1 | What is FP?

Computation: taking an instruction, and simplifying it down

- $4 + 7 = 11$.
- $(5 - 3) * 2 = 4$.

We could bind varibles

```
let val x = 3 in x * 5 end
```

We could concatenate strings!

```
"Hello" ^ "World!"
```

1.1 | Functional Programming is About Functions

Functions are first class citizens

```
let val g = (fn x = x +3) in
g(10)
end
```

We could have a piecewise function

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

g

```
val it = fn : int -> bool
```

Or an absolute value function

```
fn x => if x >= 0 then x else ~1 * x
```

Or tuples! As input. Multiple arguments, therefore, is a tuple.

```
val max = fn(x,y) => if x > y then x else y;
max (12,3)
```

1.2 | Some Code Examples

Ok so here's something

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

This code crashes. Why? $g : int \to int$, and `"hello" > g` is a comparing between integer and integer.

2 | Builtins

2.1 | Types to be had

- int
- bool
- string
- a -> b
- a * b (this is **not** a *times* b, this is a TUPLE between a *times* b)

2.2 | Patters

- x: variable
- (p,p,p,p): destructured tuple
- _: throwaway
- 7, ”hello”: rvalues

3 | Messing with Numbbers

Write a function: check if both args are zero.

```
both_zero = fn (x,y) => x = y andalso x=0;
```

but no! you are bad. you are not thinking like a functional programmer. Instead, the ideomatic way is:

```
val both_zero_better = fn (0,0) => true
 | _ => false;

both_zero_better (0,2)
```

Also, this language supports shadowing. In a scope, you could overwrite the value of a variable but its only shadowed.

```
val is_either_zero = fn (0,x) => 1 | (x,0) => 1 | _ => 0;
is_either_zero (0,2);
```

Recursion works

```
val rec fact = fn 1 => 1 | x => x * fact(x-1);
fact 10
```

Write a function that takes an integer, and returns whether or not its a multiple of three.

```
7 div 3
```

Divisibility test, slow but ”better”

```
val rec threecheck = fn 0 => true | x => if x < 0 then false else threecheck (x-3);
threecheck 13
```

Written nicely, but the same thing:

```
val rec threecheck = fn 0 => true | 1 => false | 2 => false | x => threecheck (x - 3);
threecheck 12
```

Ok, squaring a number.

```
val rec multi = fn (_,0) => 0 | (x,1) => x | (x,y) => x+multi(x, y-1);
multi (4,8)
```

4 | Messing with Types!

We are going to introduce a new type of declaration

`type t = types.`

This is a typedef! Wowzies! But, there’s something better.

4.1 | Datatypes

`datatype t = ...`

What’s a datatype? If we have a type A, and some type B, we could put them together.

”Type A”, ”Type B”. Type A+B is a type that glues A and B together.

For instance, here’s a maybe int.

```
datatype intoption = SOME of int | NONE
```

Yes, this does exactly what you think it does.

```
datatype intoption = SOME of int | NONE;
SOME 3
```

So, here’s a function:

```
val zeroout = fn SOME x => x | NONE => 0;
zeroout (SOME 43);
```

4.2 | Recursive Datatypes

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

That’s a linked list!

Multiply the list together:

```
val rec prod = fn EMPTY => 1 | FRONT (x, xs) => x * prod xs;
```

4.3 | **Lists**

This is a cons list.

```
[1,2,3,4];
```

Consing looks like this:

```
4::1::nil
```

4.4 | **Currying**

Let’s write a function that makes a function!

```
val makeadd = fn x => (fn y => y+x);
makeadd 3 7 (* function that adds 3 to 7 *)
```

Recursive applications

```
val uncurry = fn f => fn (x,y)  => f x y;
val curry = fn f => fn x => fn y => f (x, y);
```

4.5 | **Something more difficult**

```
val rec filter = fn f => fn nil => nil | x::xs => if f x then x::filter f xs else filter f xs;
```

Let’s try something different. We define a list

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;
```

Let us map over this list.

```
val rec map = fn f => fn Empty => Empty | Node (l, x, r) => Node (map f l, f x, map f r);
```

Check if elements exists in a tree.

```
val rec exists = fn f => fn Empty => false | Node (l, x, r) =>  f x orelse exists f l orelse exists f r
```

Write the function in order. turns tree and sort.

5 | **Operators are functions**

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

```
(op ::) (1,nil)
```

and etc. etc.
We could also compose functions together

```
(f o g)(x) = f(g(x))
```

6 | Libraries

Let's open a few libraries with standard implementations we introduced:

```
open Int;
open Fn;
```

7 | Accumulators

Accumulators allow us to do things that we didn't do previously. For instance, defining, the reverse function:

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

val rev = fn L => revhelper(L, nil)
```

Direct pattern match!

```
val a = 3;
case a of 3 => 3 | 4 => 4 | x =>  x;
```

Unit types exists too!

```
();
```

There is only one thing in the type Unit: the type information. It does not carry any value.

8 | Write other functions with functions

For instance, let's write map reduce!
left reduce:

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

SOME 3;
```

Right reduce

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

Therefore, we could write reverse as

```
val rev = foldl op :: []
```

Reduce tree:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;
```

```
val rec reduce = fn cmb => fn z => fn Empty => z | Node (l,x,r) => cmb (reduce cmb z l, x, reduce cmb z
```

Note! If you have infinite cores, and we have perfect parallelism:

- Trees scale by depth
- Reduce scale by length
- Trees: W = O(n), S = O(log n)
- Reduce: W = O(n), S = O(n)

8.1 | **Reduce a tree**

```
val rec helper = fn acc => fn Empty => acc | Node (l,x,r) => helper (x::helper acc r) l;
```

- W = O(n)
- S = O(n)

Rule of thumb: everything that involves a list is probbaly not very paralizable

```
datatype 'a shrub = Leaf of 'a | Branch of 'a shrub * 'a shrub;
```

Shrubs can never be empty! Therefore, we could help optimize a little

8.2 | **Immutable Array**

```
type 'a seq
```

We claim that there exists a type alpha seq, and has:

- val length: ’a seq -> int
- val nth: ’a seq -> int
- val tabulate: (int -> ’a) -> int -> ’a seq