```
open List

let rec subset a b =
  if (length a) = 0 then
    true
  else
    if (mem (hd a) b) then
      subset (tl a) b
    else
      false

let equal_sets a b = (subset a b) && (subset b a)

let set_union a b = append a (filter (fun x -> not (mem x a)) b)

let set_intersection a b = filter (fun x -> (mem x a) && (mem x b)) (set_union a b)

let set_diff a b = filter (fun x -> not (mem x b)) a

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

(* Type declaration for filter_reachable *)

type ('n, 't) symbol = N of 'n | T of 't

(* Helper Function for filter_reachable *)
let rec updateAccessibleNonT accessibleNonT rules =
  if (length rules) = 0 then
    accessibleNonT
  else
    let (ruleStartNonTRaw, newSymbols) = (hd rules) in
    let ruleStartNonT = (N ruleStartNonTRaw) in
    if (mem ruleStartNonT accessibleNonT) then
      let newNonT = filter (fun x -> match x with | N a -> true | _ -> false)
newSymbols in
      let newAccessibleNonT = set_union accessibleNonT newNonT in
      updateAccessibleNonT newAccessibleNonT (tl rules)
    else
      updateAccessibleNonT accessibleNonT (tl rules)

let filter_reachable g =
  let (startStateRaw, rules) = g in
  let startStateNonTList = [N startStateRaw] in
  let fixed_point_f x = (updateAccessibleNonT x rules) in
  let accessibleNonT = (computed_fixed_point equal_sets fixed_point_f
startStateNonTList) in

  (* Now convert accessible nonterminal symbols into grammar with reachable rules *)
  let filterFunc rule =
    let (startNonTRaw, _) = rule in
    let startNonT = (N startNonTRaw) in
    mem startNonT accessibleNonT
    in
  (startStateRaw, filter filterFunc rules)
```