```
(* ============================= type definitions ============================= *)

type ('nonterminal, 'terminal) parse_tree =
  | Node of 'nonterminal * ('nonterminal, 'terminal) parse_tree list
  | Leaf of 'terminal;;

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

(* =========================== convert_grammar gram1 ========================== *)

       (* <filtering> gets list of rules with matching nonterminal symbol *)
let filtering gram1 nt_symbol =
List.filter (fun rule -> Pervasives.compare (Pervasives.fst rule) nt_symbol = 0) (Pervasives.snd gram1);;

       (* <production_function> separates rule list returned by <filtering> and separates
        * the lhs and rhs into two lists, returning the list of rhs (alternative list) *)
let production_function gram1 nt_symbol =
Pervasives.snd (List.split (filtering gram1 nt_symbol));;

       (* <convert_grammar> converts a hw1 style grammar into a hw2 style grammar *)
let convert_grammar gram1 = match gram1 with
| (start_symbol, rules) -> (start_symbol, (fun nt_symbol -> production_function gram1 nt_symbol));;

(* ========================== parse_tree_leaves tree ========================== *)

       (* <build_leaf_pile> performs something like a depth first search to get leaves *)
let rec build_leaf_pile tree leaf_pile = match tree with
| [] -> leaf_pile
| h::t ->
       (match h with
       | Node (_, branches) -> build_leaf_pile t (build_leaf_pile branches leaf_pile)
       | Leaf fallen_leaf -> build_leaf_pile t (List.cons fallen_leaf leaf_pile));;

       (* <parse_tree_leaves gets a list of leaves with ordering from left to right *)
let parse_tree_leaves tree =
List.rev (build_leaf_pile (List.cons tree []) []);;

(* ============================ make_matcher gram ============================= *)

       (* <do_matching> returns a option type depending on if a fragment is accepted;
        * also iterates through alternative lists for a nonterminal symbol
        * <process_fragment> process the fragments with a right hand side; mutually
        * recurses with <do_matching> to expand rule expressions for matching and
        * matches are checked with acceptor *)
let rec do_matching start_symbol prod_func alt_list accept frag subtree =
let rec process_fragment prod_func sym_list accept frag branch = match sym_list with
| [] -> accept frag branch
| sym_h::sym_t ->
        (match frag with
        | [] -> None
        | frag_h::frag_t ->
                (match sym_h with
                | N sym -> do_matching sym prod_func (prod_func sym) (process_fragment prod_func sym_t accept) frag branch
                | T sym ->
                        if Pervasives.compare sym frag_h = 0 then
                                process_fragment prod_func sym_t accept frag_t branch
                        else
                                None))
in
match alt_list with
| [] -> None
| alt_h::alt_t ->
        (match process_fragment prod_func alt_h accept frag (List.append subtree [(start_symbol, alt_h)]) with
        | None -> do_matching start_symbol prod_func alt_t accept frag subtree
        | Some acceptor_return -> Some acceptor_return);;

       (* <the_matcher> is an auxiliary function to make the currying clearer
        * <matcher_acceptor> modifies the acceptor passed to <do_matching> to only get
        * if a fragment is valid or not, without a derivation *)
let the_matcher gram accept frag =
let matcher_acceptor frag derivation = (accept frag)
in
do_matching (Pervasives.fst gram) (Pervasives.snd gram) (Pervasives.snd gram (Pervasives.fst gram)) matcher_acceptor frag [];;

       (* <make_matcher> returns a matcher for a given grammar *)
let make_matcher gram = the_matcher gram;;

(* ============================ make_parser gram ============================== *)

       (* <parse_this_path> returns the desired parse tree for a given derivation
        * <draw_branch> does the actual processing by iterating through the path and
        * linking the nodes together in the tree *)
let parse_this_path derivation =
let rec draw_branch parent children rest_of_tree branch = match children with
| [] -> (Node (parent, branch))
| child_h::child_t ->
        (match child_h with
        | N sym ->
                (match rest_of_tree with
                | [] -> (Node (parent, branch)) (* if my logic is correct, this should never happen *)
                | rot_h::rot_t ->
                        if Pervasives.compare sym (Pervasives.fst rot_h) = 0 then
                                draw_branch parent child_t rest_of_tree (List.append branch [(draw_branch (Pervasives.fst rot_h) (Pervasives.snd rot_h)
rot_t [])])
                        else
                                draw_branch parent children rot_t branch)
        | T sym ->
              draw_branch parent child_t rest_of_tree (List.append branch [Leaf sym]))
in
match derivation with
| None -> None
| Some derivation ->
        (match derivation with
        | [] -> None
        | h::t ->
              Some (draw_branch (Pervasives.fst h) (Pervasives.snd h) t []));;

        (* <the_parser> is an auxiliary function to make the currying clearer
         * <parser_accept> modifies the acceptor passed to <do_matching> to only get the
         * derivation of a fragment *)
let rec the_parser gram frag =
let parser_acceptor frag derivation = match frag with
| [] -> Some derivation
| _ -> None
in
parse_this_path
(do_matching (Pervasives.fst gram) (Pervasives.snd gram) (Pervasives.snd gram (Pervasives.fst gram)) parser_acceptor frag []);;

        (* <make_parser> returns a parser for a given grammar *)
let make_parser gram = the_parser gram;;
```