Functions
Last time

- Types, expressions, values
- Extensional equivalence, referential transparency
- Declaration, binding, environment
- Started functions
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

val p = (1.0, 2.6)

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

Create bindings
Bindings

```ocaml
val p = (1.0, 2.6)

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

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

creates the binding [???/square]
```
Function closure

• The closure for square consists of

• **Code** for square  code is a lambda expression

• The **environment** in effect when square is defined

  collection of bindings
Anonymous functions
a.k.a lambda expressions

\[
\text{fn } (x : \text{ int}) \Rightarrow x * x
\]

- formal parameter
- argument type
- body
Functions are values

\[
\text{fn } (x : \text{int}) \Rightarrow x * x
\]

This expression is already a value — no further evaluation happens.
Applying a function

\[(\text{fn } (x : \text{ int}) \Rightarrow x \times x) \ 7\]

This function application evaluates to 49.
How does ML evaluate a function application $e_1 e_2$?

- Evaluate $e_1$ to a function value $f$ 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 $f$ with a binding of value $v$ to the variable $x$
- Evaluate the body in the resulting environment
Function types

Function types are of the form

$t_1 \rightarrow t_2$

argument  result
Typing rules

(fn \(\langle x : t_1 \rangle \Rightarrow e\)): \(t_1 \Rightarrow t_2\)

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

\[
[3.14/pi](\text{fn} \ (r : \text{int}) \Rightarrow \pi * x * x * x)
\]

The environment provides the bindings of nonlocal variables
Static scope

val pi : real = 3.14  (* binds pi to 3.14 *)

fun area(r:real) : real = pi * r * r
(* binds area to a function that uses 3.14 for value of pi *)

area 1.0

Evaluation looks up pi in the environment at the time of definition of the function

(* bind it to 3.14 : real *)
Shadowing

```plaintext
val pi : real = 3.14  (* binds pi to 3.14 *)

fun area(r:real) : real = pi * r * r
(* binds area to a function that uses 3.14 for value of pi *)

area 1.0  (* val it = 3.14 : real *)

val pi : real = 3.14159  (* binds pi to 3.14159 *)

fun new_area(r:real) : real = pi * r * r
(* binds new_area to a function that uses 3.14159 for pi *)

area 1.0  (* val it = 3.14 : real *)
new_area 1.0  (* val it = 3.14159 : real *)
```
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)
5th step

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

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

val 1 = fact 0
val 720 = fact 6
Patterns

- Constant (e.g. 0)
- Variable (e.g. n)
- Tuple of patterns ($p_1, \ldots, p_n$)
- Based on user-defined datatypes
Patterns in function declarations

```
fun f p₁ = e₁
  | f p₂ = e₂
  | ...
  | f pₖ = eₖ
```
Using patterns in \texttt{case} expressions

\begin{verbatim}
fun fact(n:int):int =
    case n of
        0 => 1
        | _ => n * fact(n-1)
\end{verbatim}
Patterns

- `val p = e`

  `case e of`

  `  p₁ = > e₁`

  `  ...`

  `  pₖ = > eₖ`
Using **patterns**

- **val** \( p = e \)
- **case** \( e \) **of** \( p_1 => e_1 \mid ... \mid p_k => e_k \)
- **fn** \( p => e \)
- **fun** \( f \ p = e \)
  
or, more generally
  
  **fn** \( p_1 => e_1 \mid ... \mid p_k => e_k \)

  **fun** \( f \ p_1 = e_1 \mid ... \mid f \ p_k = e_k \)
Example

fun silly(x : int) : int = case (square x, x > 0) of
    (1, true) => 0
  | (_, false) => x
  | (n, _) => n
(* 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 \texttt{fib}

(* fibe : int -> int * int  
    REQUIRES: n \geq 0  
    Ensures: fibe(n) ==> (f_n, f_{n-1})  
    *)
Efficient \textit{fib}\

\begin{verbatim}
(* fibe : int -> int * int  
  REQUIRES: n >= 0  
  ENSURES: fibe(n) ==> (f_n,f_{n-1}) *)

fun fibe 0 = (1,0) | fibe 1 = (1,1) | fibe n = let
  val (f1,f2) = fibe (n-1)
  in
  (f1 + f2, f1)
end
\end{verbatim}