CS292C Homework 2

Deepak Nathani

April 16, 2024

1 Self-Grade

Problem	Self-Grade
1: Problem Name 1	2
2: Problem Name 2	2
3: Problem Name 3	2
4: Problem Name 4	2
5: Problem Name 5	2
6: Problem Name 6	2
7: Problem Name 7	2
8: Problem Name 8	2
9: Problem Name 9	2
10: Problem Name 10	1
11: Problem Name 11	0

2 Problems

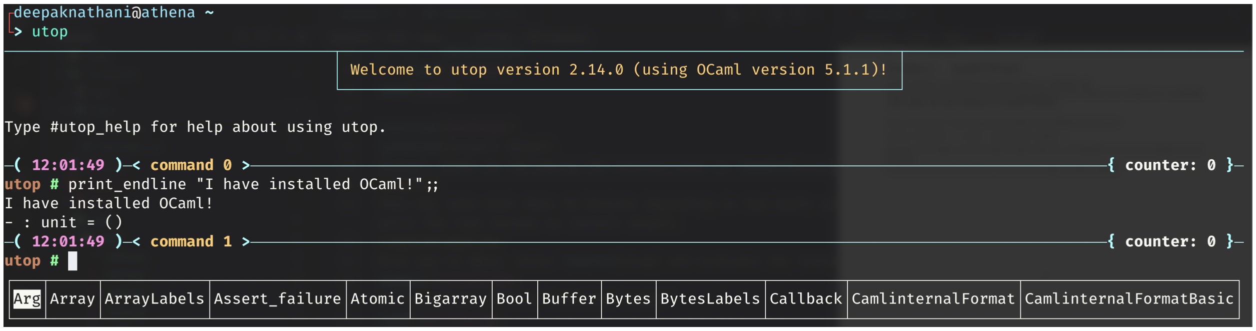
Problem 1. Install OCaml

Install OCaml by following the instructions in `install.md`. Once you're done, enter `utop` and evaluate the following expression:

```
print_endline "I have installed OCaml!"
```

Include a screenshot of all of `utop`'s output thus far (including the welcome message) to your PDF file.

Solution



```
┌deepaknathani@athena ~
└> utop
────────────────────────────────────────────────────────────────────────────────
                 │ Welcome to utop version 2.14.0 (using OCaml version 5.1.1)! │

Type #utop_help for help about using utop.

─( 12:01:49 )─< command 0 >──────────────────────────────────────{ counter: 0 }─
utop # print_endline "I have installed OCaml!";;
I have installed OCaml!
- : unit = ()
─( 12:01:49 )─< command 1 >──────────────────────────────────────{ counter: 0 }─
utop # 
│Arg│Array│ArrayLabels│Assert_failure│Atomic│Bigarray│Bool│Buffer│Bytes│BytesLabels│Callback│CamlinternalFormat│CamlinternalFormatBasic│
```

Problem 2. Compress/Duplicate Removal

Solution

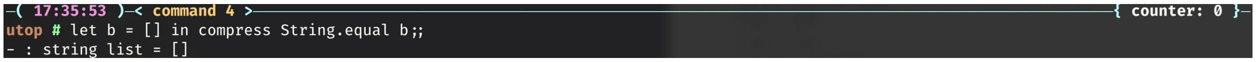
Part (a)

```
let rec compress (equal: 'a -> 'a -> bool) (l: 'a list) : 'a list =
match l with
| [] -> []
| [x] -> [x]
| x::(y::_ as t) ->
    if equal x y then
    compress equal t
    else
    x::(compress equal t)
```

Part (b)

1. **TestCase 1:** Empty string

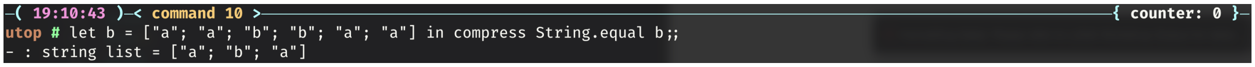
```
let b = [] in compress String.equal b;;
```



```
─( 17:35:53 )─< command 4 >──────────────────────────────────────{ counter: 0 }─
utop # let b = [] in compress String.equal b;;
- : string list = []
```

2. **TestCase 2:** String list with both consecutive and non-consecutive repeating characters.

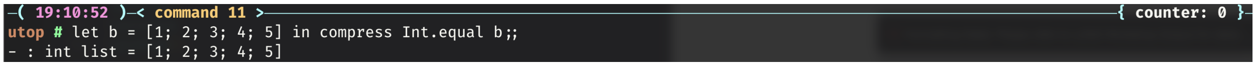
```
let b = ["a"; "a"; "b"; "b"; "a"; "a"] in compress
    ↪ String.equal b;;
```



```
─( 19:10:43 )─< command 10 >─────────────────────────────────────{ counter: 0 }─
utop # let b = ["a"; "a"; "b"; "b"; "a"; "a"] in compress String.equal b;;
- : string list = ["a"; "b"; "a"]
```

3. **TestCase 3:** Int list with no repeting characters.

```
let b = [1; 2; 3; 4; 5] in compress Int.equal b;;
```



```
─( 19:10:52 )─< command 11 >─────────────────────────────────────{ counter: 0 }─
utop # let b = [1; 2; 3; 4; 5] in compress Int.equal b;;
- : int list = [1; 2; 3; 4; 5]
```

Problem 3. Merge List

Solution

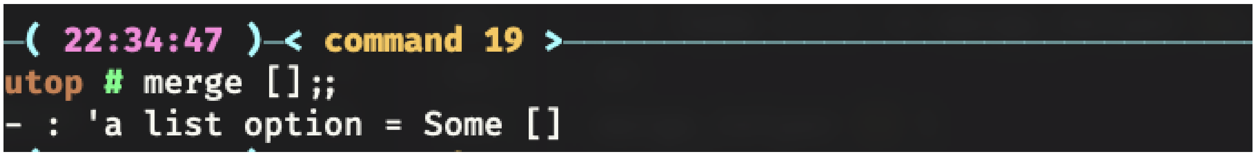
Part (a)

```
 let merge (l: 'a option list) : 'a list option =
let rec merge_helper (acc: 'a list) (l: 'a option list) =
 match l with
 | [] -> Some (List.rev acc)
 | None::_ -> None
 | Some x::t -> merge_helper (x::acc) t
in
merge_helper [] l
```

Part (b)

1. **TestCase 1:** Empty list

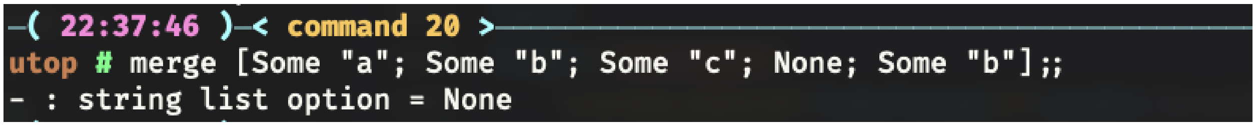
```
merge [];;
```



```
—( 22:34:47 )—< command 19 >——————————————————————————————
utop # merge [];;
- : 'a list option = Some []
```

2. **TestCase 2:** List with None

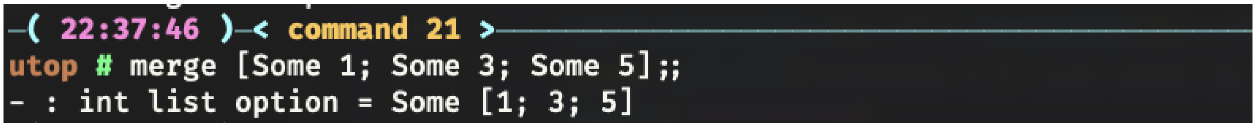
```
merge [Some "a"; Some "b"; Some "c"; None; Some "b"];;
```



```
—( 22:37:46 )—< command 20 >——————————————————————————————
utop # merge [Some "a"; Some "b"; Some "c"; None; Some "b"];;
- : string list option = None
```

3. **TestCase 3:** List with no None values

```
merge [Some 1; Some 3; Some 5];;
```



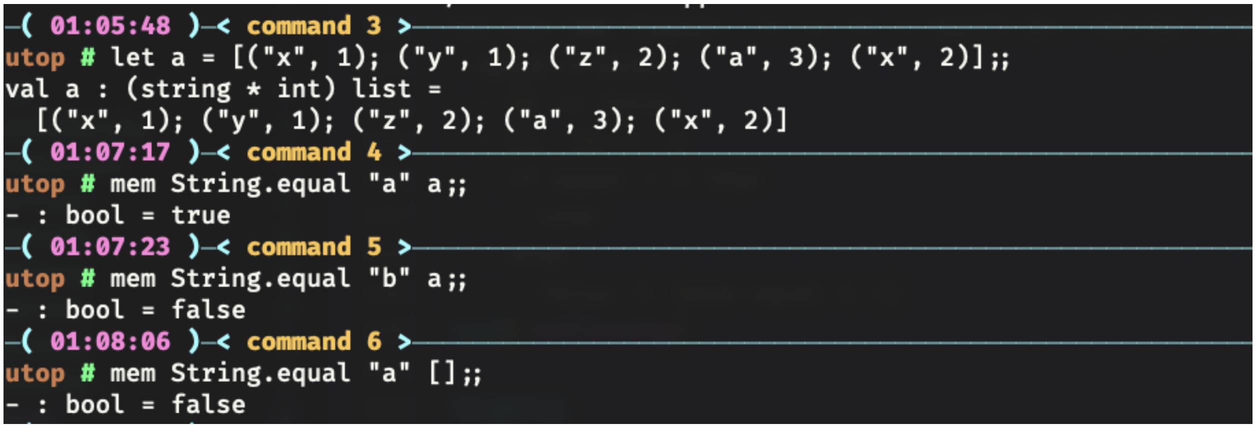
```
—( 22:37:46 )—< command 21 >——————————————————————————————
utop # merge [Some 1; Some 3; Some 5];;
- : int list option = Some [1; 3; 5]
```

Problem 4. Dictionary Functions

Solution

Part (a)

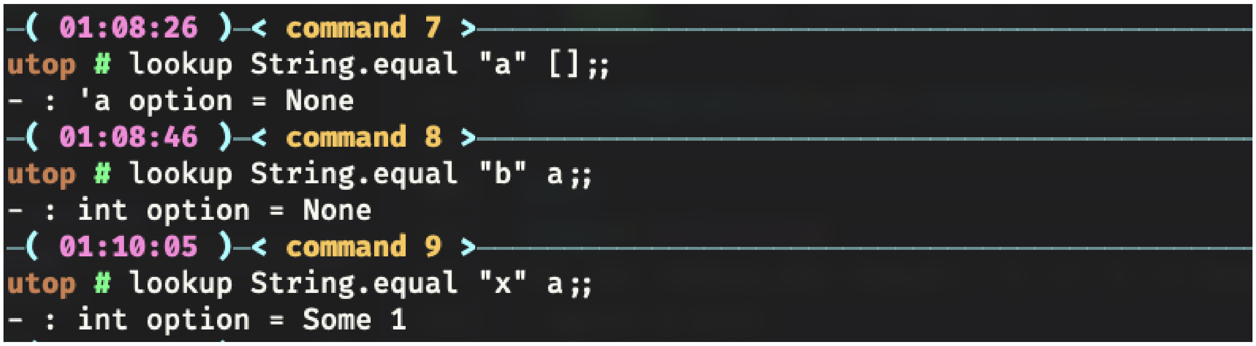
```
let rec mem (equal: 'k -> 'k -> bool) (k: 'k) (d: ('k * 'v) list) :
   ↪ bool =
  match d with
  | [] -> false
  | (l, j)::t ->
    if equal l k then
      true
    else
      false || (mem equal k t)
```



```
─( 01:05:48 )─< command 3 >─────────────────────────────────────────────
utop # let a = [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)];;
val a : (string * int) list =
  [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)]
─( 01:07:17 )─< command 4 >─────────────────────────────────────────────
utop # mem String.equal "a" a;;
- : bool = true
─( 01:07:23 )─< command 5 >─────────────────────────────────────────────
utop # mem String.equal "b" a;;
- : bool = false
─( 01:08:06 )─< command 6 >─────────────────────────────────────────────
utop # mem String.equal "a" [];;
- : bool = false
```

Part (b)

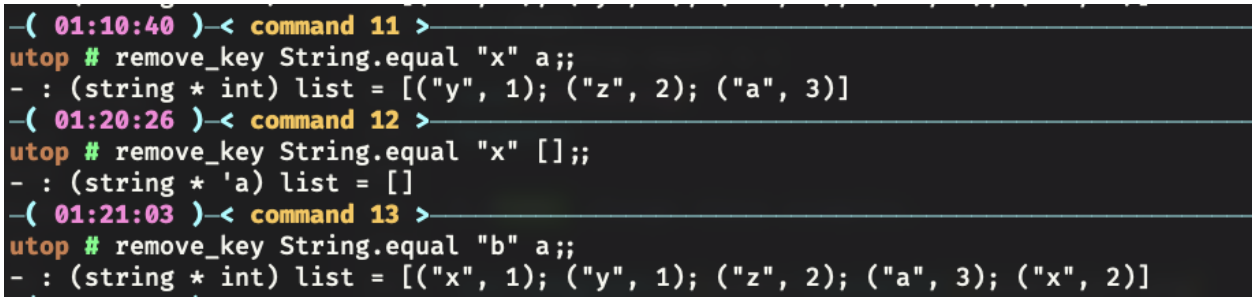
```
let rec lookup (equal: 'k -> 'k -> bool) (k: 'k) (d: ('k * 'v) list)
   ↪  : 'v option =
  match d with
  | [] -> None
  | (l, j)::t ->
    if equal l k then
      Some j
    else
      lookup equal k t
```



```
─( 01:08:26 )─< command 7 >──────────────────────────────────────────
utop # lookup String.equal "a" [];;
- : 'a option = None
─( 01:08:46 )─< command 8 >──────────────────────────────────────────
utop # lookup String.equal "b" a;;
- : int option = None
─( 01:10:05 )─< command 9 >──────────────────────────────────────────
utop # lookup String.equal "x" a;;
- : int option = Some 1
```

Part (c)

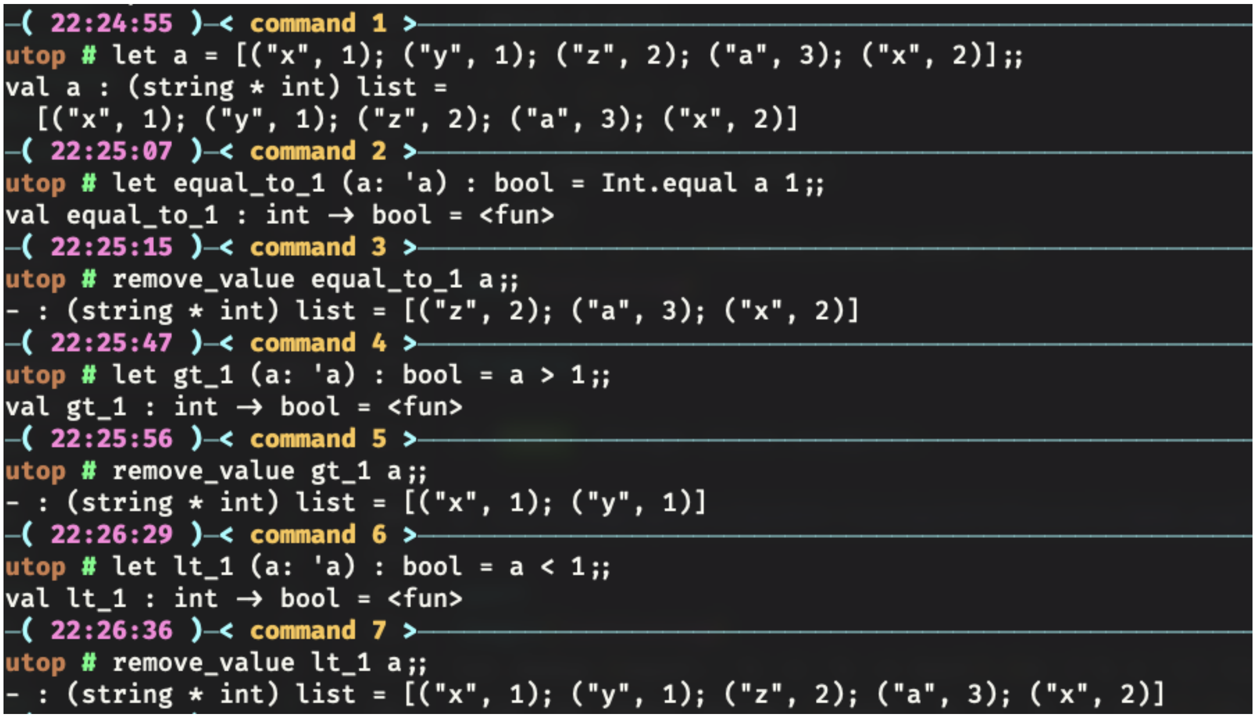
```
let rec remove_key (equal: 'k -> 'k -> bool) (k: 'k) (d: ('k * 'v)
   ↪ list) : ('k * 'v) list =
  match d with
  | [] -> []
  | (l, j)::t ->
    if equal l k then
      remove_key equal k t
    else
      (l, j) :: (remove_key equal k t)
```



```
─( 01:10:40 )─< command 11 >─────────────────────────────────────────
utop # remove_key String.equal "x" a;;
- : (string * int) list = [("y", 1); ("z", 2); ("a", 3)]
─( 01:20:26 )─< command 12 >─────────────────────────────────────────
utop # remove_key String.equal "x" [];;
- : (string * 'a) list = []
─( 01:21:03 )─< command 13 >─────────────────────────────────────────
utop # remove_key String.equal "b" a;;
- : (string * int) list = [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)]
```

Part (d)

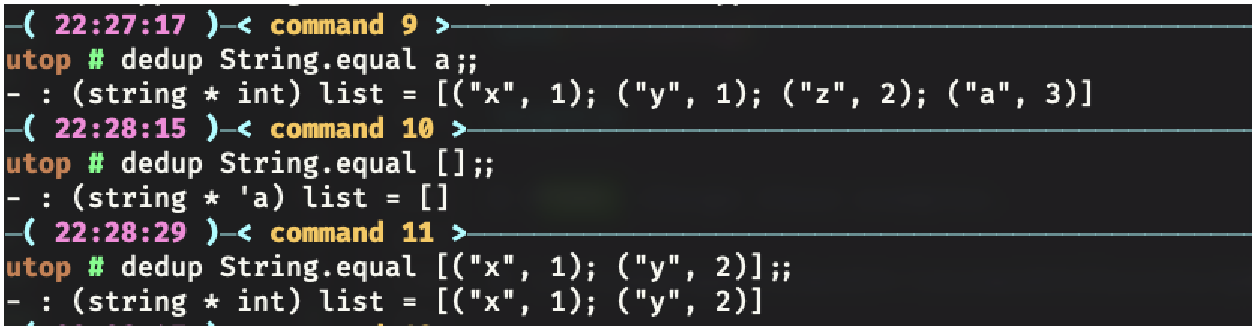
```
let rec remove_value (pred: 'v -> bool) (d: ('k * 'v) list) : ('k *
   ↪ 'v) list =
  match d with
  | [] -> []
  | (l, j)::t ->
    if pred j then
      remove_value pred t
    else
      (l, j) :: (remove_value pred t)
```



```
—( 22:24:55 )—< command 1 >——————————————————————————————————————————————————————
utop # let a = [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)];;
val a : (string * int) list =
  [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)]
—( 22:25:07 )—< command 2 >——————————————————————————————————————————————————————
utop # let equal_to_1 (a: 'a) : bool = Int.equal a 1;;
val equal_to_1 : int → bool = <fun>
—( 22:25:15 )—< command 3 >——————————————————————————————————————————————————————
utop # remove_value equal_to_1 a;;
- : (string * int) list = [("z", 2); ("a", 3); ("x", 2)]
—( 22:25:47 )—< command 4 >——————————————————————————————————————————————————————
utop # let gt_1 (a: 'a) : bool = a > 1;;
val gt_1 : int → bool = <fun>
—( 22:25:56 )—< command 5 >——————————————————————————————————————————————————————
utop # remove_value gt_1 a;;
- : (string * int) list = [("x", 1); ("y", 1)]
—( 22:26:29 )—< command 6 >——————————————————————————————————————————————————————
utop # let lt_1 (a: 'a) : bool = a < 1;;
val lt_1 : int → bool = <fun>
—( 22:26:36 )—< command 7 >——————————————————————————————————————————————————————
utop # remove_value lt_1 a;;
- : (string * int) list = [("x", 1); ("y", 1); ("z", 2); ("a", 3); ("x", 2)]
```

Part (e)

```
let dedup (equal: 'k -> 'k -> bool) (d: ('k * 'v) list) : ('k * 'v)
   ↪ list =
  let rec dedup_helper (equal: 'k -> 'k -> bool) (d: ('k * 'v) list)
     ↪  (acc: ('k * 'v) list) =
    match d with
    | [] -> (List.rev acc)
    | (l, j)::t ->
      if (mem equal l acc) then
        dedup_helper equal t acc
      else
        dedup_helper equal t ((l,j)::acc)
  in
  dedup_helper equal d []
```



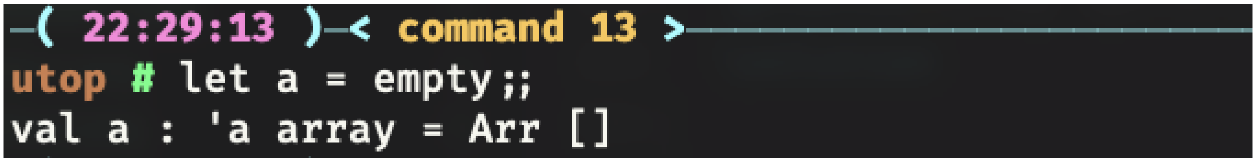
```
—( 22:27:17 )—< command 9 >——————————————————————————————————————————————————————
utop # dedup String.equal a;;
- : (string * int) list = [("x", 1); ("y", 1); ("z", 2); ("a", 3)]
—( 22:28:15 )—< command 10 >—————————————————————————————————————————————————————
utop # dedup String.equal [];;
- : (string * 'a) list = []
—( 22:28:29 )—< command 11 >—————————————————————————————————————————————————————
utop # dedup String.equal [("x", 1); ("y", 2)];;
- : (string * int) list = [("x", 1); ("y", 2)]
```

Problem 5. Array operations

Solution

Part (a)

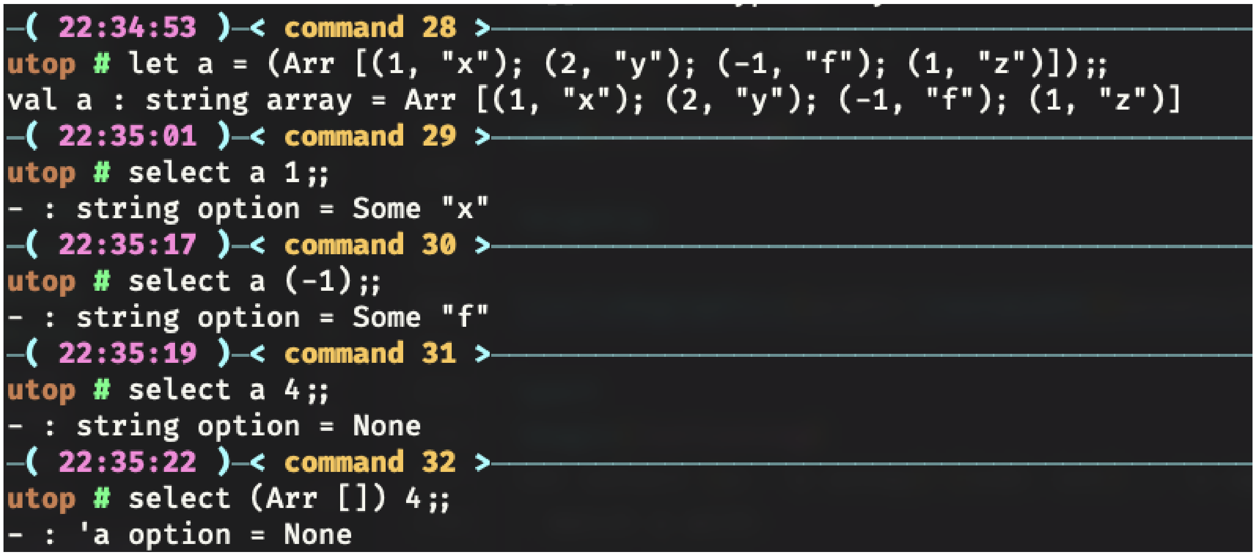
```
let empty : 'a array =
  Arr []
```



```
─( 22:29:13 )─< command 13 >──────────────────────────────
utop # let a = empty;;
val a : 'a array = Arr []
```

Part (b)

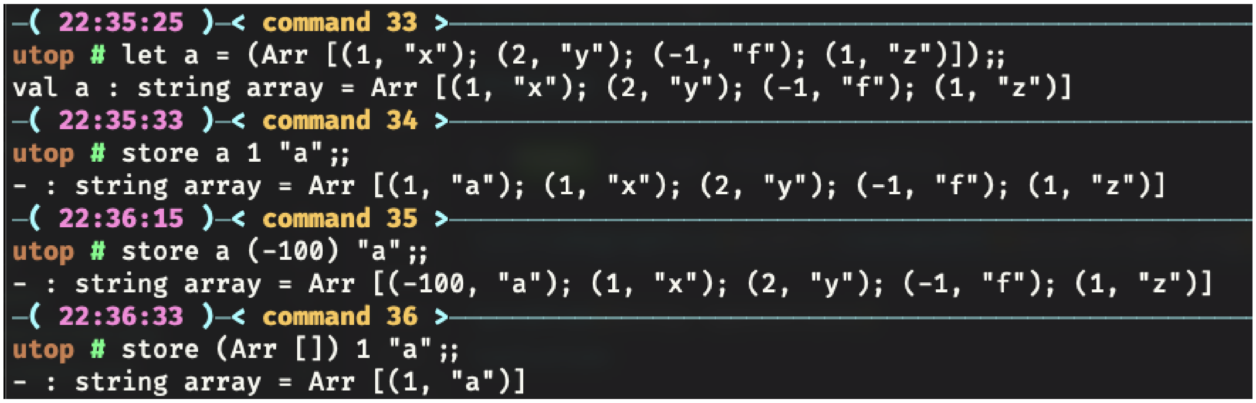
```
let select (a: 'a array) (ind: int) : 'a option =
  match a with
  | Arr [] -> None
  | Arr (x::_ as t) -> lookup Int.equal ind t
```



```
─( 22:34:53 )─< command 28 >──────────────────────────────
utop # let a = (Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]);;
val a : string array = Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]
─( 22:35:01 )─< command 29 >──────────────────────────────
utop # select a 1;;
- : string option = Some "x"
─( 22:35:17 )─< command 30 >──────────────────────────────
utop # select a (-1);;
- : string option = Some "f"
─( 22:35:19 )─< command 31 >──────────────────────────────
utop # select a 4;;
- : string option = None
─( 22:35:22 )─< command 32 >──────────────────────────────
utop # select (Arr []) 4;;
- : 'a option = None
```

Part (c)

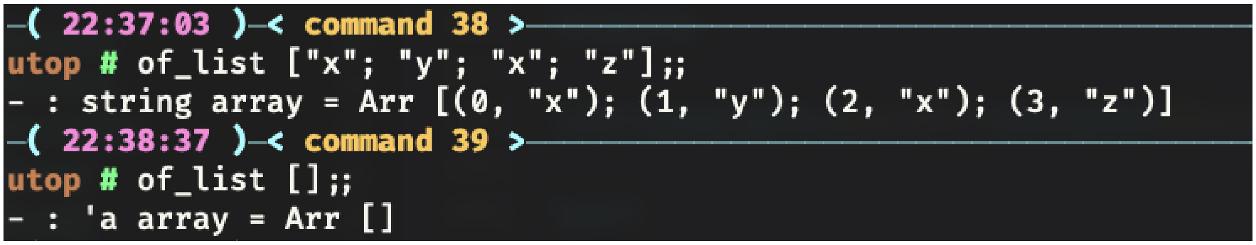
```
let store (a: 'a array) (ind: int) (value: 'a) : 'a array =
  match a with
  | Arr [] -> Arr [(ind, value)]
  | Arr (x::_ as t) -> Arr (insert ind value t)
```



```
─( 22:35:25 )─< command 33 >─────────────────────────────────────────────────
utop # let a = (Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]);;
val a : string array = Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]
─( 22:35:33 )─< command 34 >─────────────────────────────────────────────────
utop # store a 1 "a";;
- : string array = Arr [(1, "a"); (1, "x"); (2, "y"); (-1, "f"); (1, "z")]
─( 22:36:15 )─< command 35 >─────────────────────────────────────────────────
utop # store a (-100) "a";;
- : string array = Arr [(-100, "a"); (1, "x"); (2, "y"); (-1, "f"); (1, "z")]
─( 22:36:33 )─< command 36 >─────────────────────────────────────────────────
utop # store (Arr []) 1 "a";;
- : string array = Arr [(1, "a")]
```

Part (d)

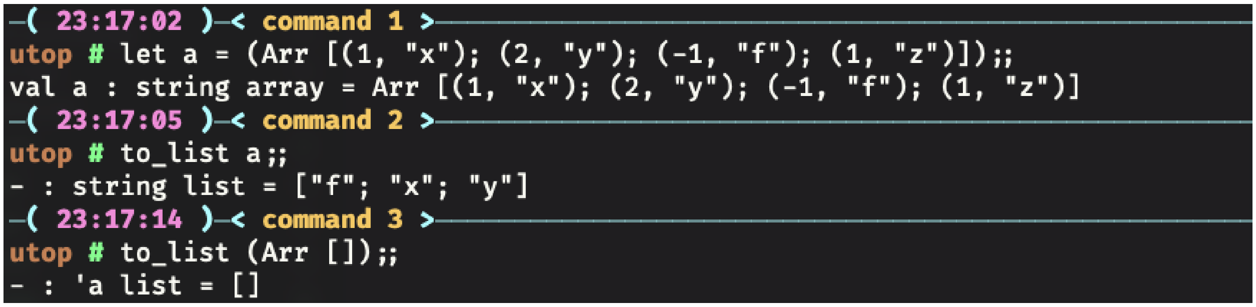
```
let of_list (l: 'a list) : 'a array =
  let d = List.mapi (fun i x -> (i, x)) l in
  Arr d
```



```
─( 22:37:03 )─< command 38 >─────────────────────────────────────────────────
utop # of_list ["x"; "y"; "x"; "z"];;
- : string array = Arr [(0, "x"); (1, "y"); (2, "x"); (3, "z")]
─( 22:38:37 )─< command 39 >─────────────────────────────────────────────────
utop # of_list [];;
- : 'a array = Arr []
```

Part (e)

```
let unique (a: (int * 'a) list) : (int * 'a) list =
  List.sort_uniq (fun (i, _) (j, _) -> compare i j) a

let to_list (a: 'a array) : 'a list =
  match a with
  | Arr [] -> []
  | Arr l -> List.map snd (unique l)
```



```
─( 23:17:02 )─< command 1 >──────────────────────────────────────────────
utop # let a = (Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]);;
val a : string array = Arr [(1, "x"); (2, "y"); (-1, "f"); (1, "z")]
─( 23:17:05 )─< command 2 >──────────────────────────────────────────────
utop # to_list a;;
- : string list = ["f"; "x"; "y"]
─( 23:17:14 )─< command 3 >──────────────────────────────────────────────
utop # to_list (Arr []);;
- : 'a list = []
```

Problem 6. Parse Concrete Syntax to Abstract Syntax Tree

Solution

Part (a)

1. **Expression**: $x + y - 10$
   AST:

   ```
   Aop(Sub, Aop(Add, Var "x", Var "y"), Int 10)
   ```

2. **Expression**: $1 - x \geq 3$
   AST:

   ```
   Comp(Geq, Aop(Sub, Int 1, Var "x"), Int 3)
   ```

3. **Expression**: *true*
   AST: *True*

4. **Expression**:

   ```
   if z < z { x := 1; } else { y := 2; }
   ```

   AST:

   ```
   If(
       Comp(Lt, Var "z", Var "z"),
       Assign("x", Int 1),
       Assign("y", Int 2)
   )
   ```

5. **Expression**:

```
x := y;
y := x;
z := x + y;
```

AST:

```
Seq(
    Assign("x", Var "y"),
    Seq(
        Assign("y", Var "x"),
        Assign("z",
            Aop(Add, Var "x", Var "y")
        )
    )
)
```

6. **Expression**:

```
while x > 0 {
    if x < 5 {
        x := x + 1;
    } else if x < 10 {
        x := x + 2;
    } else {
        x := x - 1;
    }
}
```

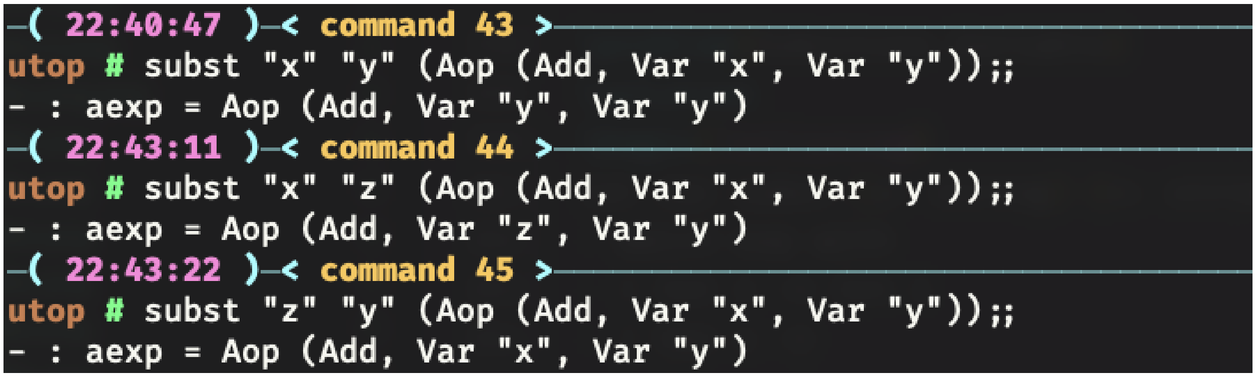
AST:

```
While(
    Comp(Gt, Var "x", Int 0),
    If(
        Comp(Lt, Var "x", Int 5),
        Assign("x", Aop(Add, Var "x", Int 1)),
        If(
            Comp(Lt, Var "x", Int 10),
            Assign("x", Aop(Add, Var "x", Int 2)),
            Assign("x", Aop(Sub, Var "x", Int 1))
        )
    )
)
```

Problem 7. Subsitute Variable

Solution

```
let rec subst (o: string) (n: string) (exp: aexp) : aexp =
  match exp with
  | Int i -> Int i
  | Var v ->
    (if String.equal v o then
      Var n
    else
      Var v)
  | Aop (aop, exp1, exp2) -> Aop (aop, (subst o n exp1), (subst o n
    ↪ exp2))
```



```
─( 22:40:47 )─< command 43 >──────────────────────────────────────────
utop # subst "x" "y" (Aop (Add, Var "x", Var "y"));;
- : aexp = Aop (Add, Var "y", Var "y")
─( 22:43:11 )─< command 44 >──────────────────────────────────────────
utop # subst "x" "z" (Aop (Add, Var "x", Var "y"));;
- : aexp = Aop (Add, Var "z", Var "y")
─( 22:43:22 )─< command 45 >──────────────────────────────────────────
utop # subst "z" "y" (Aop (Add, Var "x", Var "y"));;
- : aexp = Aop (Add, Var "x", Var "y")
```

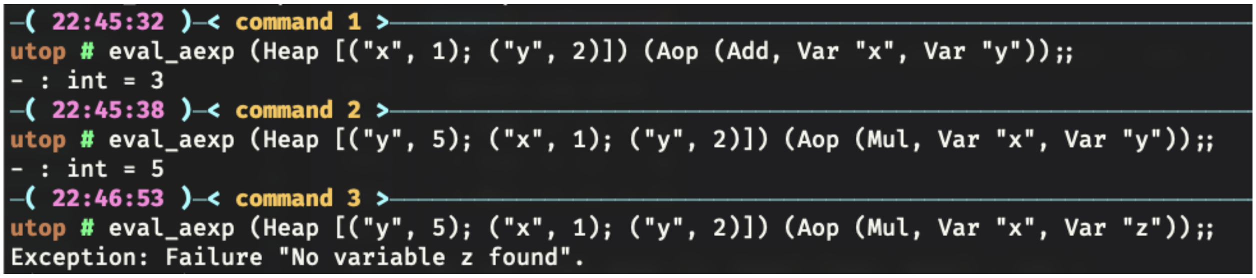
Problem 8. Evaluate Abstract Syntax Tree

Solution

Part (a)

```
let rec lookup (k: 'k) (d: heap) : int option =
  match d with
  | Heap [] -> None
  | Heap ((l, j)::t) ->
    if String.equal l k then
      Some j
    else
      lookup k (Heap t)
```

```
let insert (k: string) (v: int) (h: heap) : heap =
  match h with
  | Heap [] -> Heap [(k, v)]
  | Heap t -> Heap ((k, v) :: t)

let calculate_aop (aop: aop) (i1: int) (i2: int) : int =
  match aop with
  | Add -> i1 + i2
  | Sub -> i1 - i2
  | Mul -> i1 * i2

let rec eval_aexp (h: heap) (exp: aexp) : int =
  match exp with
  | Int i -> i
  | Var v ->
    (match (lookup v h) with
    | None -> failwith (Printf.sprintf "No variable %s found" v)
    | Some i -> i)
  | Aop (aop, exp1, exp2) -> (calculate_aop aop (eval_aexp h exp1) (
     ↪ eval_aexp h exp2))
```

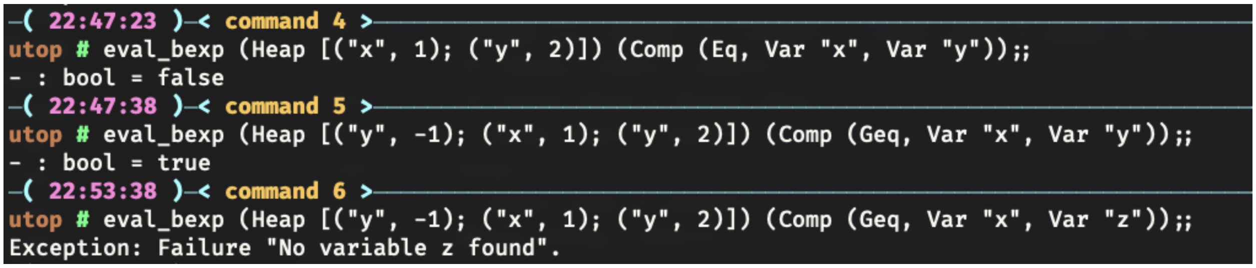


```
─( 22:45:32 )─< command 1 >──────────────────────────────────────────────
utop # eval_aexp (Heap [("x", 1); ("y", 2)]) (Aop (Add, Var "x", Var "y"));;
- : int = 3
─( 22:45:38 )─< command 2 >──────────────────────────────────────────────
utop # eval_aexp (Heap [("y", 5); ("x", 1); ("y", 2)]) (Aop (Mul, Var "x", Var "y"));;
- : int = 5
─( 22:46:53 )─< command 3 >──────────────────────────────────────────────
utop # eval_aexp (Heap [("y", 5); ("x", 1); ("y", 2)]) (Aop (Mul, Var "x", Var "z"));;
Exception: Failure "No variable z found".
```

Part (b)

```
let calculate_comp (comp: comp) (i1: int) (i2: int) : bool =
  match comp with
  | Eq -> (i1 = i2)
  | Geq -> (i1 >= i2)
  | Gt -> (i1 > i2)
  | Lt -> (i1 < i2)
  | Leq -> (i1 <= i2)
  | Neq -> (i1 <> i2)
```

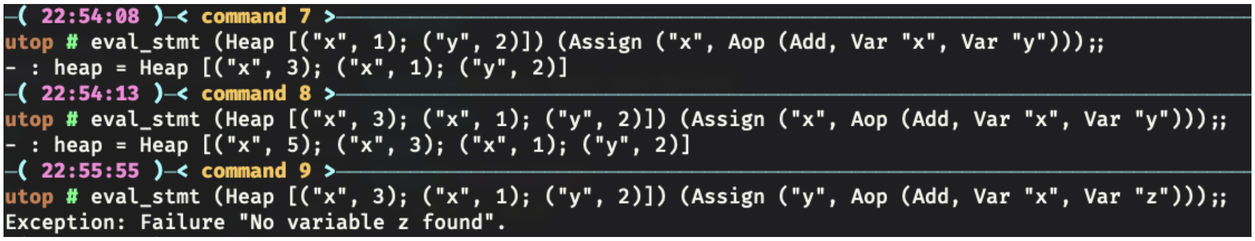
```
let rec eval_bexp (h: heap) (exp: bexp) : bool =
  match exp with
  | Bool b -> b
  | Comp (com, exp1, exp2) -> (calculate_comp com (eval_aexp h exp1)
    ↪  (eval_aexp h exp2))
  | Not exp1 ->  (not (eval_bexp h exp1))
  | And (exp1, exp2) -> ((eval_bexp h exp1) && (eval_bexp h exp2))
  | Or (exp1, exp2) -> ((eval_bexp h exp1) || (eval_bexp h exp2))
```



```
─( 22:47:23 )─< command 4 >──────────────────────────────────────────────────
utop # eval_bexp (Heap [("x", 1); ("y", 2)]) (Comp (Eq, Var "x", Var "y"));;
- : bool = false
─( 22:47:38 )─< command 5 >──────────────────────────────────────────────────
utop # eval_bexp (Heap [("y", -1); ("x", 1); ("y", 2)]) (Comp (Geq, Var "x", Var "y"));;
- : bool = true
─( 22:53:38 )─< command 6 >──────────────────────────────────────────────────
utop # eval_bexp (Heap [("y", -1); ("x", 1); ("y", 2)]) (Comp (Geq, Var "x", Var "z"));;
Exception: Failure "No variable z found".
```

Part (c)

```
let rec eval_stmt (h: heap) (stmt: stmt) : heap =
  match stmt with
  | Assign (s, aexp) -> (insert s (eval_aexp h aexp) h)
  | If (bexp, stmt1, stmt2) ->
    if (eval_bexp h bexp) then
      (eval_stmt h stmt1)
    else
      (eval_stmt h stmt2)
  | While (bexp, stmt1) ->
    if (eval_bexp h bexp) then
      (eval_stmt (eval_stmt h stmt1) (While (bexp, stmt1)))
    else
      h
  | Seq (stmt1, stmt2) -> (eval_stmt (eval_stmt h stmt1) stmt2)
```



```
─( 22:54:08 )─< command 7 >──────────────────────────────────────────────────
utop # eval_stmt (Heap [("x", 1); ("y", 2)]) (Assign ("x", Aop (Add, Var "x", Var "y")));;
- : heap = Heap [("x", 3); ("x", 1); ("y", 2)]
─( 22:54:13 )─< command 8 >──────────────────────────────────────────────────
utop # eval_stmt (Heap [("x", 3); ("x", 1); ("y", 2)]) (Assign ("x", Aop (Add, Var "x", Var "y")));;
- : heap = Heap [("x", 5); ("x", 3); ("x", 1); ("y", 2)]
─( 22:55:55 )─< command 9 >──────────────────────────────────────────────────
utop # eval_stmt (Heap [("x", 3); ("x", 1); ("y", 2)]) (Assign ("y", Aop (Add, Var "x", Var "z")));;
Exception: Failure "No variable z found".
```

Problem 9. Array Read and Write

Solution

Part (a)

```
x = a[i] * a[j]

Reads:
Path ("a", [Var "i"])
Path ("a", [Var "j"])

AST:
Assign (
  "x",
  Aop (
    Mul,
    Select (Var "a", Var "i"),
    Select (Var "a", Var "j")
  )
)
```

Part (b)

```
y = a[a[i]]

Reads:
Path ("a", [Select (Var "a", Var "i")])

AST:
Assign (
  "y",
  Select (
    Var "a",
    Select (
      Var "a",
      Var "i"
    )
  )
)
```

Part (c)

```
a[x - y] = z

Writes:
Path ("a", [Aop (Sub, Var "x", Var "y")])
```

```
AST:
Assign (
  "a",
  Store (
    Var "a",
    Aop (Sub, Var "x", Var "y"),
    Var "z"
  )
)
```

Part (d)

```
a[i + j] = a[i] + a[j];

Write:
Path ("a", [Aop (Add, Var "i", Var "j")])

AST:
Assign (
  "a",
  Store (
    Var "a",
    Aop (Add, Var "i", Var "j"),
    Aop (Add, (Select (Var "a", Var "i")), (Select (Var "a", Var "j
       ↪ ")))
  )
)
```

Part (e)

```
a[a[i]] = y

Writes:
Path ("a", [Select (Var "a", Var "i")])

AST:
Assign (
  "a",
  Store (
    Var "a",
    (Select (Var "a", Var "i")),
    Var "y"
  )
)
```

Part (f)

```
a[a[i] + a[j]] = a[a[i] * a[j]]

Reads:
Path ("a", [Aop (Mul, (Select (Var "a", Var "i")), (Select (Var "a",
   ↪  Var "i"))))])

Writes:
Path ("a", [Aop (Add, (Select (Var "a", Var "i")), (Select (Var "a",
   ↪  Var "j"))))])


AST:
Assign (
  "a",
  Store (
    Var "a",
    Aop (Add, (Select (Var "a", Var "i")), (Select (Var "a", Var "j
       ↪ ")))),
    Select (
      Var "a",
      Aop (Mul, (Select (Var "a", Var "i")), (Select (Var "a", Var "
         ↪ j")))
    )
  )
)
```

Part (g)

```
a[i][j] = a[j][i]

Reads:
Path ("a", [Var "j"; Var "i"])

Writes:
Path ("a", [Var "i"; Var "j"])

AST:
Assign (
  "a",
  Store (
    Var "a",
    Var "i",
    Store (
      (Select (Var "a", Var "i")),
      Var "j",
      (Select (Select (Var "a", Var "j"), Var "i"))
    )
  )
)
```

Part (h)

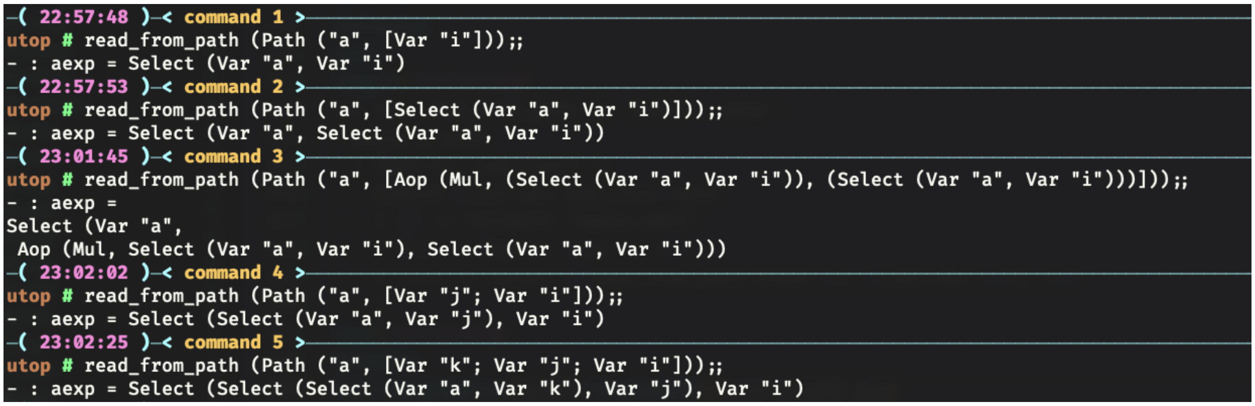
```
a[i][j][k] = a[k][j][i]

Reads:
Path ("a", [Var "k"; Var "j"; Var "i"])

Writes:
Path ("a", [Var "i"; Var "j"; Var "k"])

AST:
Assign (
  "a",
  Store (
    Var "a",
    Var "i",
    Store (
      (Select (Var "a", Var "i")),
      Var "j",
      Store(
        (Select (Select (Var "a", Var "i"), Var "j")),
        Var "k",
        (Select (Select (Select (Var "a", Var "k"), Var "j"), Var "i
            ↪ "))
      )
    )
  )
)
```

Problem 10. Read and Write from Path

Solution

Part (a)

```
let rec read_from_path (p: path) : aexp =
  match p with
  | Path (v, ind_list) ->
    (match (List.rev ind_list) with
    | [] -> failwith "Empty path"
    | [i] -> (Select (Var v, i))
    | i::t -> Select ((read_from_path (Path (v, (List.rev t))), i))
    )
```



```
─( 22:57:48 )─< command 1 >────────────────────────────────────────────────────
utop # read_from_path (Path ("a", [Var "i"]));;
- : aexp = Select (Var "a", Var "i")
─( 22:57:53 )─< command 2 >────────────────────────────────────────────────────
utop # read_from_path (Path ("a", [Select (Var "a", Var "i")]));;
- : aexp = Select (Var "a", Select (Var "a", Var "i"))
─( 23:01:45 )─< command 3 >────────────────────────────────────────────────────
utop # read_from_path (Path ("a", [Aop (Mul, (Select (Var "a", Var "i")), (Select (Var "a", Var "i")))]));;
- : aexp =
Select (Var "a",
 Aop (Mul, Select (Var "a", Var "i"), Select (Var "a", Var "i")))
─( 23:02:02 )─< command 4 >────────────────────────────────────────────────────
utop # read_from_path (Path ("a", [Var "j"; Var "i"]));;
- : aexp = Select (Select (Var "a", Var "j"), Var "i")
─( 23:02:25 )─< command 5 >────────────────────────────────────────────────────
utop # read_from_path (Path ("a", [Var "k"; Var "j"; Var "i"]));;
- : aexp = Select (Select (Select (Var "a", Var "k"), Var "j"), Var "i")
```

Part (b)

```
let write_to_path (p: path) (aexp: aexp) : stmt =
  let rec aux (v: string) (ind_list: aexp list) (aexp: aexp) (
     ↪ read_path: aexp list) : aexp =
    match ind_list with
    | [] -> failwith "Empty Path"
    | [i] -> (Store (Var v, i, aexp))
    | i::t -> (Store ((read_from_path (Path (v, read_path @ [i]))),
       ↪ (aux v t aexp (read_path @ [i])), aexp))
  in
  match p with
  | Path (v, t) -> Assign (v, (aux v t aexp []))
```