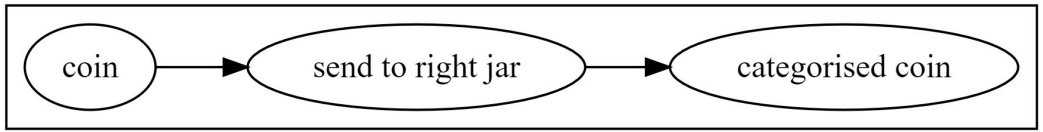
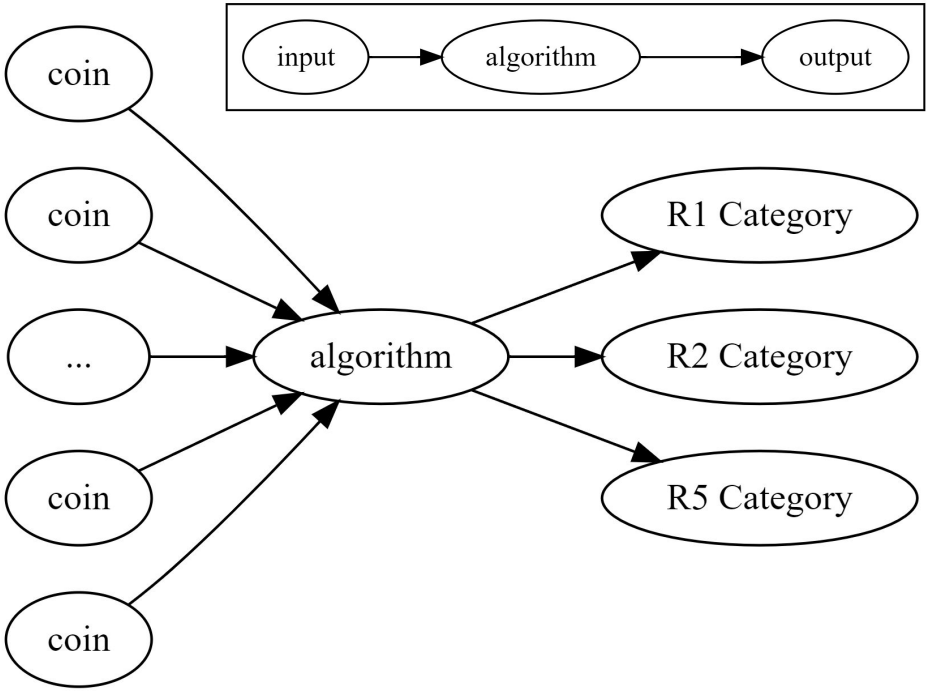
Programs By Design

Hair By Guess

A gravity-based coin-sorting machine



A gravity-based coin-sorting machine



What is “computation”?

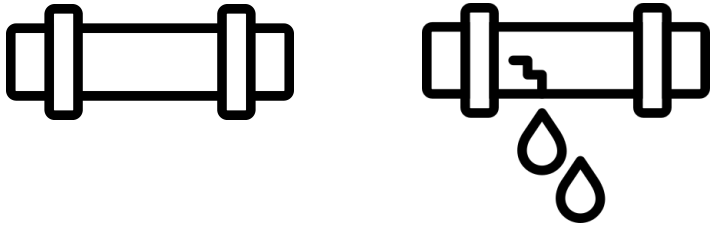
Every computable function solves a problem by:

1. Accepting an input (*from the domain*)
2. Following an algorithm
3. Providing an answer (*from the codomain*)

If an “error” occurs in a function, what can we say about the **function**?

Errors impossible

Core insight: an “error” is actually a **partial function** *in disguise*!

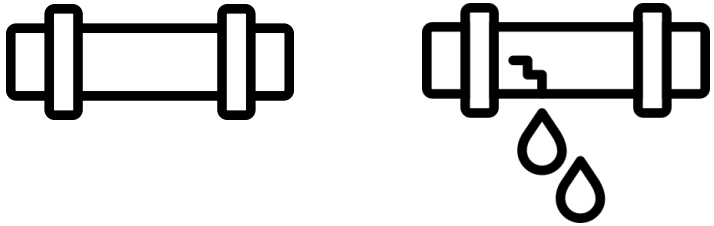


What if all our functions were **total**?

- All functions have a **domain** and **codomain**.

Errors impossible

Core insight: an “error” is actually a **partial function** *in disguise*!

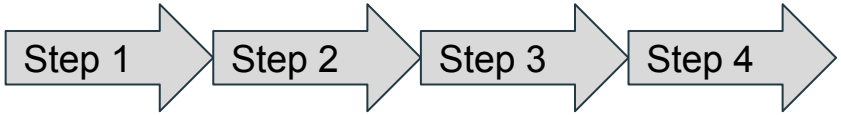


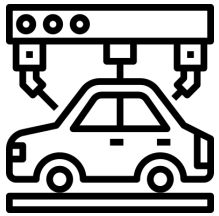
What if all our functions were **total**?

- All functions have a **domain** and **codomain**.
- Technique: expand the *codomain* so that **partial** functions become **total** functions!

On the factory floor

Imagine that you are working at a factory.





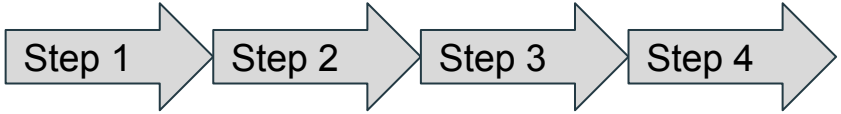
Every step in the assembly line expects *something* to be passed on from the previous step.

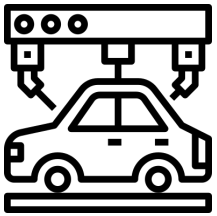
But something might go wrong during Step *n*. What do you do when this happens?

- Shout for a supervisor to help you?
  - …and blow up the factory if no supervisor is available?
- Replace the output of the step with a bomb?
- Pass along something that *looks like* a product, but isn't really a product, and hope that nobody mistakes it for a real product?

On the factory floor

Imagine that you are designing a program.





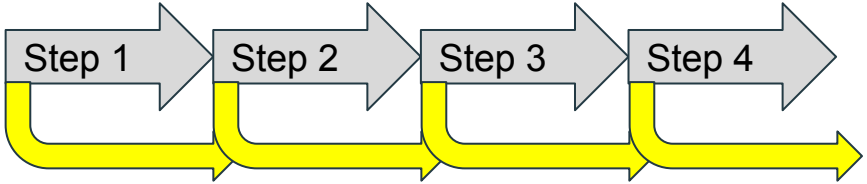
Every function in the program relies on some output to be passed in from the previous function.

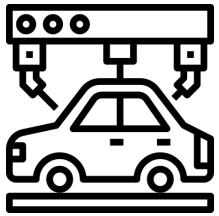
But something might go wrong during a function (e.g. division by 0). What do you do when this happens?

- Throw an exception?
  - …and crash the whole program if it's not handled?
- Return **null**, and pray that nobody tries to dereference it?
- Return an error code, and hope that nobody mistakes it for a real result?

On the factory floor

Imagine that you are working at a factory.



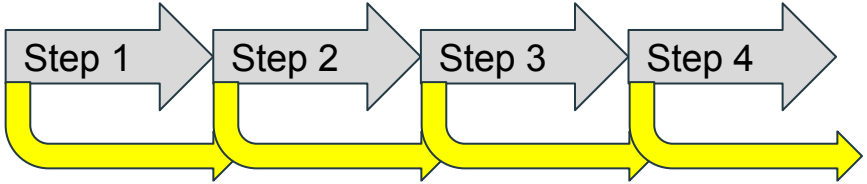


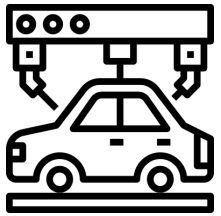
A better approach: if something goes wrong in Step *n*,

- Pass along a note that says “Sorry, something went wrong, and I couldn’t make a product.”
- Let the next step figure out how to deal with it.
  - If the next step also uses the same strategy, then it can also just pass along a note.
- Eventually, the note will either be handled by someone, or it will pass out of the system.

On the factory floor

Imagine that you are working at a factory.





Advantages:

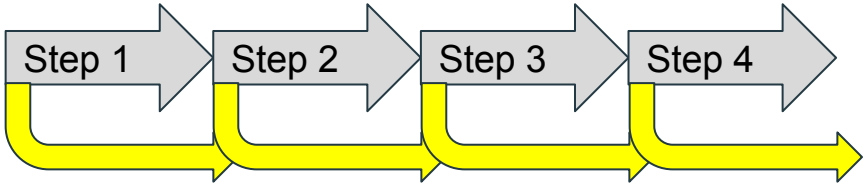
- Doesn't shut down the factory!
- You're passing a *note*, not an actual product, so nobody can mistake it for the actual product.

Disadvantages:

- Every step must be prepared to received *either* a note, or the actual output

On the factory floor

Imagine that you are designing a program.




```
type NoteOrItem<'data> =
| Sorry
| Item of 'data
```

Advantages:

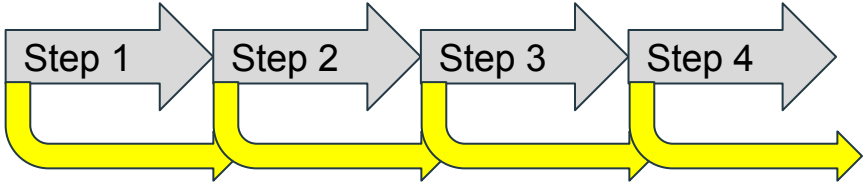
- Can’t crash the program!
- Impossible to mistake one case for the other case.

Disadvantages:

- Every function must be prepared to receive a **NoteOrItem**.

On the factory floor

Imagine that you are designing a program.



Advantages:

- Can’t crash the program!
- Impossible to mistake one case for the other case.

Disadvantages:

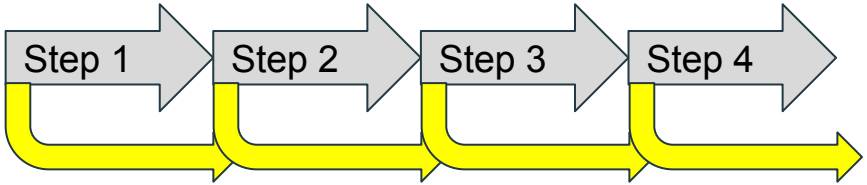
- Every function must be prepared to receive a **NoteOrItem**.

```
type NoteOrItem<'data> =
| Sorry
| Item of 'data
```

Codomain has been expanded!

Or, even better...!

Imagine that you are designing a program.




```
type ExcuseOrValue<'data, 'reason> =
| Excuse of 'reason
| Value of 'data
```

Codomain has been expanded!

Advantages:

- Can't crash the program!
- Impossible to mistake one case for the other case.

Disadvantages:

- Every function must be prepared to receive a **ExcuseOrValue**.

Option<'a> and Result<'a,'b>

In F#, there are built-in types defined as:

```
type Option<'a>
  = None
  | Some of 'a
```

```
type Result<'a,'b>
  = Ok of 'a
  | Error of 'b
```

Other than the names of the cases, these work *exactly the same* *as* our **NoteOrItem<'a>** and **ExcuseOrValue<'a,'b>** types.