Lec 1

Philosophy of class

1. Computation is naturally functional
2. Programming is ~~explanatory~~ process

Imperative: things have states that change
$x := 5$

Functional: evaluation
$3+4 \to 7$

→ Problem statement
- Think mathtical
  - ↳ Invariants
  - ↳ Specifications
  - ↳ Proof of correctness

Parrallelism

- Work – total num of operations running on one process sequentially
- Span – running time on ∞ processors. longest / critical path length

Expressions

$(3+4)*2$
$\xrightarrow{1}$ $7*2$
$\xrightarrow{2}$ 14

Num of steps

$(3+4)*(1+1)$
$\xrightarrow{1}$ $7*(1+1)$
$\xrightarrow{2}$ $7*2$
$\xrightarrow{3}$ 14

Parallel

$(3+4)*(1+1)$
$\xrightarrow{1}$ $7*2$
$\xrightarrow{2}$ 14

"hello " ^ "world"
$\xrightarrow{1}$ "hello world"

1 + "world"
⚠ Type error

type coersion
bad idea?

Type checking !

"Certainty the most important concept in ML."

* ML type checks first. Only compiles when passes

Eg.

```
(3+4) * 1 : int
"w" ^ "d"  : string      ]- "well-typed"
1 + "world  no type , "ill typed"
```

Not even worth talking about.
Don't evaluate
They don't make sense. Don't consider

Type := prediction of the form of future value, if we ever get one.

* Every well-formed expression:
  - Has a type
  - May have a value
  - May cause an effect ← e.g. print

$e : t$

$e \to v$

T

may evaluate to value

We write:

```
(3+4) * (1+1) : int
↪ 14
```

Some types

Base types: int, real, bool, char, string, ⋮

Composite: products, functions, datatypes ← will define these later, ⋮

Type check and run

$e_1 + e_2$ int if e_1 : int and e_2 : int

* We do typing "statically"
  - but we run to get value, that's "dynamic"

Evaluation rules

one step of evaluation

$e_1 + e_2 \Rightarrow e_1' + e_2$ if $e_1 \Rightarrow e_1'$

evaluated

$n_1 + e_2 \Rightarrow n_1 + e_2'$ if $e_2 \Rightarrow e_2'$

final evaluation

$n_1 + n_2 \Rightarrow n$ where $n = e_1 + e_2$

reduce as much as possible.

$e \to v$ if $e \Rightarrow v$ and v is value

Ex. 5 div 0

type check: 5 div 0 : int ← all good

run: Div exception ← Uhoh

Extensional Equivalence (eeq)

$1+2 \hookrightarrow 3$ $\quad 0+3 \hookrightarrow 3$

They eval. to same value, they are eeq!

Write $1+2 \cong 0+3$

Def of $\cong$ is type-dependent. ⚠ function isn't like this

For most types $e_1 \cong e_2$ if:

- they have same type
- they eval to same value
  OR
  they raise same exception
  OR
  they both loop

For functions, eeq if:

- same type
- eeq result given eeq argument.

Product type

name: $t_1 * t_2$

value: (v_1, v_2)

expressions: (e_1, e_2)

$(\#_1, \#_2)$ ← some deprecated thing

typing rule: $(e_1, e_2) : t_1 * t_2$ if $e_1 : t_1$ and $e_2 : t_2$

evaluation: left to right

Ex.

(5 div 0, 2+1) : int * int
↯

(8 + "hello", false) ill typed :C
Don't even eval.

(2,(true, "a")) : int * (bool * string)

this one more memory efficient ↕ Not same

(2, true, "a") : int * bool * string

Functions

```
(* square : int -> int
   REQ : true
   ENS : square (x) evals to x*x
*)
fun square (x:int):int = x
```

Binding

Indicates "1 bound to x"

```
val x:int = 1        [1/x]
val y:int = x+1      [2/y]
val x:int = 10       [10/x]   (looks for most recent binding)
val z:int = 2*x      [20/z]
```

We don't "change" binding.
Old one "shadowed"

Local binding

```
val x : int = 1          [1/x]
let
   val x : int = 10      ← This is one expression and
in                       ↳ 20.
   x+10
end
val y:int = x            [1/y]
```