Imperative vs. Functional

Command	Expression
↓	↓
· executed	· evaluated
· has an effect	· no effect
$x := 5$	$3+4$
(state)	(value)

Computation is function.

Parallelism v.s. Sequential

one can have as many processors as one wants.

n^2

-> length of longest critical path ?

-> $\log_2(n)$

e.g. $(3+4) * (2+1)$

sequential : 3 steps

parallel : 2 steps $(3+4$ & $2+1)$

Types

a prediction about the kind of value an expression will have if it winds up to a value

- well-typed ≥1 type
- ill-typed otherwise

@ type-check ! (ML compiler)

↓

@ evaluate

expression type value

$e : t$ $\quad e \Rightarrow v$

e.g. $(3+4) * 2 : \text{int}$

$(3+4) * 2 \Rightarrow 14$

Type-check $e_1 + e_2 : \text{int}$ if $e_1 : \text{int}$ & $e_2 : \text{int}$

well-typed expression w/out a value : 5 div 0 : int

if 5>4 then 1 else 5 div 0 : int

(return 1) (short circuit)

1 div 2 : int return 0

Extensional Equivalence $\cong$

Expressions are extensionally equivalent if they have the same type and

both
- reduce to the same value
- or raise the same exception

or loop forever

Functions are ... if they map equivalent arguments to equivalent results.

eg. (fn x => x+x) $\cong$ (fn y => 2*y)

[2, 7, 6] $\cong$ [1+1, 2+5, 3+3]

21 + 21 $\cong$ 42 $\cong$ 6 * 7

Basic types : int, real, bool, char, string

Constructed types:

Products

Types $t_1 * t_2$ for type t_1, t_2

values (v_1, v_2) for values v_1, v_2

Expressions (e_1, e_2), #1 e, #2 e

e.g. (3 * 4, true) : int * bool

Functions

$f : X \to Y$ map between types X, Y

f is total if f(x) returns a value for all values x in X.

```
① (* square : int -> int
   ② REQUIRES: true
   ③ ENSURES: square(x) evaluates to x * x
 *)
④ fun square (x : int) : int = x * x
⑤ (* test cases
      ...
  *)
```

Declaration

$\underset{\uparrow\ \text{keyword}}{\text{val}}\ \underset{\uparrow\ \text{identifier}}{\text{pi}} : \underset{\nwarrow\ \text{type}}{\text{real}} = \underset{\nwarrow\ \text{value}}{3.14}$

Introduces binding of pi to 3.14 [3.14 / pi]

```
                                    val x : int = 8-5   | [3 / x]
                                    val y : int = x+1   | [4 / y]
second binding of              <--  val x : int = 10    | [10 / x]
x shadows first binding             val z : int = x+1   | [11 / z]
```

Local Declaration

let ... in ... end

```
let
    val m : int = 3
    val n : int = m * m
in
    m+1
end
```

} an expression   type ? int

value ? 12

Concrete Type Def

```
type float = real
type point = float * float
val p : point = (1.0, 2.6)
```

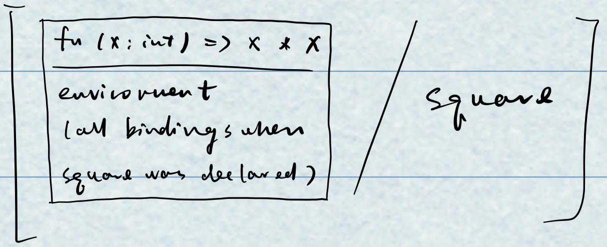
Synonym for existent types

Closures

Function declarations create value bindings.

`fun square (x: int) : int = x * x` binds the identifier square to a closure { lambda exp
environment

`fn (x : int) => x * x` Lambda expression (anonymous!)



```
fun addx (x : int) : int -> int = (fn (y: int) => x + y);
↳ val addx = fn: int -> int -> int
```

->'s are right associative! int -> int -> int same as int -> (int -> int)

```
val add5 : int -> int = addx 5;
↳ val add5 = fn : int -> int
```

Functions are values!

`fun f (x : int) = x+1` is the same as

`val f : int -> int = fn (x : int) => x+1`

Typechecking Rules

(fn (x : t_1) => body) : t_1 -> t_2

if body : t_2 assuming x : t_1

e.g. fn (x : int) => x+1 : int -> int

fn (x : real) => x+1 : real -> real

Evaluation Rules : e_2 e_1

1) Reduce e_2 to a $\underline{\text{(function) value}}$.

a closure with lambda exp & env

2) Reduce e_1 to a value v

3) Extend env with binding [v / x]

4) Evaluate body in this extended environment

Function Clauses & Pattern

```
fun f p1 = e1
  | f p2 = e2
        ⋮
  | f pk = ek
```

$\underline{pk}$ = $\underline{ek}$

pattern expression

SML will try to match v against p1, then p2 ... until a match

Pj succeeds -> SML evaluates ej.

If no pattern matches v -> fatal runtime error

pattern {
constant
variable
subpatterns
wild card _ (matches anything)

e.g. (fn (0 : int) => "good" | (1 : int) => "soso" | (_ : int) => "nope")

~~fun silly (x : int) : int = 1~~ | silly (0 : int) = 0 -> error !

redundant

$f_{-1} = 0$

fun fibb (0 : int) : int * int = (1 , 0)

Passing a function

fun sqrf (f : int -> int, x : int) : int = square (f (x))