COMP 302 Crib Sheet
by Julian Lore Side 1 of 1

1 Expressions

Most basic are numbers, strings and booleans.
3 -> int
+,-,*,/ are operators for ints
3.0 -> float
+.,-.,*.,/. are operators for floats
This is because OCaml does not keep types during runtime.

```
"comp302" -> string
'a' -> char
true -> bool
true || false -> bool
true && false -> bool
```

if statements

```
if 0 = 0 then 1.0
else 2.0
if 0 = 0 then 1.0
else 'a'
```

Second line won't work with typechecker, both parts of if statement must eval to same thing.
3/0 will pass typechecker but will throw exception at runtime if it gets run.

2 Binding

variables

```
let <name> = <exp>
let x = 3 * 3
```

x is bound to 9. Values are bounded to names. It cannot be updated or changed, only overshadowed, for ex.

```
let x = 4
```

When we look up x now, we look up the most recent x on the binding stack. Garbage collector may remove previous one if there is nothing else using it (can have a function that was defined before new x but uses old x) scope

```
let <name> = <expr1> in
  <expr2>
```

scope of expr1 ends after expr2
Functions are values. Func names bind name to body.
Recursive functions declared using let rec

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
```

Tail-recursive functions are recursive funcs that have nothing to do except return final val. i.e. no need to save stack frame. Include a parameter to accumulate result

```
let rec fact_tr n =
  let rec f (n, m) =
    if n = 0 then m
    else f(n-1, n*m)
    in f(n,1)
```

3 Types

Defining a type:

```
type suit = Clubs |
  Spades | Hearts
  | Diamonds
```

Order declared does not matter. Clubs, Spades, etc are **constructors** and must begin with a **capital letter**

Recursively defined Define hand inductively. Empty is of type hand. If c is a card and h is a hand, then Hand(c, h) is a hand. Nothing else is a hand.

```
type hand = Empty |
  Hand of card * hand
let hand0:hand = Empty
let hand1:hand =
  Hand((Ace, Hearts),
  Empty)
```

Mutual recursive data type

```
type 'a forest = Forest
  of ('a tree) list
and 'a tree = Empty |
  Node of 'a * 'a forest
```

Option Data Type

```
type 'a option = None
  | Some of 'a
```

Used in case a function might not return something.

4 Pattern Matching

```
match <exp> with
| <pattern> -> <exp>
| <pattern> -> <exp>
...
```

exp we're analyzing is called *scrutinee*.

5 Arguments

Passing all args at same time: 'a * 'b -> 'c
One arg at a time: 'a -> 'b -> 'c
curry: (('a * 'b -> 'c) -> 'a -> 'b -> 'c)

```
let curry f =
  (fun x y -> f (x,y))
```

uncurry: (('a -> b' -> 'c) -> ('a * b' -> 'c))

```
let uncurry f =
  (fun (x,y) -> f x y)
```

Note that function types are right associative: 'a -> 'b -> 'c = 'a -> ('b -> 'c) and function application is left associative: f 1 2 = (f 1) 2

6 Proofs

$e \Downarrow v$: e evals to v in multiple steps (Big-Step).
$e \Rightarrow e'$: e evals in one step to e' (small-step (single))
$e \Longrightarrow{}^{*} e'$: e evals in multiples steps to e' (small-step (multiple)) Structural induction

```
type 'a list =
  nil | :: of 'a *
           'a list
```

To inductively prove about lists, prove for empty list, assume it holds for lists t and then show for lists h :: t.

```
type 'a tree =
  Empty | Node of 'a *
          'a tree *
          'a tree
```

To inductively prove about trees, prove for empty tree, assume for trees l and r and show for tree Node(a, l, r)

Inductive Proof

```
let rec r_app l1
  l2 = match l1 with
  | [] -> l2
  | x::xs -> r_app xs
  (x::l2)

  let r_app' l1
  l2 =
  let rec rev l =
  match l with |[]->[]
  |x::xs -> xs @ [x] in
  let rec app l1 l2
  = match l1 with
  |[]->l2
  |x::l1'->x::(app l1' l2)
  in app
  (rev l1) l2
```

For all lists l1 l2, r_app l1 l2= r_app' l1 l2
Proof by structural induction on l1
Base: l1 = [].
r_app l1 l2 = r_app [] l2 = l2 (by rev_app) = app [] l2 (by def of app) = app (rev []) l2 (def of rev) = app (rev l1) l2 = rev_app' l1 l2 (by def of rev_app')
Step: l1 = h :: t,
IH: For all l2, rev_app t l2 = rev_app' t l2.
rev_app' l1 l2 = rev_app (h :: t) l2 = rev_app = t (h :: l2) (by def of rev_app) = rev_app' t (h :: l2) (by IH) = app (rev t) (h :: (app [] l2)) (def app) = app (rev t) (app [h] l2) (def app) = app (app (rev t) [h]) l2 (ass of app) = app (rev (h :: t) l2) (def rev_app') = rev_app' l1 l2

7 Higher Order Functions

Used to abstract over common functionality.
Non-generic sum: int * int -> int
Generic sum with fun as arg: (int -> int) -> int * int -> int
Common hofs:

```
List.map: ('a -> 'b) ->
  'a list -> 'b list
List.filter: ('a
  -> bool)
  -> 'a list -> 'a list
(* Folds from l -> r *)
List.fold_right:
  ('a -> 'b -> 'b) ->
  'a list -> 'b -> 'b
(* Folds from r -> l *)
List.fold_left:
  ('a -> 'b -> 'a) ->
  'a -> 'b list ->'b
List.for_all:
  ('a -> bool) ->
  'a list -> bool
List.exists:
  ('a -> bool) ->
  'a list -> bool
```

Anonymous functions: (fun x -> x+1)
(function -> |_ |else), equiv to matching argument
Implementing them: map -> apply fun to head and prepend. Return empty for empty list. filter -> Prepend on tail called if true, else call again on tail.
fold_right f l b-> ret base if empty, else f h (fold_right f t b)
fold_left f l b-> ret base if nil, else fold_left on f t and (f h b), new base is f h b

8 Partial Evaluation

```
let plus x y = x + y
let plus3 = (plus 3)
let plus3' = (fun x ->
  plus x 3)
```

plus: int -> int -> int
plus3: int -> int, although it's really a fun y -> 3 + y
plus3' is really a fun x -> x + 3
Partial evaluation doesn't evaluate inside the function, just plugs in the value you gave it. If you want to force it to evaluate a certain part, you must define the part to evaluate using let z = x in (fun y -> z + y)