

15-150
Fall 2025

Dilsun Kaynar

LECTURE 2

Functions

Last time

- Types, expressions, values
- Extensional equivalence
- Declaration

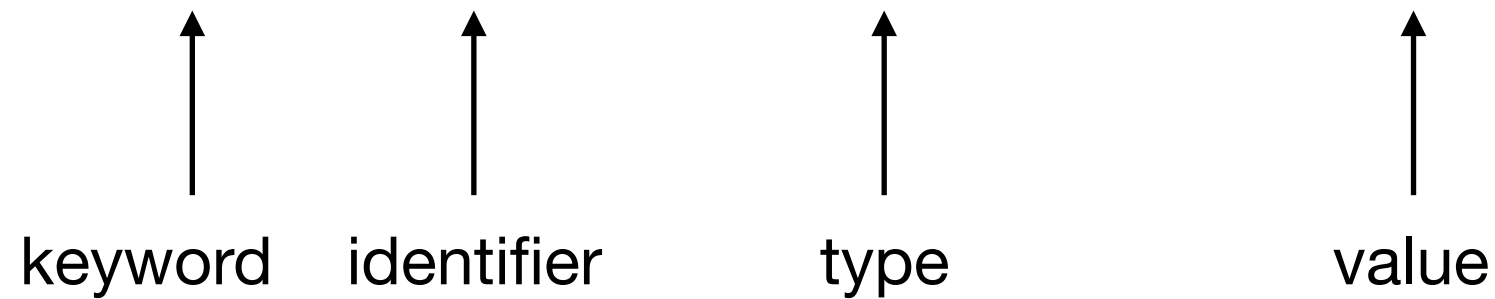
Today's Goals

- Write anonymous functions (lambda expressions)
- Declare named functions
- State what a function closure is
- Evaluate expressions that involve function application
- Use patterns
 - clausal function declarations
 - case expressions

Declarations,
Environments, Scope

Declaration

`val pi : real = 3.14`



Introduces binding of `3.14` to `pi`, sometimes written as `[3.14/pi]`

Lexically statically scoped

Bindings

```
val x : int = 8 - 5      [3 / x]  
val y : int = x + 1      [4 / y]
```

Bindings

<code>val</code>	<code>x</code>	<code>:</code>	<code>int</code>	<code>=</code>	<code>8</code>	<code>-</code>	<code>5</code>	<code>[3 / x]</code>
<code>val</code>	<code>y</code>	<code>:</code>	<code>int</code>	<code>=</code>	<code>x</code>	<code>+</code>	<code>1</code>	<code>[4 / y]</code>
<code>val</code>	<code>x</code>	<code>:</code>	<code>int</code>	<code>=</code>	<code>10</code>			<code>[10 / x]</code>
<code>val</code>	<code>z</code>	<code>:</code>	<code>int</code>	<code>=</code>	<code>x</code>	<code>+</code>	<code>1</code>	<code>[11 / z]</code>

Second binding of `x` **shadows** first binding. First binding has been *shadowed*.

Local declarations

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

This is an expression with type `int` and value 12.

Local declarations

```
val k : int = 4
```

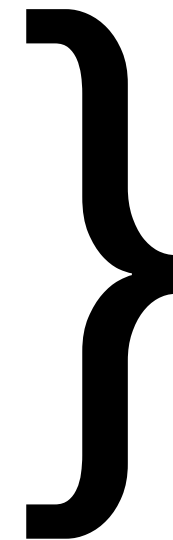
```
let
```

```
    val k : real = 3.0
```

```
in
```

```
    k * k
```

```
end
```



Type?
Value?

Local declarations

```
val k : int = 4
```

```
let
```

```
    val k : real = 3.0
```

```
in
```

```
    k * k
```

```
end
```

}

Type?
Value?

9.0 : real

k }

Type?
Value?

4 : int

Concrete Type Definitions

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

```
val p : point = (1.0, 2.6)
```

creates the binding $[(1.0, 2.6)/p]$

Functions

Function declaration

```
(* square : int -> int  
   REQUIRES: true  
   ENSURES: square(x) evaluates to x * x  
*)
```

```
fun square (x : int) : int = x * x
```

function name



function body



Function declaration

```
fun square (x : int) : int = x * x;
```

creates the binding

[???/square]

Closures

```
fun square (x : int) : int = x * x
```

binds the identifier `square` to a closure:

[ /square]

code for square, lambda expression `fn (x:int) => x * x`
environment (all prior bindings when square was declared)

Anonymous functions a.k.a. lambda

fn (x : int) => x * x



formal parameter
argument type

body

Functions are values

`fn (x : int) => x * x`

This expression is already a value —
no further evaluation happens

Alternative way to declare

```
fun square (x : int) : int = x * x;
```

is syntactic sugar for

```
val square = fn (x : int) => x * x
```

Applying a function

$$\frac{(\text{fn } (x : \text{int}) \Rightarrow x * x) \quad 7}{\text{function} \quad \text{argument}}$$

This function application evaluates to 49.

Applying a function

$$\frac{(\text{fn } (x : \text{int}) \Rightarrow x * x) \quad 7}{e_1 \quad e_2}$$

How does ML evaluate a function application $e_1 e_2$?

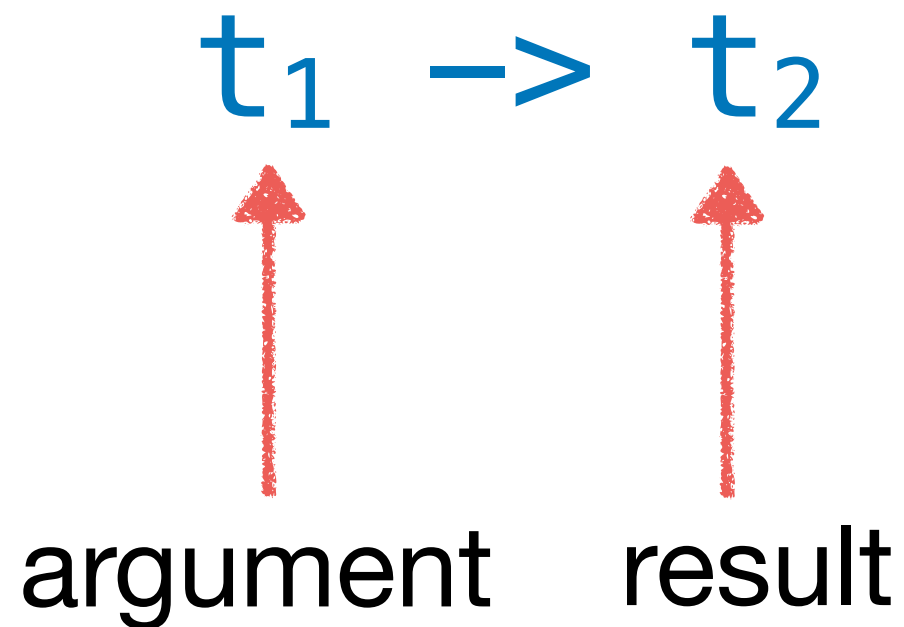
- Evaluate e_1 to a function value of the form

$$\text{fn}(x : t) \Rightarrow e$$

- Reduce e_2 to a value v
- Locally extend the environment that existed at the time of the definition of function with a binding of value v to the variable x
- Evaluate the body e in the resulting environment

Function types

Function types are of the form



Typing rules

$(\text{fn } (x : t_1) \Rightarrow e) : t_1 \rightarrow t_2$
if $e : t_2$ assuming $x : t_1$

Examples:

$(\text{fn } (x : \text{int}) \Rightarrow x) : \text{int} \rightarrow \text{int}$

$(\text{fn } (x : \text{real}) \Rightarrow x) : \text{real} \rightarrow \text{real}$

Typing rules

$(\mathbf{fn} \ (x : t_1) \Rightarrow e) : t_1 \rightarrow t_2$

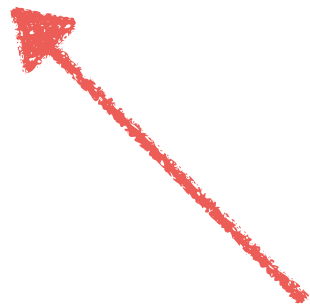
if $e : t_2$ assuming $x : t_1$

$e_1 \ e_2 : t_2$ if $e_1 : t_1 \rightarrow t_2$ and $e_2 : t_1$

Function closures

Environment together with a function expression

```
[3.14/pi](fn (r : real) => pi * r * r)
```



The environment provides the bindings of nonlocal variables

Static scope

```
val pi : real = 3.14
```

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

Function Application

```
val pi : real = 3.14
```

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

What does SML do with the following expression?

```
area (1.9 + 2.1)
```

Answer: First type-check, if well-typed then evaluate

Type-check

area (1.9 + 2.1)

area: real -> real

because area is (fn (r : real) => pi * r * r)

and pi * r * r : real

given that pi: real by its declaration

and r: real by type annotation

area (1.9 + 2.1) : real

Evaluate `area (1.9 + 2.1)`

`area (1.9 + 2.1)`

`==> [3.14/pi] (fn (r : real) => pi * r * r)(1.9 + 2.1)`

`==> [3.14/pi] (fn (r : real) => pi * r * r)(4.0)`

`==> [3.14/pi] [4.0/r] (pi * r * r)`

`==> 50.24`

Also written as

`area (1.9 + 2.1) ↦ 50.24`

Shadowing

```
val pi : real = 3.14
```

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

area 1.0 $\hookrightarrow$ 3.14

```
val pi : real = 3.14519
```

area 1.0 $\hookrightarrow$???

Shadowing

```
val pi : real = 3.14
```

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

```
area 1.0  $\hookrightarrow$  3.14
```

```
val pi : real = 3.14519
```

```
area 1.0  $\hookrightarrow$  3.14
```


Shadowing

```
val pi : real = 3.14
```

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

```
area 1.0  $\hookrightarrow$  3.14
```

```
val pi : real = 3.14519
```

```
fun new_area(r:real) : real = pi * r * r
```

```
area 1.0  $\hookrightarrow$  3.14
```

```
new_area 1.0  $\hookrightarrow$  3.14519
```

Example: 5-step methodology

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

```
fun fact(0: int): int = 1
  | fact(n: int): int = n * fact(n-1)
```

Example: 5-step methodology

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

```
fun fact(0: int): int = 1
  | fact(n: int): int = n * fact(n-1)
```

```
fact 0
```

```
fact 6
```

pattern matching determines which clause is evaluated

Patterns

- Constant (e.g. 0, “and”, no reals)
- Variable (e.g. n)
- Tuple of patterns ($p_1, \dots, p_n$)
- Based on user-defined datatypes
- The wildcard `_`

Patterns

Matching

- A *pattern* can be *matched* against a *value*
- If the match *succeeds*, it produces *bindings*

Patterns in function declarations

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


Patterns in variable declarations

```
val p = e
```

```
val (k, r) : int * real = (2, 3.14)
```

This declaration creates two variable bindings:

```
[2/k, 3.14/r]
```


Constant patterns

```
val p = e
```

```
val 49 = square (7)
```

Succeeds only if the value returned by `square` is 49.

fib

n	0	1	2	3	4	5
f_n	1	1	2	3	5	8

```
(* fib : int -> int
   REQUIRES: n >= 0
   ENSURES: fib(n) ==> fn (nth Fibonacci number)
*)
```

```
fun fib (0 : int) : int = 1
| fib (1 : int) : int = 1
| fib (n : int) = fib (n-1) + fib (n-2)
```

Efficient fib

n	0	1	2	3	4	5
f_n	1	1	2	3	5	8

```
(* fibb : int -> int * int
```

```
  REQUIRES: n >= 0
```

```
  ENSURES: fibb(n) ==> ( $f_n$ ,  $f_{n-1}$ )
```

```
  with  $f_n$  the nth Fibonacci number (let  $f_{-1} = 0$ )
```

```
*)
```

Efficient fib

n	0	1	2	3	4	5
f_n	1	1	2	3	5	8

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES: fibb(n) ==> (fn, fn-1)
   with fn the nth Fibonacci number (let f-1 = 0)
*)
```

```
fun  fibb (0:int):int * int = (1,0)
|    fibb 1 = (1,1)
|    fibb n = let
                val
                in
                end
```

Efficient fib

n	0	1	2	3	4	5
f_n	1	1	2	3	5	8

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES: fibb(n) ==> (fn, fn-1)
   with fn the nth Fibonacci number (let fn-1 = 0)
   *)
```

```
fun  fibb (0:int):int * int = (1,0)
|    fibb 1 = (1,1)
|    fibb n = let
                val (f1:int, f2:int) = fibb (n-1)
                in
                end
```

Efficient fib

n	0	1	2	3	4	5
f_n	1	1	2	3	5	8

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES: fibb(n) ==> (fn, fn-1)
   with fn the nth Fibonacci number (let fn-1 = 0)
   *)
```

```
fun  fibb (0:int):int * int = (1,0)
|    fibb 1 = (1,1)
|    fibb n = let
                val (f1:int, f2:int) = fibb (n-1)
                in
                  (f1 + f2, f1)
                end
```

Using patterns in case expressions

```
fun fact(n:int):int =  
  case n of  
    0 => 1  
  | _ => n * fact(n-1)
```


Patterns in case expressions

```
( case e : t' of
    | p1 => e1
    | ...
    | pk => ek )
```

How do we find the type of this case expression?

Example: case

```
(* example: int -> int
   REQUIRES: true
   ENSURES: example(x) returns 0 if x = 1,
              x * x - 1 if x < 1,
              and 1 - x * x * x if x > 1.
*)
```

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

Functions as first-class values

```
(* square : int -> int
   REQUIRES: true
   ENSURES: square(x) evaluates to x * x
*)
```

```
fun square (x : int) : int = x * x
```

```
(* sqrf : (int -> int) * int -> int)
   REQUIRES: true
   ENSURES: sqrf (f,x) ==> (f(x) * f(x))
*)
```

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

```
val 36 = sqrf ((fn (n:int) => n+2), 4)
```

```
(* sqrf : (int -> int) * int -> int)
   REQUIRES: true
   ENSURES: sqrf (f,x) ==> (f(x) * f(x))
*)
```

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

```
val 36 = sqrf ((fn (n:int) => n+2),4)
```

```
fun dotwice (f: int -> int, x: int): int =
```

```
(* sqrf : (int -> int) * int -> int)
   REQUIRES: true
   ENSURES: sqrf (f,x) ==> (f(x) * f(x))
*)
```

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

```
val 36 = sqrf ((fn (n:int) => n+2),4)
```

```
fun dotwice (f: int -> int, x: int): int =
    sqrf (fn (n:int) => sqrf(f,n), x)
```

```
dotwice (fn (k:int) => k, 3)  $\hookrightarrow$  ???
```

identity function

Some comments about $\cong$

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$ then $e_1 \cong e_2$.
- If $e_1 \Rightarrow e_2$, then $e_1 \cong e_2$.
- If $e_1 \Rightarrow e$ and $e_2 \Rightarrow e$, then $e_1 \cong e_2$.
- Caution: $e_1 \cong e_2$ does not necessarily

mean $e_1 \Rightarrow e_2$ or $e_1 \Rightarrow e_2$

Example: $1+1+1+7 \cong 2+8$

Some comments about $\cong$

`[3/y,5/z] (fn (x:int) => x + y + z)  $\cong$  (fn (x:int) => x + 8)`

Totality

- Issue: Computationally a function may not always return a value. This complicates equivalence checking.

Definition: A function $f : X \rightarrow Y$ is *total* if f reduces to a value and $f(x)$ reduces to a value for all values x in X .

```
fun fact(0: int): int = 1
  | fact(n: int): int = n * fact(n-1)
```

Is fact total?

Some comments about totality

Suppose $f: \text{int} \rightarrow \text{int}$

If f is total,

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