HW1.ML

```
open List;;

let rec subset a b = match a with
        |[]->true
        |_ ->if mem (hd a) b then subset (tl a) b else false;;

let equal_sets a b =
        if subset a b && subset b a then true else false;;

let rec set_union a b = match a with
        |[]->b
        |_ -> if mem (hd a) b then set_union (tl a) b else (hd a)::
        set_union (tl a) b;;

let rec set_intersection a b =  match a with
        |[]->[]
        |_-> if mem (hd a) b then (hd a):: (set_intersection (tl a) b)
                else set_intersection (tl a) b;;

let rec set_diff a b = match a with
        |[]->[]
        |_-> if mem (hd a) b then set_diff (tl a) b
                else (hd a)::set_diff (tl a) b;;

let rec computed_fixed_point eq f x =
        if eq x (f x) then x else computed_fixed_point eq f (f x);;

type ('nonterminal, 'terminal) symbol =
  | N of 'nonterminal
  | T of 'terminal;;

(* hint : use equal_sets and computed_fixed_point*)



let t_f_e (a,b) = a;;
let t_s_e (a,b) = b;;

(*note: inputs are tuples not lists*)
let equal_second_elem_sets a b =
        equal_sets (t_s_e a) (t_s_e b);;
```

```
(*let test_non_terminal a = match a with
        | N n -> true
        | T t -> false;;*)

(*let filter_non_terminal a = filter test_non_terminal a;;*)

(*let rec filter_non_terminal a = match a with
        | [] -> []
        | _ -> if test_non_terminal (hd a) then (hd a)::filter_non_terminal (tl a)
                        else filter_non_terminal (tl a);;*)

let rec filter_non_terminal a = match a with
        []->[]
        | N head::tail -> head::filter_non_terminal tail
        | T head::tail -> filter_non_terminal tail;;


(*a:rules, a list of tuples of (symbol,symbol list)
  b: reachable symbols, a list*)

 (*todo don't change a*)
let rec grshelper (a,b) = match (a,b) with
        | ([],b)-> (a,b)
        | _-> if mem (t_f_e (hd a)) b
                then grshelper ((tl a),(b @ (filter_non_terminal (t_s_e (hd a)))))
                else grshelper ((tl a) ,b) ;;

let get_reachable_symbols (a,b) =
        (a,t_s_e (grshelper (a,b)));;


(*keep only the reachable rules in grammar g
   parameters:
      g: a pair, first element is a symbol, second element is a list of rules
   return:
      grammar after filtered the symbols
   sample usage: filter_reachable(sample_grammar)*)

let rec filter_reachable g = match g with (symbol,rules)->
        let (a,b) = computed_fixed_point equal_second_elem_sets get_reachable_symbols
(rules,[symbol]) in
        let filtered_rules = filter (fun x -> mem (t_f_e x) b )  rules in
        (symbol, filtered_rules);;
```

HW1 TEST

```
let my_subset_test0 = subset [] [1;2;3;4;5]
let my_subset_test1 = subset [1;2;2;2;2] [1;2;3]

let my_equal_sets_test0 = equal_sets [1;1] [1;1;1]
let my_equal_sets_test1 = equal_sets [1;2;2;2;2] [1;2]

let my_set_union_test0 = equal_sets (set_union [1;2;3] [4;5]) [1;2;3;4;5]
let my_set_union_test1 = equal_sets (set_union [3;1;3] [1;2;3;5]) [1;2;3;5]
let my_set_union_test2 = equal_sets (set_union [1] []) [1]

let my_set_intersection_test0 =
  equal_sets (set_intersection [] [1;2;3;3]) []
let my_set_intersection_test1 =
  equal_sets (set_intersection [2;2;2;2] [3;2]) [2]
let my_set_intersection_test2 =
  equal_sets (set_intersection [2] []) []

let my_set_diff_test0 = equal_sets (set_diff [1;2;3] [1]) [2;3]
let my_set_diff_test1 = equal_sets (set_diff [1;2;2;2] [1;2]) []
let my_set_diff_test2 = equal_sets (set_diff [4;3;1] []) [1;3;4]
let my_set_diff_test3 = equal_sets (set_diff [] [4;3;1]) []

let my_computed_fixed_point_test0 =
  computed_fixed_point (=) (fun x -> x ) 0= 0
let my_computed_fixed_point_test1 =
  computed_fixed_point (=) (fun x -> x*x) 1 = 1
let my_computed_fixed_point_test2 =
  computed_fixed_point (=) sqrt 10. = 1.


(* An example grammar for a small subset of Awk.  *)

type awksub_nonterminals =
  | Expr | Lvalue | Incrop | Binop | Num

let awksub_rules =
   [Expr, [T"("; N Expr; T")"];
    Expr, [N Num];
    Expr, [N Expr; N Binop; N Expr];
    Expr, [N Lvalue];
    Expr, [N Incrop; N Lvalue];
```

```
  Expr, [N Lvalue; N Incrop];
  Lvalue, [T"$"; N Expr];
  Incrop, [T"++"];
  Incrop, [T"--"];
  Binop, [T"+"];
  Binop, [T"-"];
  Num, [T"0"];
  Num, [T"1"];
  Num, [T"2"];
  Num, [T"3"];
  Num, [T"4"];
  Num, [T"5"];
  Num, [T"6"];
  Num, [T"7"];
  Num, [T"8"];
  Num, [T"9"]]

let awksub_grammar = Expr, awksub_rules

let my_filter_reachable_test0 =
  filter_reachable awksub_grammar = awksub_grammar

let my_filter_reachable_test1 =
  filter_reachable (Binop, awksub_rules) = (Binop, [ (Binop, [T"+"]); (Binop, [T"-"])])

let my_filter_reachable_test2 =
  filter_reachable (Incrop, awksub_rules) = (Incrop, [ (Incrop, [T"++"]); (Incrop, [T"--"])])
```

REPORT

The first five functions are pretty trivial. Basically all I need to do is to think about how I can revursively implment it and then express it in terms of Ocaml functions. I had experience in Lisp before so this part is not too bad for me. Basically all I needed to do is to do pattern matching with the input, and then write both the base case and the resursive case,

For the last function , I followed the algorithm in the Python code hint given by the TAs. I find it really helpful since it would otherwise take me a massive longer time to think about how to implement this. Basically the idea is to make use of the computed_fixed_point function written earlier and pass in a funciton called get_reachable_symbols as the function parameter so that it keeps calling the get_reachable_symbols function until the output doesn't change. What get_reachable_symbols does is it takes in a list of rules and a list of reachable symbols, and it checks if the left hand side of the rules is in the reachable symbole list. If it is in there, we include the symbols in the right hand side in the reacheble

symbol list as well. If not, we do nothing. Here we are basically performing a search. Of course this part is also implemented recursively: check the first element in the list first and recursively call the function.

After that, we get a list of (final) reachable symbols and we can filter the rules based on if the left hand side is in it. The result are the rules we want.

Some problems I met:

I spent a lot of time trying to figure out type related problems. Thank you TAs for answering my questions on Piazza and helping me out in office hours. I especially find the TA's post pinned on Piazza top most helpful.

I had some problems when translating the python hint codes to Ocaml since they are very different languages and have entirely diffrent data structures. I find writing small helpt functions really helpful.

Specificly, I also had this problem of when I called get_reachable_symbols it keeps deleting elements in the first parameter list when recursively calls itself. It is intended to do that for implementaiton reason(resursion) but that's not good for our purpose of using fixed_point to call its output again since it eventually deletes all the rules. I had to make a helper function outisde of the get_reachable_symbols and keep the original list in this funciton and call the actual function inside this helper funciton. And after that I will pass the helper function as the parameter to the findfixedpoint funciton. This solves the problem.