HW2.ML

```
open List;;

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

(*convert_grammar gram1 that returns a Homework 2-style grammar
hw1 grammer:
A pair, consisting of a start symbol and a list of rules.
The start symbol is a nonterminal value.
hw2 grammer:
A pair, consisting of a start symbol and a production function.
The start symbol is a nonterminal value.
production function:
A function whose argument is a nonterminal value. It returns a
grammar's alternative list for that nonterminal.

*)

(*production takes in nonterminal and rule and returns the alternative
list*)

let rec production rules nt = match rules with
        |[]-> []
        |hd::tl-> if fst hd = nt
                                then (snd hd)::production tl nt
                        else production tl nt;;


let convert_grammar gram1 = match gram1 with
        (a,b)-> (a,production b) ;;

(*passed test*)

(*parse_tree_leaves tree that traverses the parse tree tree left
to right and yields a list of the leaves encountered.

parse tree
a data structure representing a parse tree in the usual way.
It has the following OCaml type:

If you traverse a parse tree in preorder left to right,
```

```
the leaves you encounter contain the same terminal symbols as the
parsed fragment, and each internal node of the parse tree corresponds
to a rule in the grammar, traversed in a leftmost derivation order.*)

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

(*preorder lef to right*)

(*helper1 takes in a list of trees and return a list of leaves*)
let rec helper1 trees = match trees with
          | [] -> []
          | hd::tl -> match hd with
                                  | Leaf a -> a:: (helper1 tl)
                                  | Node (b,c)-> helper1 c @(helper1 tl);;

let parse_tree_leaves tree = helper1 [tree];;
(*passed test 5*)

(*Write a function make_matcher gram that returns a matcher for the
 grammar gram. When applied to an acceptor accept and a fragment frag,
 the matcher must try the grammar rules in order and return the result
 of calling accept on the suffix corresponding to the first acceptable
 matching prefix of frag; this is not necessarily the shortest or the
 longest acceptable match. A match is considered to be acceptable if
accept succeeds when given the suffix fragment that immediately follows
the matching prefix. When this happens, the matcher returns whatever
the acceptor returned. If no acceptable match is found, the matcher
returns None.*)

(*acceptor
a function whose argument is a fragment frag. If the fragment is not
acceptable, it returns None; otherwise it returns Some x for some value
x.*)


(*matcher
a curried function with two arguments: an acceptor accept and a fragment
frag. A matcher matches a prefix p of frag such that accept (when passed
the corresponding suffix) accepts the corresponding suffix (i.e., the
suffix of frag that remains after p is removed). If there is such a
match, the matcher returns whatever accept returns; otherwise it returns
None.
```

```
*)

(*rules actually mean right hand sides*)

let rec topsearch grammer rules accept frag = match rules with
| [] -> None
| current_rule::other_rules ->
                    match botomsearch grammer current_rule accept frag with
                    | None -> topsearch grammer other_rules accept frag
                    | a -> a

(*frag: hd::tl
rule: nonter/ter::tl1/2*)

and botomsearch  grammer rule accept frag = match frag with
| [] -> if rule = [] then accept [] else None
| hd::tl -> match rule with
          | [] -> accept frag
          | N nonter:: tl1 ->
                    topsearch grammer ((snd grammer) nonter)
                    (botomsearch grammer tl1 accept) frag
          | T ter:: tl2->
                    if ter = hd
                              then botomsearch grammer tl2 accept tl
                    else None;;

let make_matcher gram accept frag = topsearch gram ((snd gram) (fst gram)) accept frag;;

(*Write a function make_parser gram that returns a parser for the
grammar gram. When applied to a fragment frag, the parser returns
an optional parse tree. If frag cannot be parsed entirely (that is,
from beginning to end), the parser returns None. Otherwise, it return
some tree where tree is the parse tree corresponding to the input
fragment. Your parser should try grammar rules in the same order as
make_matcher.*)

(*rhs is a list of the right hand side of rules
 rhstopsearch returns a pair of option and final rhs*)
(*rules is the *)

(*rules actually means right hand sides*)

let p_accept_empty_suffix = function
   | [] -> Some []
```

```
  | _  -> None ;;

let rec rhstopsearch grammer rules accept frag = match rules with
| [] -> None
| current_rule::other_rules ->
                match rhsbotomsearch grammer current_rule accept frag with
                | None -> rhstopsearch grammer other_rules accept frag
                | Some a -> Some (current_rule::a)

(*frag: hd::tl
rule: nonter/ter::tl1/2*)

and rhsbotomsearch  grammer rule accept frag = match frag with
| [] -> if rule = [] then accept [] else None
| hd::tl -> match rule with
        | [] -> accept frag
        | N nonter:: tl1 ->
                rhstopsearch grammer ((snd grammer) nonter)
                (rhsbotomsearch grammer tl1 accept) frag
        | T ter:: tl2->
                if ter = hd
                        then rhsbotomsearch grammer tl2 accept tl
                else None;;




let make_rhs gram frag =  rhstopsearch gram ((snd gram) (fst gram))
        p_accept_empty_suffix frag ;;

(*takes in a list of nodes/leafs and a rhs and full list of them
according to the rhs. not recursive, only does one layer*)
(*RHS IS ONE PIECE OF RHS*)
let rec rhs2children children rhs = match rhs with
        | [] -> children
        |h::t -> match h with
                |T ter->rhs2children (children@[Leaf ter]) t
                |N nonter-> rhs2children (children@[Node (nonter,[])]) t;;

(*TODO CHECK RHS2CHILDREN TYPE*)

(*returns new children and new rhs*)

let rec helper chilren rhs = match chilren with
```

```
        |[] -> ([],rhs)
        |h::t -> match  constructing_tree rhs h with
                |([],subtree)->([subtree],[]) (*?*)
                |(newrhs,subtree)->match helper t newrhs with
                        |(nchildren,nrhs)->(subtree::nchildren,nrhs)


(*
        match the fisrt one with
        if [] (nrhs, [subtree])
        anything match helper nrhs rest with rhs subs->(rhs,subtree:subs)
*)
(*return tuple of newrhs and newroot*)
and constructing_tree rhs_traced temp_root = match rhs_traced with
        |[]->(rhs_traced,temp_root)
        |hrhs::trhs -> match temp_root with
                |Node (n,treelist) ->
                        let children = rhs2children treelist hrhs in (*?*)

                        let newchildren,newrhs = helper children trhs in
                        let newroot = Node (n,newchildren) in
                        (newrhs,newroot)

                        (*  node of label and children
                        and return tuple of newrhs and newroot*)
                |Leaf l -> (rhs_traced,temp_root);;



let make_parser gram frag = match frag with
        |[] -> None
        |_ -> match make_rhs gram frag with
                |None->None
                |Some rhs->match constructing_tree rhs (Node ((fst gram),[] )) with
                                |([],newroot)->Some newroot
                                |_ -> None;;
```

HW2.REPORT

First of all, I want to say that my algorithm for make_parser roughly follows the hint code provided by the TAs. Again, I find the hint code extremly helpful.

I wrote make_parser on top of make_matcher, although I didn't explicitely use the function make_matcher inside my function make_parser. My make_matcher function uses two mutually recursive functions topsearch and botomsearch. Topsearch takes cares of picking which specific rhs and botomsearch takes care of testing every element in the rhs.

In implementing my make_parser function, like the hint code implies, I first made a make_rhs function, which returns the rhs in order when we traverse the fragments. Here is when the make_matcher comes in handy. The make_rhs function is totally based on make_matcher: it also uses a pair of mutually recursive function: rhstopsearch and rhsbotomsearch, which are very much like topsearch and botomsearch,but this time instead of returning the ultimate result of the acceptor they return list of rhs recursively. Of course, we also need to set the acceptor as accepting only empty.

After getting the rhs list, we need to be able to build the tree based on it. First I made a rhs2children function that shallowly makes a two level tree based on single rhs. After that I made a pair of recursive funtions constructing_the_tree and helper which actually builds the tree up. Constructing_the_tree builds a shallow two level tree, and then helper is called on the branches to explore deeper.

As to why I developed make_parser on top of make_matcher this way is that I followed the hint code and it's very intuitively obvious to build rhstopsearch and rhsbotomsearch on top of topsearch and botomsearch.

In term of weakness, if the frag is wrong (say there is a terminate that has never been defined), instead of returning None my make_parser will get into infinite recursion and eventually have a overflow stack.