Lec 2

- Exceptions are not considered values
- Regarding equal sign
  - Binding   val x : int = 3
  - Operators
    ⋮

* Local Binding

```
let
  val x : int = 3
  fun f(y : int) : int = y * x
  val x : int = 300
in
  f 2
end
```

still binds to this! (x in `y * x` refers to `val x : int = 3`)

x (`val x : int = 300`): Does not refer to binding when called.

Candidate answers:

- 6
- 600 ✗

* Static scope — ability to figure what sth refers to just by looking at code.
  i.e.
  meaning of variable same as when it was when function is defined.

Closures

Given binding
[3/x]

A closure
- * Code part:
  fn (y: int) => y * x
- * Environment part
  [3/x]

We get:

[A function
* Code part:
  fn (y: int) => y * x
* Environment part
  [3/x]
/ f]

whole thing bind to f

Doesn't change what's already there

Shadowing:
[300/x]

Function Types

A function ...

* name $t_1 \to t_2$
* value closure
* expressions fn (x: t_1) => body

Ways to define function:

Not required but do it to learn.

* Declaration fun f(x:int):int = x
* As expression val f = fn (x:int) = x

Typing Rules

fn (x: t_1) => body : $t_1 \to t_2$

This is true if:

body : t_2 assuming $x : t_1$

i.e. output is that type assuming input type

Evaluation Rules

Consider:

fn(x:t) => body ← ⚠ This function is a value! We don't even evaluate it! It's already a value!

Function Application

Just e_1 e_2
↑ in which this could be a function.

Then $e_1\ e_2 : t_2$ if:

- $e_1 : t_1 \rightarrow t_2$
- $e_2 : t_1$

Steps to evaluate.

Consider: e_1 e_2

1. Evaluate e_1 to obtain a function

fn x ⇒ body
[ENV]

2. Evaluate e_2 to obtain v
3. Extend environment [ENV] with [v/x]

fn x ⇒ body
[... ENV, v/x]

4. Evalue body using using new environment

```
val pi : real = 3.14

fun area (r:real):real = pi * r * r

area (2.1 + 1.9)
⇒ [3.14/pi](fn r ⇒ pi*r*r)  (2.1+1.9)
⇒ [3.14/pi](fn r ⇒ pi*r*r)  4.0
⇒ [3.14/pi, 4.0/r](fn r ⇒ pi*r*r)
   └ Extended ENV.

⇒ [3.14/pi, 4.0/r](pi*r*r)
  ⋮
⇒ 50. whatever
   ↑ ENV no longer needed. (still in `area`'s binding though)

val pi : real = 0.0        Didn't change the `pi` that's in `area`
                                      ↓
area (2.1 + 1.9)       ↪ 50. whatever
```

Recursion

Factorial in math:

$fact(0) = 1$
$fact(n) = n \times fact(n-1)$ for $n > 0$

Pattern Matching

```
(* fact : int -> int
   REQUIRES : n >= 0
   ENSURES  : fact(n) => n!
*)
fun fact (0:int):int = 1
  | fact (n:int):int = n * fact (n-1)
```

Required specs for this course

Must use fun form for the recursive referencing

omit since already written

Can't try to match against other types

ML will try to match from top to bottom.

To test:

```
val 720 = fact 6
```

To try match fact 6 to 720.

Typing Rule

```
fun f pattern1 = e1
  | f pattern2 = e2
  | f pattern3 = e3
```

See compiler errors ...

$f : t_1 \to t_2$ if
- all patterns match t_1
- all $e_i : t_2$.

Patterns

* variable x, y, ... — match anything
* constants 0, 1, 2, "hello" ← Can't match real for math reason
* tuple (patt1, patt2)
* wildcard _ — matches anything without creating binding

```
val (k,r) = (5, 2.0)
```

Fibbonacci example

```
(* fibb : int -> int * int
     REQ  n >= 0
     ENS ...
*)
fun fibb (0:int) : int * int = (1,0)
  | fibb n = let val (a:int, b:int) = fibb (n-1)
             in (a+b, a)
             end

val (21, _) = fibb(7).
```

↳ Only care about first one.

Case statement

```
(Case  e   of
     patt1  => e1
   | patt2  => e2
      ⋮
   | pattk  => ek )
```

type consistency! (the expression e and the patterns patt1 … pattk)

type consistency! (the results e1 … ek)

Always put ()!

⚠ Can't match for unspefied type since equality may not be defined.

Note that

$$\text{if } b \text{ then } e_1 \text{ else } e_2 \quad \equiv \quad \begin{array}{l}(\text{case } b \text{ of} \\ \quad \text{true} \Rightarrow e_1 \\ \quad | \text{ else} \Rightarrow e_2)\end{array}$$

```
fun example (x : int) : int =
   (case (square x, x > 0) of
         (1, true) => 0
         (sqr, false) => sqr - 1
         (sqr, _) => 1 - x * sqr
   )
```

type check

More on $\cong$

- $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$ $\Rightarrow$ $e_1 \cong e_2$
- $e_1 \Rightarrow e_2$ $\Rightarrow$ $e_1 \cong e_2$
- $e_1 \Rightarrow e \wedge e_2 \Rightarrow e$ $\Rightarrow$ $e_1 \cong e_2$

But

- $e_1 \cong e_2$ $\not\Rightarrow$ $e_1 \Rightarrow e_2 \vee e_2 \Rightarrow e_1$

On totality

if $f : int \to int$ is total,
then,

$$f(1) + f(2) \cong f(2) + f(1)$$

If not total, they could raise different exception.