```
open List

(* CONVERT_GRAMMAR *)

(* Helper function for convert_grammar *)

let rec join_rules rules inSym = match rules with
        | [] -> []
        | (lhs, rhs)::rest ->
                if inSym = lhs then
                        rhs::(join_rules rest inSym)
                else
                        join_rules rest inSym

let convert_grammar gram1 =
        let (startSym, rules) = gram1 in
        (startSym, (fun x -> join_rules rules x))


(* PARSE_PREFIX *)

(* Symbol type declaration *)
type ('a, 'b) symbol =
        | N of 'a
        | T of 'b

(*
Recursively runs through a state (list of symbols), checking that it the given
fragment.
If it encounters a nonterminal symbol, it calls expand_NT, which handles moving down
to the next
level of parse tree, and deploys new check_state instances with the state transformed
by each possible rule.
*)
let rec check_state curDeriv curState rules accept frag =
        match curState with
        | [] -> accept curDeriv frag
        | firstSym::restOfState ->
                (
                match firstSym with
                | (T rawSym) ->
                        (* First, check that frag is nonempty *)
                        (
                        match frag with
                        | [] -> None
                        | firstFrag::restOfFrag ->
                                (* Check that first char of state matches first char
of frag *)
                                if (rawSym = firstFrag) then
                                        check_state curDeriv restOfState rules accept
restOfFrag
                                else
                                        None
                        )
                | (N rawSym) ->
                        (* Now we must find all the possible rules from rawSym, and
then call expand_NT, which will run check_state with the new states formed by these
possible rules *)
                        let possibleDerivs = (rules rawSym) in
                        expand_NT curDeriv possibleDerivs rawSym restOfState rules
accept frag
                )
```

```
(*
Recursively deploys check_state instances with each possible rule (from the left-most
nonterminal) applied.
*)
and expand_NT curDerivList possibleDerivs nonT restOfState rules accept frag =
        match possibleDerivs with
        | [] -> None (* This occurs only when all derivations from nonT have been
exhausted, and all of them returned None *)
        | deriv::restOfDerivs ->
                let newDerivList = curDerivList @ [(nonT, deriv)] in
                let newState = deriv @ restOfState in
                (* check this new state. If it returns None, we'll try the next deriv
in the list *)
                (
                match (check_state newDerivList newState rules accept frag) with
                | None -> expand_NT curDerivList restOfDerivs nonT restOfState rules
accept frag
                | result -> result (* If we get a result != None, then we want to
pass that result up the recursion levels all the way to the final result of
parse_prefix *)
                )

(* The overall function runs check_state with the basic nonterminal starting state *)
let parse_prefix gram = match gram with (startNonT, rules) ->
        (fun accept frag -> check_state [] [N startNonT] rules accept frag)
```