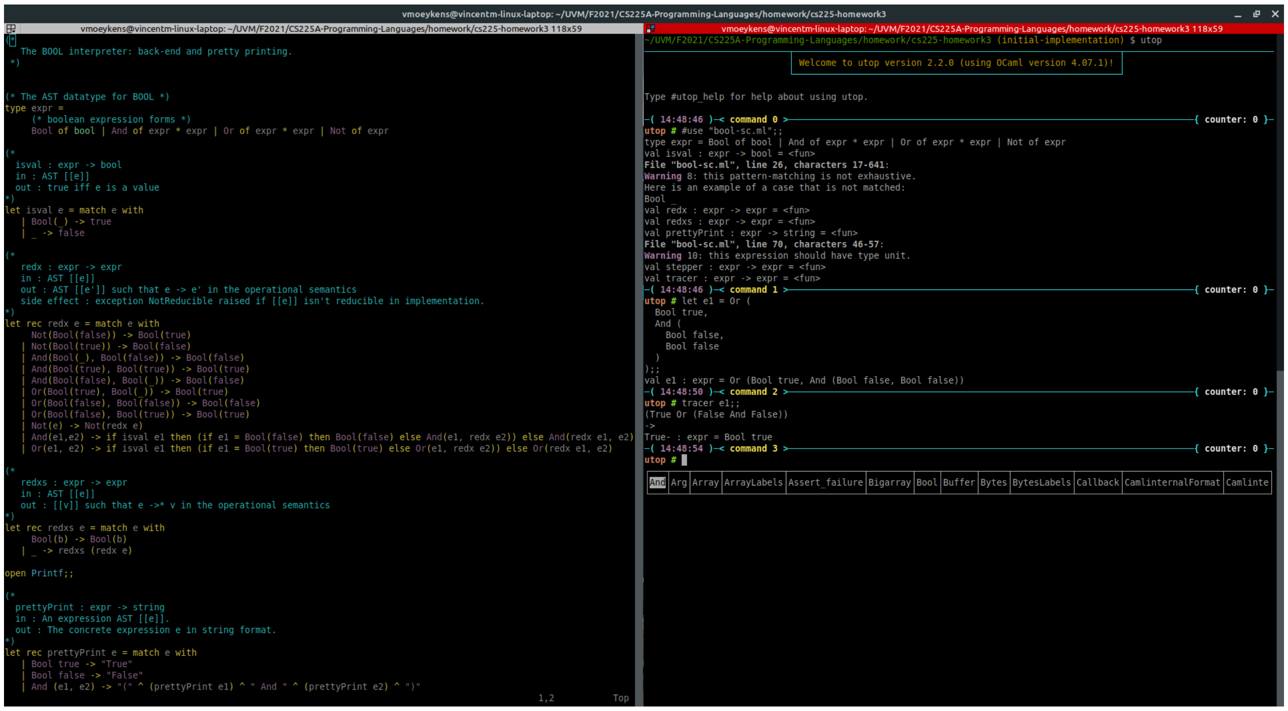
Problem 1

Question: Provide a screen shot of the OCaml editor and interpreter combination you are using for the class.
Solution: I am using the Vim editor and the `utop` interpreter.


vmoeykens@vincentm-linux-laptop: ~/UVM/F2021/CS225A-Programming-Languages/homework/cs225-homework3
vmoeykens@vincentm-linux-laptop: ~/UVM/F2021/CS225A-Programming-Languages/homework/cs225-homework3 118x59
vmoeykens@vincentm-linux-laptop: ~/UVM/F2021/CS225A-Programming-Languages/homework/cs225-homework3 118x59

```
(*
  The BOOL interpreter: back-end and pretty printing.
 *)


(* The AST datatype for BOOL *)
type expr =
     (* boolean expression forms *)
     Bool of bool | And of expr * expr | Or of expr * expr | Not of expr

(*
  isval : expr -> bool
  in : AST [[e]]
  out : true iff e is a value
*)
let isval e = match e with
   | Bool(_) -> true
   | _ -> false

(*
   redx : expr -> expr
   in : AST [[e]]
   out : AST [[e']] such that e -> e' in the operational semantics
   side effect : exception NotReducible raised if [[e]] isn't reducible in implementation.
*)
let rec redx e = match e with
     Not(Bool(false)) -> Bool(true)
   | Not(Bool(true)) -> Bool(false)
   | And(Bool(_), Bool(false)) -> Bool(false)
   | And(Bool(true), Bool(true)) -> Bool(true)
   | And(Bool(false), Bool(_)) -> Bool(false)
   | Or(Bool(true), Bool(_)) -> Bool(true)
   | Or(Bool(false), Bool(false)) -> Bool(false)
   | Or(Bool(false), Bool(true)) -> Bool(true)
   | Not(e) -> Not(redx e)
   | And(e1,e2) -> if isval e1 then (if e1 = Bool(false) then Bool(false) else And(e1, redx e2)) else And(redx e1, e2)
   | Or(e1, e2) -> if isval e1 then (if e1 = Bool(true) then Bool(true) else Or(e1, redx e2)) else Or(redx e1, e2)

(*
   redxs : expr -> expr
   in : AST [[e]]
   out : [[v]] such that e ->* v in the operational semantics
*)
let rec redxs e = match e with
     Bool(b) -> Bool(b)
   | _ -> redxs (redx e)

open Printf;;

(*
  prettyPrint : expr -> string
  in : An expression AST [[e]].
  out : The concrete expression e in string format.
*)
let rec prettyPrint e = match e with
   | Bool true -> "True"
   | Bool false -> "False"
   | And (e1, e2) -> "(" ^ (prettyPrint e1) ^ " And " ^ (prettyPrint e2) ^ ")"
                                                                              1,2           Top
```

```
~/UVM/F2021/CS225A-Programming-Languages/homework/cs225-homework3 (initial-implementation) $ utop
──────────────────────────────────────────────────────────────────────────────────────────────────
                     Welcome to utop version 2.2.0 (using OCaml version 4.07.1)!


Type #utop_help for help about using utop.

─( 14:48:46 )─< command 0 >────────────────────────────────────────────────────{ counter: 0 }─
utop # #use "bool-sc.ml";;
type expr = Bool of bool | And of expr * expr | Or of expr * expr | Not of expr
val isval : expr -> bool = <fun>
File "bool-sc.ml", line 26, characters 17-641:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Bool _
val redx : expr -> expr = <fun>
val redxs : expr -> expr = <fun>
val prettyPrint : expr -> string = <fun>
File "bool-sc.ml", line 70, characters 46-57:
Warning 10: this expression should have type unit.
val stepper : expr -> expr = <fun>
val tracer : expr -> expr = <fun>
─( 14:48:46 )─< command 1 >────────────────────────────────────────────────────{ counter: 0 }─
utop # let e1 = Or (
  Bool true,
  And (
    Bool false,
    Bool false
  )
);;
val e1 : expr = Or (Bool true, And (Bool false, Bool false))
─( 14:48:50 )─< command 2 >────────────────────────────────────────────────────{ counter: 0 }─
utop # tracer e1;;
(True Or (False And False))
->
True- : expr = Bool true
─( 14:48:54 )─< command 3 >────────────────────────────────────────────────────{ counter: 0 }─
utop # 
 And│Arg│Array│ArrayLabels│Assert_failure│Bigarray│Bool│Buffer│Bytes│BytesLabels│Callback│CamlinternalFormat│Camlinte
```

Problem 2

Question: Recall that in OCaml, record fields can be constructed from expressions that need to be evaluated. For example, given: `type rt = { a : int; b : int }` we have: `{ a = 1 + 2; b = 3 * 7 }` ⇓ `{ a = 3; b = 21 }` and it must be the case that either 1 + 2 is evaluated before 3 * 7 or vice-versa in any given implementation. Show which is the case on yours by devising an example that reveals it, and copy-paste or provide a screenshot of your interaction with the interpreter that demonstrates the evidence.

Solution: The following code helps demonstrate that the record constructor evaluates right to left. We utilze a reference and a method that takes an integer argument, adds that argument to the current record value, and then returns the record value itself. We then use this method as the value when constructing our record, and we can see that if there was a left to right evaluation order, `a` would be set to `1` since `se` begins as 0, and then `b` would subsequently be set to `3`.

```
let se = ref 0;;
type rt = { a : int; b : int };;
let inc x = (se := (x + !se); !se);;
{ a = inc 1; b = inc 2; };;
```

Instead, the output we get from the above OCaml code is the following:
`- : rt = {a = 3; b = 2}`
This shows us that `b` is evaluated first, and sets the record to 2, and then `a` is evaluated, and is set to 3.

Problem 3

Question: Suppose you wanted to add an xor operation as a primitive to BOOL. To accomplish this, do the following:
a. Propose a symbol for xor and extend the BOOL language of expressions to admit it as a binary operation
b. Extend the computational and contextual reduction rules of BOOL to provide an operational semantics for xor. You can specify either a left-to-right or right-to-left reduction semantics, but your semantics should be deterministic in the sense of Lecture Notes 9, Proposition 1.1.

Solution:

a. The proposed symbol for xor is "Xor". The new BOOL:

$$\begin{array}{lll} v & ::= \text{True} \mid \text{False} & \textit{values} \\ e & ::= v \mid e \text{ And } e \mid e \text{ Or } e \mid \text{ Not } e \mid (e) \mid e \text{ Xor } e & \textit{expressions} \end{array}$$

b. Our additional computational rules are as follows:

$$\begin{array}{rl} \textbf{Xor1} & \textbf{Xor2} \\ \text{True Xor False} \to \text{True} & \text{False Xor True} \to \text{True} \end{array}$$

$$\begin{array}{rl} \textbf{Xor3} & \textbf{Xor4} \\ \text{False Xor False} \to \text{False} & \text{True Xor True} \to \text{False} \end{array}$$

Our additional contextual rules are as follows:

$$\begin{array}{rl} \textbf{XorCL} & \textbf{XorCR} \\ \dfrac{e_1 \to e_1'}{e_1 \text{ Xor } e_2 \to e_1' \text{ Xor } e_2} & \dfrac{e \to e'}{v \text{ Xor } e \to v \text{ Xor } e'} \end{array}$$

Problem 4

Question: Is your modified interpreter still correct in the sense of Lecture Notes 10, Theorem 1.1? If you say no, provide an example that illustrates the fallacy.
Solution: Yes, the modified interpreter is still correct according to Theorem 1.1. This is because we can still generate a new expression from `redx` if the expression would normall reduce to another expression. Similarly, we can still reduce any expression to a value with any number of steps using `redxs` if the expression would have reduced to a value.