Assignment 3: Natural Deduction

This assignment is a combinatorial exercise in modelling Natural Deduction Proof-trees and various functions on these trees. It is both an exercise in OCaml programming with trees and composing functions, as well as a way to learn about the Natural Deduction Proof system in a hands-on fashion.

First, the rules of Natural Deduction: (Read G as "Gamma" a set of propositions, p,q,r are propositions, and read "~" as "not").

```
(Ass) ---------- (provided p \in G)             (T-I)   ------------------
         G |- p                                          G |-  T


                G,p |- q                        G |- p->q        G |- p
(->- I)  --------------------           (->-E) -----------------------------
             G |- p->q                                    G |- q


             G |- p      G|- q                        G |- p /\ q                    G |- p /\ q
(/\-I)     --------------------------      (/\-E_l)    ---------------------   (/\-E_r)    ---------------------
                 G |- p /\ q                             G |- p                         G |- q


             G |- p                        G |- q                      G |- p \/ q      G, p |-  r       G, q |- r
(\/-I_l)    ---------------------    (\/-I_r)    -------------------       (\/-E) --------------------------------------------------
             G |- p \/ q                   G |- p \/ q                                   G |- r


~p is DEFINED as p -> F
                                           G, ~p |- F                  G |- F
(F-E)   -------------------        (~-Class)   -----------------------      (~-Int)  --------------
              G, F |- p                       G |- p                    G |- p
```

First we need to model propositions, as in Assignment 2, and sets/lists of propositions as in assignment 1

Now let us define

type sequent = prop set * prop   (* representing G |- p *)

Now let us define prooftrees as a data type as follows:

```
type prooftree  = Ass of sequent | tI of sequent | fE of sequent
                       |  ImpI of prooftree * sequent | ImpE of prooftree * prooftree * sequent
                        | AndI of prooftree * prooftree * sequent | AndEleft of prooftree * sequent | AndEright of prooftree * sequent |
                        | OrIleft of prooftree * sequent | OrIright of prooftree * sequent | OrE of prooftree * prooftree * prooftree * sequent
                        NotClass of  prooftree * sequent |  NotIntu of prooftree * sequent ;;
```

1. Define the function **ht**, which returns the height of a prooftree.
2. Define the function **size**, which returns the number of nodes (rules used) in a prooftree
3. Define the function **wfprooftree** that check that a given candidate proof tree is indeed a well-formed proof tree (i.e., the main formula is of the form expected by the rule, the side formulas are consistent with the main formula, and the extra formulas agree as specified in each rule).
4. Write a function **pad**, that given a well-formed proof tree and a set of additional assumptions, creates a new well-formed proof tree with the set of additional assumptions added at each node. (F1)
5. Write a function **pare** that given a well-formed proof tree, returns a well-formed proof tree with minimal assumptions in each sequent. (F2)
6. Write a function **graft** that given a proof tree pi_0 of D |- p and a list of *i* proof trees pi_*i* for G |- q_*i* for each q_*i*   in D, returns a proof tree for G |- p, where the trees for G |- q_*i* have replaced the leaves of the form D' |- q_*i* in a proof tree similar in shape to pi_0. (F3)
7. Write a program **normalise** which removes all occurrences of r-pairs in a given well-formed proof tree (i.e., where an introduction rule is followed only by an elimination rule of the main connective).