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

let convert_grammar gram1 =
  let check_element e = function (nonterm, _) -> e = nonterm in
  let rec group_rules returnVal ruleList =
    match ruleList with
    | [] -> returnVal
    | (left, right)::t ->
       (let add_element = function (nonterm, rules) -> if left = nonterm
                                                        then (nonterm, rules@[right])
                                                        else (nonterm, rules) in
        if List.exists (check_element left) returnVal
        then group_rules (List.map add_element returnVal) t
        else group_rules ((left, [right])::returnVal) t) in
  let production_function ruleList x =
    let grouped_rules = List.rev (group_rules [] ruleList) in
    if List.exists (check_element x) grouped_rules
    then snd (List.find (check_element x) grouped_rules)
    else [[]] in
  match gram1 with (s, a) -> (s, production_function a)

let rec first_nonterm expr =
  match expr with
  | [] -> None
  | (T _)::t -> first_nonterm t
  | (N nonterm)::_ -> Some nonterm

let check_terminal expr =
  match first_nonterm expr with
  | Some _ -> false
  | None -> true

let apply_nonterm expr nonterm replacement =
  let rec apply_nonterm_helper return_expr expr nonterm replacement =
    match expr with
    | [] -> return_expr
    | (T symb)::t -> apply_nonterm_helper ((T symb)::return_expr) t nonterm replacement
    | (N symb)::t -> if symb = nonterm
                     then List.rev_append t (List.rev_append replacement return_expr)
                     else apply_nonterm_helper ((N symb)::return_expr) t nonterm replacement in
  List.rev (apply_nonterm_helper [] expr nonterm replacement)

let next_production_rules expr prod =
  match first_nonterm expr with
  | Some nonterm -> prod nonterm
  | None -> []

let rec prefix_match frag expr =
  match expr with
  | (T symb)::t_expr -> (match frag with
                         | h::t_frag -> if h = symb then prefix_match t_frag t_expr else false
                         | [] -> false)
  | (N _)::_ -> true
  | [] -> true

let generate_derivations eval prod deriv =
  let construct_deriv deriv nonterm replacement = (nonterm, replacement)::deriv in
  match first_nonterm eval with
  | Some nonterm -> (match next_production_rules eval prod with
                   | [] -> List.map (construct_deriv deriv nonterm) [[]]
                   | rules -> List.map (construct_deriv deriv nonterm) rules)
  | None -> []

let filter_derivations frag prev derivs =
  let evaluate_prefix_match frag deriv =
    match deriv with
    | (eval, (_, _)::_) -> prefix_match frag eval
    | _ -> false in
  let calculate_expr prev deriv =
    match deriv with
    | (nonterm, replacement)::_ -> (apply_nonterm prev nonterm replacement, deriv)
    | [] -> (prev, deriv) in
  List.filter (evaluate_prefix_match frag) (List.map (calculate_expr prev) derivs)

let rec generate_suffix eval frag =
  match eval with
  | (T symb)::t_eval -> (match frag with
                       | [] -> []
                       | h::t_frag -> if h = symb
                                      then generate_suffix t_eval t_frag
                                      else frag)
  | (N _)::_ -> frag
  | [] -> frag

let rec generate_valid_derivation start prod accept frag derivs =
  let self = generate_valid_derivation start prod accept frag in
  match derivs with
  | [] -> None
  | (eval, h)::t -> if not (check_terminal eval)
                    then self ((filter_derivations frag eval (generate_derivations eval prod h))@t)
                    else (match accept (List.rev h) (generate_suffix eval frag) with
                          | Some (deriv, suf) -> Some (deriv, suf)
                          | None -> self t)

let parse_prefix gram =
  let matcher start production accept frag =
    generate_valid_derivation start production accept frag [(start, [])] in
  match gram with (s, p) -> matcher [N s] p
```