Programs

- It's all about functions.

DEFINITION	USE
let x = 6 * 7	
let incr n = n + 1	incr 42
let plus x y = x + y	plus 1 2
let double n = n+n	double 6
let square n = n*n	square 3
let avg x y = (x+y)/2	avg 3 9

Data Structures: Lists

[42; 1; 55]

42 :: [1; 55]

head :: tail

[]

List.hd [1; 2; 3]

List.rev [1; 2; 3]

Data Structures: Arrays

```
[|"John"; "Doe"|]

Array.append [| 1; 2 |] [| 3; 4; 5 |]
Array.get [| 42; 51; 32 |] 2
[| 42; 51; 32 |].(2)
```

Data Structures: Strings

```
String.make 10 'x’
"Mary" ^ " and " ^ "John”
String.length "abcdefghijklmnopqstuvwxyz”
String.lowercase "MARY”
String.concat "/" [""; "usr"; "local"; "bin"]
int_of_string "546"
```

Data Structures: Tuples

```
(42, "John", true)
fst (42, "John")
snd (42, "John")
```

DEFINITION

```
let plus_and_divide (x, y, z) = (x + y) / z
let divide x y = (x / y, x mod y)
```

USE

```
plus_and_divide (1,2,3)
divide 10 3
```

Program Control

```
let max x y = if x >= y then x else y
let isempty l = if l = [] then true else false
```

DEFINITION

```
let sqavg x y = avg (square x) (square y)
let rec fact x = if x <= 1 then 1 else x * fact (x - 1)
let rec fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2)
```

USE

```
sqavg 3 9
fact 4
fib 9
```

Strong Static Typing and Type Inference

```
let double n = n+n;;
val double : int -> int = <fun>
                              let double (n:int) = n+n;;
let avg x y = (x+y)/2;;
val avg : int -> int -> int = <fun>

let sum1to n = n*(n+1) / 2;;
val sum1to : int -> int = <fun>

let max x y = if x >= y then x else y;;
val max : 'a -> 'a -> 'a = <fun>
```

Partial application and anonymous functions

```
let plus x y = x + y
let incr = plus 1;;
val incr : int -> int = <fun>

let mul x y = x * y
let double = mul 2

(fun x -> x + 1) 42
let incr = fun x -> x + 1
```

Pattern-Matching and Polymorphism

```
let isempty l = if l=[] then true else false;;
    val isempty : 'a list -> bool = <fun>

let isempty = function
  | [] -> true
  | _ -> false;;
    val isempty : 'a list -> bool = <fun>

let rec length = function
    | [] -> 0
    | h :: t -> 1 + length t;;
    val length : 'a list -> int = <fun>
```

More examples

```
let rec addtonegs a = function
    | [] -> []
   | h::t -> if (h < 0) then (a + h) :: addtonegs a t
                   else h :: addtonegs a t
      val addtonegs : int -> int list -> int list = <fun>

let rec ismember x l = if l = [] then false
                                else if x = (List.hd l) then true
                                else ismember x (List.tl l)

let rec ismember x = function
    | [] -> false
    | h :: t -> if (x=h) then true else ismember x t
```

Further Pattern Matching Examples

```
let rec addupto = function
        | 0 -> 0
        | n -> n + addupto (n - 1)

let rec listasfaras x = function
    | [] -> []
    |  h :: t -> if (x=h) then [x] else h :: listasfaras x t

let rec doublelist = function
    | [] -> []
    | h :: t -> double h :: doublelist t

let rec sumlist = function
    | [h] -> h
    |  h :: t -> h + sumlist t

let listavg l = sumlist l/length l
```

Insertion Sort

```
let rec sort = function
        | [] -> []
        | x :: l -> insert x (sort l)
and insert elem = function
        | [] -> [elem]
        | x :: l -> if elem < x then elem :: x :: l
                                 else x :: insert elem l
```

Evaluation: reduction or rewriting

```
listavg [3;7;2];;
sumlist [3;7;2] / length [3;7;2]
(3 + sumlist [7;2]) / (1 + length [7;2])
(3 + (7 + sumlist [2])) / (1 + (1 + length [2]))
(3 + (7 + 2)) / (1 + (1 + (1 + length [])))
(3 + 9) / (1 + (1 + (1 + 0)))
12 / (1 + (1 + 1))
12 / (1 + 2)
12 / 3
4
```

Higher-Order Functions (map and fold)

```
let rec incrementall = function
        |[] -> []
        | h :: t -> (h+1) :: incrementall t

List.map (fun x -> x + 1) [ 1; 2; 3; 4 ]
---
let plus = fun acc x -> acc + x
List.fold_left plus 0 [ 1; 2; 3; 4 ]

List.fold_left (fun acc x -> acc + x) 0 [ 1; 2; 3; 4 ]

plus (plus (plus (plus 0 1) 2) 3) 4
---
let doublelist = List.map double
let sqaurelist = List.map square
```

https://www.cs.cornell.edu/courses/cs3110/2009sp/lectures/lec05.html

Abstract Data Types

```
type inttree = Empty | Node of node
and node = { value: int; left: inttree; right: inttree }

Node {value=2;
        left=Node {value=1; left=Empty; right=Empty};
        right=Node {value=3; left=Empty; right=Empty}}

let rec search((t: inttree), (x:int)): bool = match t with
  Empty -> false
 | Node {value=v; left=l; right=r} ->
          v = x || search(l, x) || search(r, x)
```

https://www.cs.cornell.edu/courses/cs3110/2009sp/lectures/lec04.html

Lazy Evaluation

- OCaml is strict (eager), but laziness can be forced.

```
type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)
let rec from (n : int) : int stream =
        Cons (n, fun () -> from (n + 1))
let naturals = from 0

let rec take (s : 'a stream) (n : int) : 'a list =
if n <= 0 then [] else match s with
                Nil -> [] | _ -> hd s :: take (tl s) (n - 1)

sumlist (take naturals 10)
```