**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.