# Functions and pattern matching

Let us recap the types we have constructed so far: enumerations, sums, products:

```
data Void : Set where

data Unit : Set where
  empty : Unit

data Boolean : Set where
  true false : Boolean

data Prod (A B : Set) : Set where
  pair : A -> B -> Prod A B

data Sum (A B : Set) : Set where
  left  : A -> Sum A B
  right : B -> Sum A B
```

A data-type, as you know, is not useful unless we can construct elements of the type, and also access data inside an element of a type. The *constructors* are part of the definition of the type, as we have seen.

To access the data inside a type in functional languages we have

*destructors*. These destructors are built into function definitions via a mechanism known as *pattern-matching*.

# Pattern matchine enumerations

Suppose that you want to write the `nand` operator for `Booleans`. The type of the "negated and" operator is `nand : Boolean -> Boolean -> Boolean`. The `Boolean` data type has two elements (constructors) `true` and `false`. The Agda syntax (and similarly in other functional languages such as Haskell and OCaml is) for defining the function is:

```
nand : Boolean -> Boolean -> Boolean
nand true  true  = false
nand true  false = true
nand false true  = true
nand false false = true
```

The *pattern-matching* mechanism will check that all cases are covered. This is an important distinctions between functional and other languages which provide case definition ( `case` or `switch` statements in various languages).

For example, if we forget a case Agda will complain and will actually issue an error (OCaml and Haskell are more forgiving and will issue a warning):

```
nand : Boolean -> Boolean -> Boolean
```

```
nand true  true  = false
nand true  false = true
nand false true  = true

Incomplete pattern matching for and. Missing cases:
  and false false
when checking the definition of and
```

A great deal of programming errors happen because cases are forgotten, leading to runtime errors. In the case of `nand` the missing case is easy to check, but complicated types are harder to check.

In order to avoid the explosion of cases variables can be used in the patterns (i.e. on the *left* of the `=`):

```
nand : Boolean -> Boolean -> Boolean
nand true  true  = false
nand true  false = true
nand false x     = true
```

When a function is computed then the appropriate case is identified by going top-to-bottom in the list of cases. For example `nand true false` does not match the first case but it matches the second so it returns `true`. But `nand false false` will only match `nand false x` case in which `x` is taken to be `true`.

Because the matching goes top-to-bottom we can further simplify,

without ambiguity, to

```
nand : Boolean -> Boolean -> Boolean
nand true true = false
nand x    y    = true
```

Indeed `nand true true` matches both cases, with `x = true` and `y = true` but because the first clause is attempted first there is no confusion that the result should be `false`.

Finally, it is considered "nice style" that if a variable is not actually used in the computation it is given the special `_` name:

```
nand : Boolean -> Boolean -> Boolean
nand true true = false
nand _    _    = true
```

Note that the `nand` is "universal" in the sense that all boolean operations can be defined in terms of it. For example negation is

```
not : Boolean -> Boolean
not x = nand x x
```

> ***Exercise:*** *Define directly (by pattern matching) the "negated or" operator* `nor`.

> ***Exercise:*** *Define* `Boolean` *operators only using* `nand`*:* `and`*,* `or`*,* `imply`*,* `nor`*.*

# Pattern matching `Prod`

The standard functions on `Prod` are the so-called “projections” which extract one of the element or the other. Note the use of pattern-matching on the left of the `=`:

```
proj1 : {A B : Set} -> Prod A B -> A
proj1 (pair a b) = a

proj2 : {A B : Set} -> Prod A B -> B
proj2 (pair a b) = b
```

First note that the functions are *polymorphic*, i.e. work for any types `A`, `B`. The type of the function specifies the the type variables explicitly in `{A B : Set} ->`.

Now it is clearer why this mechanism is called “pattern matching”. All elements of `Prod A B` must have this “pattern”, `pair a b`, so whenever we are given an element of that type we can “match” it against `pair a b`. The tag `pair` is constant and it must match, in the process `a` becomes the first component and `b` the second. This way the argument is “destructed” into its constituent data.

To respect the convention that unused variables are written `_` we

should write:

```
proj1 : {A B : Set} -> Prod A B -> A
proj1 (pair a _) = a

proj2 : {A B : Set} -> Prod A B -> B
proj2 (pair _ b) = b
```

> ***Exercises:***

- Write a function `swap` which given an element of `Prod A B` creates another product using the same components but in the other order. Provide two solutions:
    - direct definition, using pattern matching
    - indirect definition, not using pattern matching but using the projections.
- Write an `and : Prod Boolean Boolean -> Boolean` function.

# Pattern matching `Sum`

The same idea applies in defining functions for `Sum` types.

Here is a simple observation that will turn out to be important: The `Unit` type has 1 element. The `Boolean` type has two elements. The type `Sum A B` has as many elements as `A` plus `B`, so `Boolean` and `Sum Unit Unit` both have two elements. Let us *convert* between these two types, and see `Sum` pattern matching at work:

---

```
f : Sum Unit Unit -> Boolean
f (left x)  = true
f (right y) = false
```

All elements of `Sum A B` (in general) must have a `left` or `right` tag, which will be used to select the appropriate `f` case. The remaining data payload of the element will be matched against the variables (`x` and `y` in this case).

Pattern-matching can go arbitrarily deep, so `x` will be matched against `Unit` and `y` against `Unit` (the two component types happen to be the same) so the function is also:

```
f : Sum Unit Unit -> Boolean
f (left empty)  = true
f (right empty) = false
```

Now lets look at the opposite function

```
g : Boolean -> Sum Unit Unit
g true  = left empty
g false = right empty
```

**Example:** Since in general `A + A = 2 * A` let's see how we can convert between `Sum A A` and `Prod Boolean A`:

```
h : {A : Set} -> Sum A A -> Prod Boolean A
h (left a)  = pair true a
h (right a) = pair false a

i : {A : Set} -> Prod Boolean A -> Sum A A
i (pair true a)  = left a
i (pair false a) = right a
```

> ***Exercise:*** *Construct conversion functions between*
>
> - `Prod Unit A` and `A`
> - `Sum A B` and `Sum B A`
> - `Prod A (Sum B C)` and `Sum (Prod A B) (Prod A C)`

# A special case

We can also pattern-match against the empty case. Consider the function

```
impossible : {A : Set} -> Void -> A
impossible v = ?
```

This function cannot produce an `a` because no actual `v` exists. However, as you may know from maths, functions $`f : \emptyset \rightarrow A`$ are well defined. There is special pattern-matching syntax for the empty pattern:

```
impossible : {A : Set} -> Void -> A
impossible ()
```

Note the lack of an `=`: we need to provide no definition.

**Example:** Convert between `Sum A Void` and `A`:

```
j : {A : Set} -> Sum A Void -> A
j (left a)   = a
j (right ())
```

There are no elements tagged with `right` so we can see the empty pattern `()` again.

> ***Exercises:***
>
> - Convert between `A` and `Sum A Void`, `Sum Void A`.
> - Convert between `Prod A Void` and `Void`.

# Types, generally

Note that arbitrary types can be created using the language of type definition, and they are always destructed via pattern matching. Consider this totally artificial example:

```
data Foo (A : Set) : Set where
```

```
  foo1 : Foo A
  foo2 : A -> A -> Foo A
  foo3 : A -> A -> Foo A

ff : {A : Set} -> Foo A -> Foo A
ff foo1 = foo1
ff (foo2 x y) = foo3 y x
ff (foo3 x _) = foo2 x x
```

> ***Exercise (hard):*** *Construct two functions that convert between the* `Foo` *type and a type formed only with* `Sum`, `Prod`, *and* `Unit`.