Functions

```ocaml
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
val plus_two : int -> int = <fun>
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
val plus_two : int -> int = <fun>
- : int = 16
```

First definition syntactic sugar for second

Using a nameless function

```ocaml
# (fun x -> x * 3) 5;;  (* An application *)
val (fun x -> x * 3) : (int -> int) = <fun>
- : int = 15
# ((fun y -> y +. 2.0), (fun z -> z * 3));;
(* As data *)
val (fun y -> y +. 2.0), (fun z -> z * 3) : (float -> float) * (int -> int) = (<fun>, <fun>)
```

Note: in fun v -> exp(v), scope of variable is only the body exp(v)

Values fixed at declaration time

```ocaml
# let x = 12;;
val x : int = 12
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
```

What is the result?

Values fixed at declaration time

```ocaml
# let x = 7;;  (* New declaration, not an update *)
val x : int = 7
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
```

What is the result this time?
Values fixed at declaration time
# let x = 7;; (* New declaration, not an update *)
val x : int = 7
# plus_x 3;;
What is the result this time?

Question
Observation: Functions are first-class values in this language
Question: What value does the environment record for a function variable?
Answer: a closure

Closure for plus_x
When plus_x was defined, had environment:
\[ p_{plus_x} = \{ ..., x \rightarrow 12, ... \} \]
Recall: let plus_x y = y + x
is really let plus_x = fun y -> y + x
Closure for fun y -> y + x:
\[ <y \rightarrow y + x, p_{plus_x}> \]
Environment just after plus_x defined:
\[ \{ plus_x \rightarrow <y \rightarrow y + x, p_{plus_x}> \} + p_{plus_x} \]
Functions with more than one argument

```ocaml
define add_three x y z = x + y + z

Value: add_three : int -> int -> int -> int = <fun>

let t = add_three 6 3 2

Value: t : int = 11
```

Again, first syntactic sugar for second parameter.

Functions with more than one argument

```ocaml
define add_three x y z = x + y + z

Value: add_three : int -> int -> int -> int = <fun>

let add_three = fun x -> (fun y -> (fun z -> x + y + z))

Value: <x ->fun y -> (fun z -> x + y + z), add_three>
```

Partial application of functions

```ocaml
let add_three x y z = x + y + z

let h = add_three 5 4

Value: h : int -> int = <fun>

h 3

Value: h 3 : int = 12

h 7

Value: h 7 : int = 16
```

Partial application also called sectioning.

Functions as arguments

```ocaml
let thrice f x = f (f (f x))

Value: thrice : ('a -> 'a) -> 'a -> 'a = <fun>

let g = thrice plus_two

Value: g : int -> int -> int = <fun>

# thrice (fun s -> "Hi! " ^ s) "Good-bye!"

Value: thrice (fun s -> "Hi! " ^ s) "Good-bye!" : string = "Hi! Hi! Hi! Good-bye!"
```

Tuples as Values

```ocaml
let s = (5, "hi", 3.2)

Value: s : int * string * float = (5, "hi", 3.2)
```

```ocaml
let t = (5, "hi", 3.2)

Value: t : int * string * float = (5, "hi", 3.2)
```
Pattern Matching with Tuples

\[ \rho_8 = \{ s \rightarrow (5, "hi", 3.2), \]
\[ c \rightarrow 4, \text{test} \rightarrow 3.7, \]
\[ a \rightarrow 1, b \rightarrow 5 \} \]

# let (a,b,c) = s;; (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2

# let x = 2, 9.3;; (* tuples don't require parens in Ocaml *)
val x : int * float = (2, 9.3)

### Nested Tuples

# (*Tuples can be nested *)
let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float =
((1, 4, 62), ("bye", 15), 73.95)

# (*Patterns can be nested *)
let (p,(st,_,_),_) = d;; (* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"

### Functions on tuples

# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>

# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>

# let triple_to_pair triple =
match triple
with
(0, x, y) -> (x, y)
| (x, 0, y) -> (x, y)
| (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>

### Match Expressions

# let triple_to_pair triple =
match triple
with
0, x, y -> (x, y)
| x, 0, y -> (x, y)
| x, y, _ -> (x, y);;
val triple_to_pair : int * int * int -> int * int = <fun>

### Closure for plus_pair

- Assume \( \rho_{\text{plus\_pair}} \) was the environment just before plus_pair defined
- Closure for plus_pair: \(< (n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} >\)
- Environment just after plus_pair defined:
  \( \{ \text{plus\_pair} \rightarrow < (n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} > \} + \rho_{\text{plus\_pair}} \)