```
open List;;
open Pervasives;;

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


let rec find_non_terminal_rules r n =
    match r with
    | [] -> []
    | (lhs, rhs)::tail ->
        if (lhs = n) then rhs::(find_non_terminal_rules tail n)
        else find_non_terminal_rules tail n

let convert_grammar gram1 =
    match gram1 with
    | (n, rhs) -> (n, find_non_terminal_rules rhs)
;;


let get_rule_list n = function | (_,rhs) -> rhs n
let rec match_first_mem gram rules accept deriv frag =
    if length frag < length rules then None else (
    match rules with
    | [] -> (accept deriv frag)
    | (T term_rule)::other_rules->
        (match frag with
        | [] -> None
        | f1::f -> if (f1 = term_rule) then match_first_mem gram other_rules accept deriv
f else None )
    | (N rule)::other_rules -> move_horizontal gram rule other_rules (get_rule_list rule
gram) accept deriv frag
    )
and move_horizontal gram curr_nonterm rules alt accept deriv frag =
    match alt with
    | [] -> None
    | h::t ->
            match match_first_mem gram (h @ rules) accept (deriv @ [(curr_nonterm, h)])
frag with
                | None -> move_horizontal gram curr_nonterm rules t accept deriv frag
                | good -> good

let find_root g =
    match g with
    | (lhs, _) -> lhs

let make_matcher gram accept frag =
    move_horizontal gram (find_root gram) [] (get_rule_list (find_root gram) gram) accept
[] frag

let parse_prefix gram =
    make_matcher gram


let accept_all derivation string = Some (derivation, string)
let accept_empty_suffix derivation = function
   | [] -> Some (derivation, [])
   | _ -> None

(* An example grammar for a small subset of Awk.
   This grammar is not the same as Homework 1; it is
   instead the same as the grammar under
   "Theoretical background" above.  *)

type awksub_nonterminals =
  | Expr | Term | Lvalue | Incrop | Binop | Num

let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
     [[N Num];
      [N Lvalue];
      [N Incrop; N Lvalue];
      [N Lvalue; N Incrop];
      [T"("; N Expr; T")"]]
     | Lvalue ->
     [[T"$"; N Expr]]
     | Incrop ->
     [[T"++"];
      [T"--"]]
     | Binop ->
     [[T"+"];
      [T"-"]]
     | Num ->
     [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
      [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])

let test0 =
  ((parse_prefix awkish_grammar accept_all ["ouch"]) = None)

let test1 =
  ((parse_prefix awkish_grammar accept_all ["9"])
   = Some ([(Expr, [N Term]); (Term, [N Num]); (Num, [T "9"])], []))

let test2 =
  ((parse_prefix awkish_grammar accept_all ["9"; "+"; "$"; "1"; "+"])
   = Some
       ([(Expr, [N Term; N Binop; N Expr]); (Term, [N Num]); (Num, [T "9"]);
     (Binop, [T "+"]); (Expr, [N Term]); (Term, [N Lvalue]);
     (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Num]);
     (Num, [T "1"])],
    ["+"]))

let test3 =
  ((parse_prefix awkish_grammar accept_empty_suffix ["9"; "+"; "$"; "1"; "+"])
   = None)

(* This one might take a bit longer.... *)
let test4 =
 ((parse_prefix awkish_grammar accept_all
     ["("; "$"; "8"; ")"; "-"; "$"; "++"; "$"; "--"; "$"; "9"; "+";
      "("; "$"; "++"; "$"; "2"; "+"; "("; "8"; ")"; "-"; "9"; ")";
      "-"; "("; "$"; "$"; "$"; "$"; "$"; "++"; "$"; "$"; "5"; "++";
      "++"; "--"; ")"; "-"; "++"; "$"; "$"; "("; "$"; "8"; "++"; ")";
      "++"; "+"; "0"])
  = Some
     ([(Expr, [N Term; N Binop; N Expr]); (Term, [T "("; N Expr; T ")"]);
       (Expr, [N Term]); (Term, [N Lvalue]); (Lvalue, [T "$"; N Expr]);
       (Expr, [N Term]); (Term, [N Num]); (Num, [T "8"]); (Binop, [T "-"]);
       (Expr, [N Term; N Binop; N Expr]); (Term, [N Lvalue]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term; N Binop; N Expr]);
       (Term, [N Incrop; N Lvalue]); (Incrop, [T "++"]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term; N Binop; N Expr]);
       (Term, [N Incrop; N Lvalue]); (Incrop, [T "--"]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term; N Binop; N Expr]);
       (Term, [N Num]); (Num, [T "9"]); (Binop, [T "+"]); (Expr, [N Term]);
       (Term, [T "("; N Expr; T ")"]); (Expr, [N Term; N Binop; N Expr]);
       (Term, [N Lvalue]); (Lvalue, [T "$"; N Expr]);
       (Expr, [N Term; N Binop; N Expr]); (Term, [N Incrop; N Lvalue]);
       (Incrop, [T "++"]); (Lvalue, [T "$"; N Expr]); (Expr, [N Term]);
       (Term, [N Num]); (Num, [T "2"]); (Binop, [T "+"]); (Expr, [N Term]);
       (Term, [T "("; N Expr; T ")"]); (Expr, [N Term]); (Term, [N Num]);
       (Num, [T "8"]); (Binop, [T "-"]); (Expr, [N Term]); (Term, [N Num]);
       (Num, [T "9"]); (Binop, [T "-"]); (Expr, [N Term]);
       (Term, [T "("; N Expr; T ")"]); (Expr, [N Term]); (Term, [N Lvalue]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Lvalue]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Lvalue]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Lvalue; N Incrop]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Lvalue; N Incrop]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Incrop; N Lvalue]);
       (Incrop, [T "++"]); (Lvalue, [T "$"; N Expr]); (Expr, [N Term]);
       (Term, [N Lvalue; N Incrop]); (Lvalue, [T "$"; N Expr]); (Expr, [N Term]);
       (Term, [N Num]); (Num, [T "5"]); (Incrop, [T "++"]); (Incrop, [T "++"]);
       (Incrop, [T "--"]); (Binop, [T "-"]); (Expr, [N Term]);
       (Term, [N Incrop; N Lvalue]); (Incrop, [T "++"]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]); (Term, [N Lvalue; N Incrop]);
       (Lvalue, [T "$"; N Expr]); (Expr, [N Term]);
       (Term, [T "("; N Expr; T ")"]); (Expr, [N Term]);
       (Term, [N Lvalue; N Incrop]); (Lvalue, [T "$"; N Expr]); (Expr, [N Term]);
       (Term, [N Num]); (Num, [T "8"]); (Incrop, [T "++"]); (Incrop, [T "++"]);
       (Binop, [T "+"]); (Expr, [N Term]); (Term, [N Num]); (Num, [T "0"])],
      []))

let rec contains_lvalue = function
  | [] -> false
  | (Lvalue,_)::_ -> true
  | _::rules -> contains_lvalue rules

let accept_only_non_lvalues rules frag =
  if contains_lvalue rules
  then None
  else Some (rules, frag)

let test5 =
  ((parse_prefix awkish_grammar accept_only_non_lvalues
      ["3"; "-"; "4"; "+"; "$"; "5"; "-"; "6"])
   = Some
      ([(Expr, [N Term; N Binop; N Expr]); (Term, [N Num]); (Num, [T "3"]);
    (Binop, [T "-"]); (Expr, [N Term]); (Term, [N Num]); (Num, [T "4"])],
       ["+"; "$"; "5"; "-"; "6"]))

let accept_comment deriv suff =
    let rec find_last_entry = function
    | [] -> []
    | h::[] -> [h]
    | h::t -> find_last_entry t
    in
    match suff with
    | [] -> None
    | h::t -> if h = "/*" then
        (match find_last_entry t with
            | [] -> None
            | l::_ -> if l = "*/" then Some(deriv, suff)
        else None)
    else None

let accept_all derivation string = Some (derivation, string)


type some_nonterminals =
    | Function | Def | Paramlist | Param | Name | Char | Special


let mygrammar_1 =
    (Function, function
        | Function -> [[N Name; N Paramlist; T ";" ; N Def];
                        [N Name; N Paramlist; T ";"]]
        | Def -> [[N Function]]
        | Paramlist -> [[T "("; N Param; T ")"]]
        | Param -> [[N Name; T ","; N Param];
                    [N Name]]
        | Name -> [[N Char; N Name];
                    [N Char]]
        | Char -> [[T "a"];[T "b"];[T "c"];[T "d"];[T "e"];[T "f"];
                    [T "g"];[T "h"];[T "i"];[T "j"];[T "k"];[T "l"];
                    [T "m"];[T "n"];[T "o"];[T "p"];[T "q"];[T "r"];
                    [T "s"];[T "t"];[T "u"];[T "v"];[T "w"];[T "x"];
                    [T "y"]; [T "z"]]
    )


(* intent of this is to check that it backtracks correctly*)
(* it follows the first definition of function first before
    reaching the end and finding there's nothing that conforms
    to definition of def, thus it must backtrack to original definition
    of function and use the second definition, undoing all the parsing
    it did for the parameter list*)
let test_1 = parse_prefix mygrammar_1 accept_comment
    ["t";"e";"s";"t";"(";"a";"r";"g";"o";",";"a";"r";
    "g";"a";"r";"g";"t";")";";";"/*";
    "b";"a";"c";"k";"t";"r";"a";"c";"k";
    "m";"o";"r";"e";"*/"]
    = Some
    ([(Function, [N Name; N Paramlist; T ";"]);
    (Name, [N Char; N Name]); (Char, [T "t"]);
    (Name, [N Char; N Name]); (Char, [T "e"]);
    (Name, [N Char; N Name]); (Char, [T "s"]);
    (Name, [N Char]); (Char, [T "t"]);
    (Paramlist, [T "("; N Param; T ")"]);
    (Param, [N Name; T ","; N Param]);
    (Name, [N Char; N Name]); (Char, [T "a"]);
    (Name, [N Char; N Name]); (Char, [T "r"]);
    (Name, [N Char; N Name]); (Char, [T "g"]);
    (Name, [N Char]); (Char, [T "o"]);
    (Param, [N Name]);
    (Name, [N Char; N Name]); (Char, [T "a"]);
    (Name, [N Char; N Name]); (Char, [T "r"]);
    (Name, [N Char; N Name]); (Char, [T "g"]);
    (Name, [N Char; N Name]); (Char, [T "a"]);
    (Name, [N Char; N Name]); (Char, [T "r"]);
    (Name, [N Char; N Name]); (Char, [T "g"]);
    (Name, [N Char]); (Char, [T "t"])],
    ["/*"; "b"; "a"; "c"; "k";
    "t"; "r"; "a"; "c"; "k";
    "m"; "o"; "r"; "e";
   "*/"]);;


(*can it handle loops in definitions*)
(*def is defined to be a function *)
let test_2 = parse_prefix mygrammar_1 accept_all
    ["t";"(";"e";")"; ";";
    "s";"(";"t";",";"t";")";";";
    "w";"(";"o";")";";";
    "/*"; "loops";"*/"]
    = Some
   ([(Function, [N Name; N Paramlist; T ";"; N Def]);
    (Name, [N Char]); (Char, [T "t"]);
    (Paramlist, [T "("; N Param; T ")"]);
    (Param, [N Name]); (Name, [N Char]); (Char, [T "e"]);
    (Def, [N Function]);
    (Function, [N Name; N Paramlist; T ";"; N Def]);
    (Name, [N Char]); (Char, [T "s"]);
    (Paramlist, [T "("; N Param; T ")"]);
    (Param, [N Name; T ","; N Param]);
    (Name, [N Char]); (Char, [T "t"]);
    (Param, [N Name]); (Name, [N Char]); (Char, [T "t"]);
    (Def, [N Function]);
    (Function, [N Name; N Paramlist; T ";"]);
    (Name, [N Char]); (Char, [T "w"]);
    (Paramlist, [T "("; N Param; T ")"]);
    (Param, [N Name]); (Name, [N Char]); (Char, [T "o"])],
    ["/*"; "loops"; "*/"]);;
```

```
# #use "hw2.ml";;
type ('nonterminal, 'terminal) symbol = N of 'nonterminal | T of 'terminal
val find_non_terminal_rules : ('a * 'b) list -> 'a -> 'b list = <fun>
val convert_grammar : 'a * ('b * 'c) list -> 'a * ('b -> 'c list) = <fun>
val get_rule_list : 'a -> 'b * ('a -> 'c) -> 'c = <fun>
val match_first_mem :
  'a * ('b -> ('b, 'c) symbol list list) ->
  ('b, 'c) symbol list ->
  (('b * ('b, 'c) symbol list) list -> 'c list -> 'd option) ->
  ('b * ('b, 'c) symbol list) list -> 'c list -> 'd option = <fun>
val move_horizontal :
  'a * ('b -> ('b, 'c) symbol list list) ->
  'b ->
  ('b, 'c) symbol list ->
  ('b, 'c) symbol list list ->
  (('b * ('b, 'c) symbol list) list -> 'c list -> 'd option) ->
  ('b * ('b, 'c) symbol list) list -> 'c list -> 'd option = <fun>
val find_root : 'a * 'b -> 'a = <fun>
val make_matcher :
  'a * ('a -> ('a, 'b) symbol list list) ->
  (('a * ('a, 'b) symbol list) list -> 'b list -> 'c option) ->
  'b list -> 'c option = <fun>
val parse_prefix :
  'a * ('a -> ('a, 'b) symbol list list) ->
  (('a * ('a, 'b) symbol list) list -> 'b list -> 'c option) ->
  'b list -> 'c option = <fun>
```

```
# #use "hw2sample.ml";;
val accept_all : 'a -> 'b -> ('a * 'b) option = <fun>
val accept_empty_suffix : 'a -> 'b list -> ('a * 'c list) option = <fun>
type awksub_nonterminals = Expr | Term | Lvalue | Incrop | Binop | Num
val awkish_grammar :
  awksub_nonterminals *
  (awksub_nonterminals -> (awksub_nonterminals, string) symbol list list) =
  (Expr, <fun>)
val test0 : bool = true
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
val contains_lvalue : (awksub_nonterminals * 'a) list -> bool = <fun>
val accept_only_non_lvalues :
  (awksub_nonterminals * 'a) list ->
  'b -> ((awksub_nonterminals * 'a) list * 'b) option = <fun>
val test5 : bool = true
# #use "hw2test.ml";;
val accept_comment : 'a -> string list -> ('a * string list) option = <fun>
val accept_all : 'a -> 'b -> ('a * 'b) option = <fun>
type some_nonterminals =
    Function
  | Def
  | Paramlist
  | Param
  | Name
  | Char
  | Special
File "hw2test.ml", line 24, characters 15-680:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Special
val mygrammar_1 :
  some_nonterminals *
  (some_nonterminals -> (some_nonterminals, string) symbol list list) =
  (Function, <fun>)
val test_1 : bool = true
val test_2 : bool = true
```