# 1 | **Day 1**

Functional programming comes from math!

```
fn 0 => true | x => false
```

Here's a function for absolute value.

```
fn x => if x >= 0 then x else 0 - x;
```

We can bind it to a name:

```
val abs = fn x => if x >= 0 then x else 0 - x;
abs ~4
```

We can use tuples like `(1,2,3)`:

```
val max = fn (x,y) => if x > y then x else y;
max (1,2)
```

Functions are called like `max (x,y)`.

```
val it = 2 : int
```

Every function has a single input type and single output type: to give multiple arguments you just pass a tuple.

This won't run.

```
let
  val g = fn x = x + 3
in
  if "hello" > g then 1 else false
end
```

Why? First off, SML is *statically typed* (unlike some other notable FP languages), and `"hello"` and `g` are not the same type.

What types does SML have?

- `int`
- `bool`
- `string`
- `a->b` (function)
- `a*b` (tuples)
- `type frac = int*int` (we can do type aliasing)

It also has `real`, but since floating points are… unique, they can't be compared nicely due to inaccuracies.

Let's try to make a function that checks if a tuple is `(0,0)`!

```
val thing = fn (x,y) => if x=y andalso x=0 then true else false;
thing (0,0)
```

… `andalso`? What? Apparently this is to get around the fact that `and` means something else.

We could also do it like:

```
val thing = fn (0,0) => true | x => false;
thing (0,0)
```

We can also do the Rust-style thing and have the second part be `_ => false` if we didn't care about naming the argument. This whole `something => 2 | x => 3` thing is called a *pattern*: these clauses can be a constant, number, tuple, but not an expression.

Variables are immutable, but you can "shadow" variables like Rust:

```
val abs = 2;
val abs = 3;
abs
```

Let's make a function to check if either part of a tuple is zero:

```
val either_zero = fn (0, x) => true | (x, 0) => true | _ => false;
either_zero (1,0)
```

Now, a factorial! We can allow a value to be recursive by using the `rec` keyword. Letting this not be default lets you shadow things.

```
val rec fact = fn 1 => 1 | x => x * fact(x-1);
fact 5
```

Let's make a function that checks evenness (slowly…)!

```
val rec even = fn 0 => true | x => even(x-2) | 1 => false;
```

This won't work, because it evaluates these clauses in order.

```
val rec even = fn 0 => true | 1 => false | x => even(x-2);
even 5
```

Challenge problem: write a function that takes a nonnegative integer and returns whether or not it's a multiple of 3 (without modulo!)

```
val mul3 = fn x => if (x div 3) * 3 = x then true else false;
mul3 2
```

Now, without `div`.

```
val rec f = fn x => if x - 3 = 0 then true else if x - 3 < 0 then false else f(x-3);
f 20
```

Again.

```
val rec f = fn 0 => true | 1 => true | 2 => false | x => f(x-3);
```

`if` is compiled away into pattern matches, and acts as a ternary of sorts.

Let's implement a basic math function:

```
val rec mult = fn (0, _) => 0 | (_, 0) => 0 | (x, 1) => x | (x, y) => x+mult(x, y-1);
mult (2,3)
```

We can also define our own *type aliases* (as we mentioned earlier) like so:

```
type t = int*string
```

We can also define *datatypes* (Rust enums!). Addition in this context means "this or that type":

```
datatype intparse = Success of int | Failure of string;
```

Let's try something simpler (`Option<int>` but in SML):

```
datatype intoption = Some of int | None;
val whee = Some 3
```

Sidenote: you can use a colon to do type annotations.

```
val whee: int = 3
```

We can do operations on datatypes:

```
datatype intoption = Some of int | None;
val unwrap = fn Some x => x | None => 0
```

We can have *recursive datatypes*:

```
datatype ilist = Empty | Front of (int*ilist);
val rec length = fn Empty => 0 | Front (x, xs) => 1+length xs;
```

Let's make a `map` func!

```
datatype ilist = Empty | Front of (int*ilist);
val rec map: (int->int)*ilist -> ilist = fn (f, Empty) => Empty | (f, Front(x,xs)) => Front(f x, map(f,
```

There are actual built-in lists in SML, though:

```
val rec map: (int->int)*int list -> int list = fn (f, nil) => nil | (f, x::xs) => (f x)::(map (f, xs));
```

It's time to do something known as *currying*: producing functions for us:

```
val mkadd: int->(int->int) = fn x => fn y => y+x;
mkadd 2 4 = 2+4 (* ensure our mkadd func works *)

val f: int->(int->int) = fn x => fn y => y+x;
val f': (int*int)->int = fn (x,y) => f x y;

val uncurry: (int->(int->int)) -> (int*int)->int = fn f => fn(x,y) => f x y;
val curry: (int*int->int)->(int->int->int) = fn f => fn x => fn y => f(x,y);
```

Polymorphism can be done with "generic" types that are prefixed by an apostrophe!

```
val f: 'a->'a = fn x => x;
```

The well-known `filter` operator:

```
val rec filter: ('a -> bool) -> ('a list -> 'a list) =
 fn f => (fn nil => nil | x::xs => if f x then x::filter f xs else filter f xs);
```

Trees:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;
val rec map = fn f => (fn Empty => Empty | Node (l,x,r) => Node(map f l, f x, map f r))

datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;
val rec exists: ('a->bool)->'a tree -> bool = fn f => (fn Empty => false | Node (l,x,r) => f x orelse ex
```

We can use the `op` keyword to use an infix operator like a normal function.

```
op + (1, 2)
```

Let's define a list-reversing function!

```
val rec revhelper = fn (nil, acc) => acc | (x::xs, acc) => revhelper(xs, x::acc);
val rec rev = fn L => revhelper(L, nil);

val rec helper = fn nil => nil | NONE::xs => helper xs | (SOME x)::xs => x::helper xs
val mapPartial: ('a -> 'b option) -> 'a list -> 'b list =
 fn f => helper map o f;
```

We can pattern match directly!

```
val rec mapPartial = fn f => (fn nil => nil | x::xs => case f of SOME y => y::mapPartial xs | NONE => ma
```

We have a void type, too (like Rust).

```
val test: unit = ()

val rec length = fn nil => 0 | _::xs => 1+length xs;
val rec sum = fn nil => 0 | x::xs => x+sum xs;
val rec lmax = fn nil => 0 | x::xs => max(x, lmax xs);
val rec allpos = fn nil => true | x::xs => if x >= 0 then allpos xs else false;
```

Let's automate our list creation functions. This is basically MapReduce!

```
val rec foldl = fn cmb => fn z => (fn nil => z | x::xs => foldl cmb (cmb(z,x)) xs);
val sum = foldl op + 0;

val sum = foldl op + 0;
```

What if we want to handle overflows?

```
val lmax: int list -> int option = foldl (fn (NONE, x) => x | (SOME y, x) => max(x,y)) NONE;
val lmax_unchecked = foldl max 0;
```

We can also fold from the right to the left.

```
val rec foldr = fn cmb => fn z => (fn nil => z | x::xs => cmb(x, foldr cmb z xs))

val rev = foldl op:: [];

val rec reduce: ('b*'a*'b -> 'b) -> 'b -> 'a tree -> 'b =
 fn cmb => fn z => fn Empty => z
  | Node(l, x, r) => cmb(reduce cmb z l, x, reduce cmb z r);
val size = reduce (fn (l, _, r) => l+r+1) 0;
val depth = reduce (fn (l, _, r) => max(l,r)+1) 0;

val mapreduce: = fn (f, cmb, z) => reduce cmb z o map f;
```

SML has… immutable arrays! Speed, without unsafety.

```
type 'a seq;
val length: 'a seq -> int;
val nth: 'a seq->int 'a;
val tabulate: (int->'a)->int->'a seq;
```