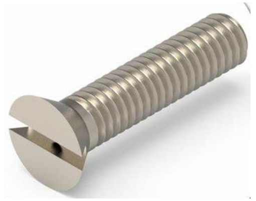
Function Techniques

“Just Functions?” What can you do with “just functions”??!

Toolboxes

Imagine that you are a handyman.

If you have to screw in a screw like →
what will you use?

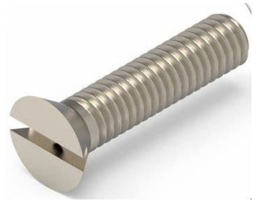


(a) A hammer
(b) A star-shaped screwdriver
(c) Pliers
(d) A flat-tipped screwdriver
(e) Bolt-cutters

Toolboxes

Imagine that you are a handyman.

If you have to screw in a screw like → what will you use?



(a) A hammer
(b) A star-shaped screwdriver
(c) Pliers
(d) A flat-tipped screwdriver
(e) Bolt-cutters

Why? Because different **tools** can be used in different **ways**. That is totally normal—in fact, that is a big part of the reason why we make different tools in the first place!

Types

Let’s go back to being a programmer.

If you have to count the number of characters in a string, what type will you use?

(a) A string
(b) A floating-point number
(c) Doubly-linked list
(d) An integer
(e) A null reference

Types

Let’s go back to being a programmer.

If you have to count the number of characters in a string, what type will you use?

(a) A string
(b) A floating-point number
(c) Doubly-linked list
(d) An integer
(e) A null reference

Why? Because different **types** can be used in different **ways**. That is totally normal—in fact, that is a big part of the reason why we use different types in the first place!

Different types can do different things

- With an integer, you can use + and - and * and /
- With a floating point number, you can do all that and use **
- With a string, you can use +
- With a function, … 🤯?

A function is **anything** which accepts **one input value** and, when given that input, **generates one output value.**

The type of a function is defined by its input and output types:

$$\textbf{type}_{\textbf{input}} \longrightarrow \textbf{type}_{\textbf{output}}$$

Functions: what are they good for?

Helping out

Imagine that you are part of a team that is working on a project.

A member of the team doesn’t know how to do something, but you do.

What do you do?

Helping out

Imagine that a lecturer wants your friend to write an essay for Politics.

The friend knows how to write things, but has never written a Politics essay before.

What do you do?

Helping out

The ideas of **sending help** and **getting help** are as old as humanity. We all need help sometimes, and we all help other people sometimes. This idea is in all religions, all societies, and all ethical systems.

Let’s assume that there are two people, “Adam” and “Kim”. If Kim doesn’t know how to do a small part of a bigger job, but Adam does, then Adam should go and do that small part for Kim.

This doesn’t mean that Kim is incompetent, or isn’t able to do the job. It just means that Adam can do a small part of the job better. Maybe Adam has been researching in that area for many years, or maybe it is a matter of taste and Adam is the one unique person in the world who happens to know exactly how that small part of the job should be done.

We can say that for whatever reason, Adam is a **specialist** in doing that part of the job.

A specific problem

Let’s say that Adam asks Kim: “please add 1 to all these numbers”, and gives Kim a list of numbers.

Kim can do that.

If Adam asks Kim to “multiply these numbers by 5”, Kim can do that too.

What if Adam asks Kim: “please **do something** with each of these numbers”?

Can we write a function that would satisfy Adam?

“Specialist” functions

In programming—and in mathematics—we often want to use “specialist” functions. For example, when determining *which* value we want in a sequence, we can pass a “specialist” function that identifies the value.

Very often, a “specialist” function will say <u>**how**</u> to do something, but not <u>**what**</u> to do.

In our example, a ***find*** function knows that it should find a value—this is <u>**what**</u> it should do.

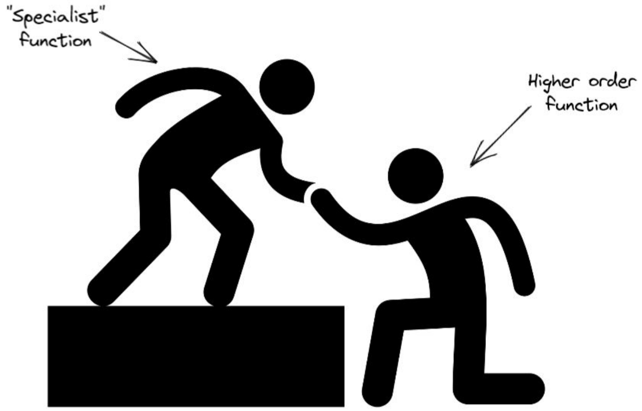
A “specialist function” tells it <u>**how**</u> to find the value.

Terminology: *higher order* function

If a function **accepts** or returns a "specialist" function, then that function is called a **higher order function**.

find, ***filter***, ***map***, etc are all **higher order functions**.

(in case you are wondering, there is no agreed-upon name for a "specialist" function! Some sources call them *callback functions*, but that isn't a well-established name—yet.)



Helping out, Part 2

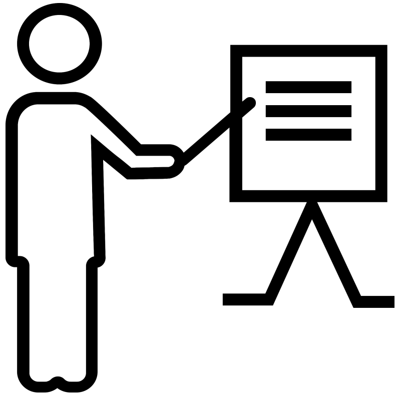
When we need to “give help” to a **higher order** function, we can send a “specialist” function to it, and that “specialist” function can help the higher order function to do its work.

But the other way of helping is by *educating* or *training* the other person, so that they can do the job themselves.

Let’s return to our example of people helping each other.

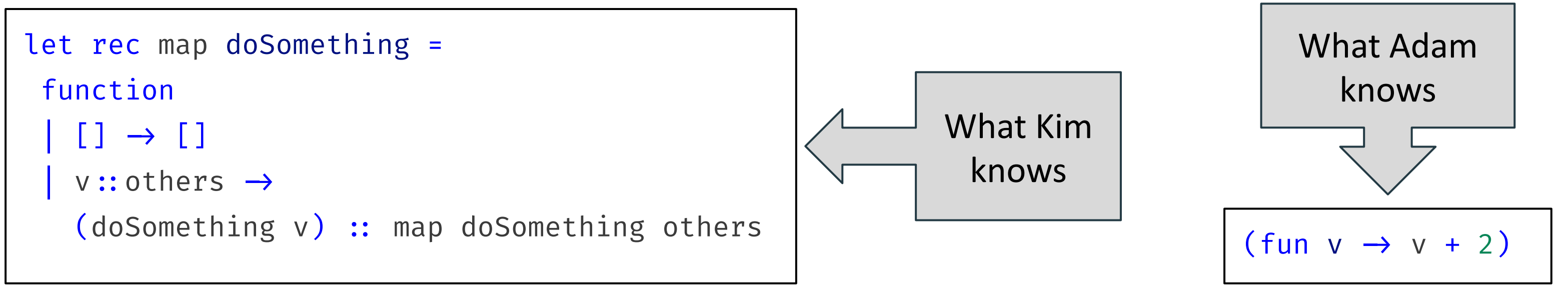
Kim knows how to go through lists. Adam knows what to do with list items.

If Adam *educates* Kim about *how to* “do something”, then
Kim will be able to always remember and do it—for any list!



Helping out, Part 2

Initially, Kim knows how to go through a list, but not how to do anything in particular with the items in that list.



```
let rec map doSomething =
 function
 | [] → []
 | v::others →
   (doSomething v) :: map doSomething others
```

```
(fun v → v + 2)
```

After being educated, we want Kim know how to do a specific thing (which Adam teaches Kim), with any list.

Interlude: back to being a handyman!

To fix a leak, our handyman might need **many things**:

- Knowledge of *how to use a wrench to fix the leak*.
- A *wrench*.
- *Plumbers' tape*.

…and at the end, the result is a fixed (non-leaking!) pipe. As long as we provide a handyman with all these three things, we won't have leaks.

We can say that these things are *dependencies* for the result. If one is missing, our house will be flooded.

(this is a terrible simplification, please don't show it to any actual plumbers! 🙏🏾)

Helping out, Part 2

```
let rec map doSomething =
 function
 | [] → []
 | v::others →
   (doSomething v) :: map doSomething others
```

Initially, Kim knows how to go through a list, but not how to do anything in particular with the items in that list.

After being educated, we want Kim know how to do a specific thing (which Adam teaches Kim), with any list.

Let’s look at the **type** of the **higher-order function**.

`('a → 'b) → (List<'a> → List<'b>)`

But we usually write this with fewer brackets, as:

`('a → 'b) → List<'a> → List<'b>`

Helping out, Part 2

```
('a → 'b) → List<'a> → List<'b>
```

Let’s understand this in terms of “what is needed to construct something else”—in other words, in terms of dependencies.

That last bit (purple) is what is generated after the first two things are provided. Those first two things are dependencies of the last thing.

Helping out, Part 2

```
('a → 'b) → List<'a> → List<'b>
```

Usually, we provide this dependency first.

Helping out, Part 2

```
('a → 'b) → List<'a> → List<'b>
```

We provide this dependency second.

Does the **order of dependencies** matter?

🤯 No! 😲

Helping out, Part 2

```
('a → 'b) → List<'a> → List<'b>
```

If we provide the *first* dependency only, we get:

```
List<'a> → List<'b>
```

If we provide the *second* dependency only, we get:

```
('a → 'b) → List<'b>
```

What does this mean in terms of “Adam” and “Kim”?

Helping out, Part 2

```
('a → 'b) → List<'a> → List<'b>
```

If we provide the *first* dependency only, we get:

```
List<'a> → List<'b>
```

Example code: `map (fun x → x + 2)` or `fun a → map (fun x → x + 2) a`

If we provide the *second* dependency only, we get:

```
('a → 'b) → List<'b>
```

Example code: `fun f → map f [5;6;7;2]`

Bound and free variables

When looking at things like this, a concept that might be useful is that of **bound** and **free** variables.

A **bound** symbol has been bound to a value.

A **free** symbol is used in a function, but doesn’t have a value bound to it—yet!

We turn our **free** symbols into a **bound** symbols through **function application**, and if the result is a function, then we can say (informally) that it is a more “educated” function 😄👍🏾!

Terminology: *partial application*

When we specify a *part* of a function’s inputs, and leave the rest of them to be specified later, that is like a function technique called *partial application**.

Intuitively, this is about *building new functionality* by bringing together functionality that we already have. It is about *teaching* a “general” function how to do something more specific.

In our example, we would say that ***map*** has been *partially applied*.

* actually, *partial application* involves reducing the “arity” of a function too. But we’ll cover that later.

Finding closure

Terminology: arity

When you *partially apply* a function, the result is an “educated” function with fewer “dependencies”.

We say that a *partially applied* function has a smaller **arity**: a smaller number of “input slots”. The minimum arity for a function is **1**.

- Don’t confuse this with the “arity” of a tuple, which is the number of elements in the tuple

Terminology: closure

A **bound** symbol has been bound to a value.

A **free** symbol is used in a function, but doesn’t have a value bound to it—yet!

A **closure** is a function value that contains one or more bound symbols.

Why do we use closures?

Closures let us make *new functions* that have some kind of education. But they can also serve another purpose.

Imagine that you work in a shop. Somebody calls and says “I want to order a double-bed, and a sofa, and a dressing-table”, and they give you all of the necessary information (name, delivery address, cellphone number, payment details, etc etc)—but at the end of the call, they say:

“Just <u>wait a bit</u>, I want to check something. I might <u>call back later</u> to confirm.”

…what do you do?

Delaying functions

Closures let us make *new functions* that have some kind of education. But they can also serve another purpose.

Let's go back to being software developers.

- **Scenario 1**. Your user has given you all the details about something, and when your program asks "Are you sure you want to make these changes?", the user says "Um—just wait a bit. I'm not sure."
- **Scenario 2**. Some things are very computationally expensive to calculate. You want to delay calculating them until you're sure that they really *need* to be calculated, but you also want to remember all the values that you'll need to calculate them *if* it's needed.

In each case, you want remember what to do , but also delay doing it until later on. How do we do that?

Delaying functions

What we need is some way to have an educated function that will execute later on, ***without* requiring any additional information**. But we have a problem with this:

- A function is anything which accepts one input value and, **when given that input**, generates one output value.

How do we make a function that can be given an input, but *not* given any additional information?!

🤔

Why do we use closures?

How do we make a function that can be given an input, but *not* given any additional information?

We use a **unit value**, which *cannot* pass along any information.

Instead of returning a value directly, we return a closure which calculates the value: **fun () -> …**

The **()** is a pattern which will only match the unit value ().

Until we call the function , passing it a unit value (), the body will not be executed!