Values, functions, and symbols are what do the work of computation
-