

# Logic and Computation Notes

William Traub

## Contents

<b>1</b>	<b>1/7, Defining Functions</b>	<b>1</b>
1.1	Course Plan . . . . .	2
<b>2</b>	<b>1/8 Functional Programming</b>	<b>3</b>
2.1	OCaml notes . . . . .	3
<b>3</b>	<b>1/12</b>	<b>4</b>
3.1	Functions . . . . .	4
3.2	Specification . . . . .	4
<b>4</b>	<b>1/14 Pattern Matching and Data</b>	<b>6</b>
4.1	Functions . . . . .	6
4.2	ADT's . . . . .	7
<b>5</b>	<b>1/15 Recap and Structural Recursion</b>	<b>9</b>
5.1	Recursive types . . . . .	9

## 1 1/7, Defining Functions

mk-requests:(ListID)

$\forall S, \forall r, \forall a, \exists R, \text{mk-requests}(S, r, a) = R$

$\forall S, \forall r, \forall a, \forall R, \text{mk-requests}(S, r, a) = R \implies \sum_{(\text{fr}, \text{to}, \text{amt})} \text{amt} = a$

$\forall S, \forall r, \forall a, a < 0 \implies \nexists R, \text{mk-requests}(S, r, a) = R$

Advantages of proofs:

- Computable - we can turn these statements into tests and proofs
- Independent - decoupled from the code itself. We don't need to worry about runtime in order to test.

Coding a property?

Because we are universally quantifying, we would generally need to enumerate over the entire set of each  $\forall$

Write a testing harness to substitute for the enumeration. Use randomness to generate test data.

## **1.1 Course Plan**

Modules:

1. Programming and PBT
2. Engineering with Abstractions
3. Analyzing Programming Languages

## 2 1/8 Functional Programming

What is functional programming?

When we write software and it gets complicated, it helps to break it down into smaller pieces. With functional programming, we get 'glue' where you are composing functions with each other.

Goal: get us to write a fizzbuzz function that takes in an int and gives a fizzbuzz string output

```
function fizzbuzz:  
    input n number  
    output string  
  
    if (n % 3) = (n % 5) = 0 return "fizzbuzz"  
    else if n % 3 = 0 return "fizz"  
    else if n % 5 = 0 return "buzz"  
    else return n as a string
```

### 2.1 OCaml notes

- in the REPL (utop) terminate expressions with ;;
- standard arithmetic, carrot is string concat
- must use +. to add floats
- use float of int to cast float to int
- double quotes for string
- not binds tighter than other
- if false then 2 else 3 (ternary must have same type)
- comparison operators, single =
- let x = 4 in x / 2
- overwriting variables instead of mutating them
- ```
let <var> = <expr> in <body> ;;  
< store, let <var> = <expr>> -> <store[<var> ;;
```

## 3 1/12

```
let x = 19 in let x = x < 10 in x ;;
(* goes to *)  
  
let x = 3 in x + x ;; (* -> 6 *)
let x = 4 in let y = 5 in x * y ;; (* 20 *)
let x = 19 in (let x = x < 10 in x) ;;
```

### 3.1 Functions

Functions are boring and everywhere.

```
(* anonymous function *)
let f = fun (x : int) -> x + 1 ;;
f : int -> int -> int
```

OCaml is right-associative so

```
let f = fun (x : int) -> fun (y : int) -> x * y ;;
let f (x : int) (y : int) = x * y ;;
```

These functions are isomorphic and will result in about the same thing.

```
let f (g : int -> int) (y : int) = g y;;
let adder = f (fun (y : int) )
```

Lexical Scope

```
let y    = 1 in
let f x = x + y in
let y    = 2 in
f 3 ;;
```

There is closure used where the function substitutes y into its body and just becomes  $f(x) = x + 1$ .

### 3.2 Specification

We have high-order functions that can take in other functions.

```
let max3 (i : int) (j : int) (k : int) =
  if i >= j
  then if i >= k
    then i
    else k
  else if l >= k
    then j
    else k
```

We want to specify the function in formal logic.

$$\begin{aligned} \forall i, j, k \in \mathbb{Z} \\ max3(i, j, k) \geq i \wedge \\ max3(i, j, k) \geq j \wedge \\ max3(i, j, k) \geq k \wedge \\ max3(i, j, k) \in \{i, j, k\} \end{aligned}$$

# 4 1/14 Pattern Matching and Data

## 4.1 Functions

```
fun (x : int -> string) -> (string -> string) (z : int)
```

Functions are everything. You can think of let bindings as function application.

```
let x = 19 in          (fun (x) ->
let y = x < 10        (fun (y) -> y
in y ;;                  ) (x < 10)
) 19
```

What is pattern matching?

```
let show (x : int) =
  if x = 1 then "one" else
  if x = 2 then "two" else
  if x = 3 then "three" else
"> three";;

let show (x : int) =
  match x with
  | 1 -> "one"
  | 2 -> "two"
  | 3 -> "three"
  | _ -> "> three";;
```

Because we have typing, we can more aggressively optimize. <https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/>

In OCaml, we have pairs or tuples. Always use pattern matching to unfold or destruct.

```
let student_of_2800 (name_stu : string * bool) : bool =
match (name, is_student) with
| (_, false) -> false
| (_, true) -> true ;;
```

You can have a nested match but you must wrap with parenthesis.

```
let shift_x (pt : int * int) (delta : int) : int * int =
  match pt with
  | (x, y) -> (x + delta, y)
  ;;

let shift_x ((x, y) : int * int) (delta : int) : int * int =
  (x + delta, y)
;;
```

**Definition 1.** Sometimes we have basic types but sometimes we want to provide a light form of documentation. We can write a **type alias** to define our own.

```
type point2d = int * int;;
let shift_x (point2d : int * int) (dx, dy : int * int) :
  point2d =
  match (xy, dxdy) with | (x, y), (dx, dy) -> (x + dx, y + dy)
  ;;
```

## 4.2 ADT's

**Definition 2.** And **Algebraic Data Type (ADT)** is a composite data type. This is similar to a union or *Enum*. An example of a sumtype.

```
type weekday =
| Monday
| Tuesday
| Wednesday
| Thursday
| Friday ;;

type two_days = (weekday * weekday);;

let next_weekday (d : weekday) : weekday =
  match d with
  | Monday  -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday  -> Monday ;;

type my_bool = | myTrue | myFalse ;;

type distance =
| Inches of float
| Feet of float
| Yards of float ;;

let d1 = Inches 10. ;;
let d2 = Yards 12. ;;

let distance_to_inches (d : distance) : float =
  match d with
  | Inches x -> x
  | Feet x -> x *. 12.
  | Yards x -> x *. 36. ;;

distance_to_inches d2 ;; (* returns *)
```

```

type quantity =
| Fraction of int * int
| Float of float
| Arbitrary of string ;;

let string_of_quantity (q : quantity) : string =
  match q with
  | Fraction (n, d) -> string_of_int n ^ "/" ^ string_of_int d
  | Float f -> string_of_float f
  | Arbitrary s -> s ;;

```

Record type in OCaml

```

type offset
= { location : point2d ; delta distance * distance}

let small_offset = {location = (3, 3)
                    ; delta = (Float 3., Fraction (1, 2))}

let _ =
  match small_offset.delta with
  | (_, Fraction (_, _)) -> print_endline "fractional"
  | (_, Float _) -> print_endline "float"
  | (_, Arbitrary _) -> print_endline "arbitrary" ;;

```

## 5 1/15 Recap and Structural Recursion

```
type int_result =
| IntOk of int
| IntError of string ;;

type person = { name : string ; age : int; registered : bool} ;;
let alice = { name = "alice" ; age = 24 ; registered = true } ;;

let string_of_person (p : person) : string =
  p.name ^ " is " string_of_int p.age ;;

let other_string_of_person { name ; age = their_age ; _ } :
  string =
  name ^ " is " string_of_int their_age ;;
```

The million dollar mistake: nullity. Can be solved by options.

```
type int_option =
| IntOpt of int
| IntOptEmpty
```

### 5.1 Recursive types

Lists in OCaml

```
[1 ; 2 ; 3]
```

Statically typed, can only have single type.

```
type int_list =
| Empty
| Cons of int * int_list

let intlist1 = Cons (2, (Cons (1, Empty)))

let rec int_with_list_sum (xs : int_list) : int =
  match xs with
  | Empty -> 0
  | Cons (x, xs') -> x + int_list_sum xs'

let rec int_list_length (xs : int_list) : int =
  | Empty -> 0
  | Cons (_x, xs) -> 1 + int_list_length xs

let rec int_list_add1 (xs : int_list) : int_list =
  | Empty -> Empty
  | Cons (x, xs) -> Cons (x * 2, int_list_add1 xs)

let rec int_list_mul2 (xs : int list) : int list =
  | [] -> []
  | x::xs -> (x * 2) :: (int_list_mul2 xs)
```

We can create a high-order function map for this

```

let rec map (f : int -> int) (xs : int list) : int list =
  match xs with
  | [] -> []
  | x::xs -> (f x) :: (map f xs)

let add1 = map (fun (x : int) -> x + 1)
let mul2 = map (fun (x : int) -> x * 2)

We can reduce or fold to find the sum of a list. We are taking a data structure and an
accumulator and apply a binary function to the acc and each element of the structure.

let rec fold (f : int -> int -> int) (acc : int) (xs : int
list) : int_list =
  match xs with
  | [] -> acc
  | (x::xs') ->
    let acc' = f acc x in
    fold f acc' xs'

let sum = fold (fun (acc : int) (x : int) -> acc + x) 0
let product = fold (fun (acc : int) (x : int) -> acc * x) 1

let rec append (xs : int list) (ys : int list) : int list =
  match xs with
  | [] -> ys
  | (x :: xs') -> x :: append xs' ys

let rec reverse (xs : int list) : int list =
  match xs with
  | [] -> []
  | (x :: xs') -> append (reverse xs') [x]

```

Functions sum and reverse are commutative.  $\forall x \in \text{int list}, \text{sum}(\text{reverse}(xs)) = \text{sum}(xs)$   
 How to test a list function.

Generating a random list: start with deciding the length.

```

let rec mk_list_of_length (len : int) : int list =
  if len <= 0 then [] else
    let hd = Random_int 500 in
    let tl = mk_list_of_length (len - 1) in
    hd :: tl

let list_gen () : int list =
  let len = Random.int 300 in
  mk_list_of_length len

```

We need to serialize a counterexample to create a tester

```

let prop_sum_rev (sum L int list -> int) (rev : int list -> int
list) : int list option =
  let xs = list_gen () in
  if sum (reverse xs) = sum xs then None else
    Some xs

```

```
let rec do_tests (sum : int list -> int) (rev : int list -> int list) (n : int) =
  if n <= 0 then
    print_endline "no issues"
  else
    match (prop_sum_rev sum rev) with
    | None -> do_tests sum rev (n - 1)
    | Some s -> print_endline("found a bug: " ^ stringify s)
```