```
(* ******************** *)
(* Analyseur Chapitre 2  *)
(*  Graignic Guillaume   *)
(*  Cadoret Olivier      *)
(* ******************** *)

#use "scanner.ml";;
open List;;

(* Vocabulaires non terminals codé en types énumérés *)
type vnonterm = S | Expr | Termb | SuiteExpr | Facteurb | SuiteTermb | Relation | Op ;;

(* Vocabulaires terminals codé en types énumérés *)
type vterm = T_OU | T_ET |
             T_SUP | T_INF | T_EGAL | T_DIFF |
             T_SUPEGAL | T_INFEGAL |
             T_SI | T_SINON | T_ALORS | T_FSI |
             T_IDENT | T_EOF |
             T_PAROUV | T_PARFERM;;

(* v est l'union des vocabulaires terminals et non terminals *)
type v = Non_term of vnonterm | Term of vterm;;

(* Un arbre_concret est soit une liste d'autres arbre_concrets non terminaux (ACNT) soit une
unite_lexicale terminal (ACT) *)
type arbre_concret = ACNT of vnonterm * (arbre_concret list)
                   | ACT of unite_lexicale;;

(* Un arbre_abstrait lui est soit deux arbre_abstrait lié par un Ou ou un Et, soit une condition de
trois autre arbre_abstraits soit deux string séparés par une unite_lexicale (un operateur en
l'occurence)*)
type arbre_abstrait =
    Cond of arbre_abstrait * arbre_abstrait * arbre_abstrait
  | Comp of string * unite_lexicale * string
  | Ou of arbre_abstrait * arbre_abstrait
  | Et of arbre_abstrait * arbre_abstrait;;

(* fonction transformant les UL en Terminaux sauf pour UL_ERR*)
let term_of_ul = function
    UL_IDENT(_) -> T_IDENT
  | UL_OUVR -> T_PAROUV
  | UL_FERM -> T_PARFERM
  | UL_SUP -> T_SUP
  | UL_INF -> T_INF
  | UL_EGAL -> T_EGAL
  | UL_DIFF -> T_DIFF
  | UL_SUPEGAL -> T_SUPEGAL
  | UL_INFEGAL -> T_INFEGAL
  | UL_SI -> T_SI
  | UL_ALORS -> T_ALORS
  | UL_SINON -> T_SINON
  | UL_FSI -> T_FSI
  | UL_ET -> T_ET
  | UL_OU -> T_OU
  | UL_EOF -> T_EOF;;

(* Exception lorsque l'on sort de la grammaire *)
exception Pas_de_derivation of vnonterm * unite_lexicale;;

(* Fonction donnant la dérivation des non terminaux de la grammaire en fonction de l'unité lexicale
suivante *)
let (ma_derivation : vnonterm * unite_lexicale -> v list) = function
    S,_ -> [Non_term Expr;Term T_EOF]
  | Expr,_ -> [Non_term Termb;Non_term SuiteExpr]
  | SuiteExpr,UL_OU -> [Term T_OU;Non_term Expr]
  | SuiteExpr,_ -> []
  | Termb,_ -> [Non_term Facteurb;Non_term SuiteTermb]
  | SuiteTermb,UL_ET -> [Term T_ET;Non_term Termb]
  | SuiteTermb,_ -> []
  | Facteurb,UL_OUVR -> [Term T_PAROUV;Non_term Expr;Term T_PARFERM]
  | Facteurb,UL_SI -> [Term T_SI;Non_term Expr;Term T_ALORS;Non_term Expr;Term T_SINON;Non_term
Expr;Term T_FSI]
```

```
  | Facteurb,_ -> [Non_term Relation]
  | Relation,UL_IDENT(_)  -> [Term T_IDENT;Non_term Op;Term T_IDENT]
  | Relation,ul -> raise(Pas_de_derivation(Relation,ul))
  | Op,(UL_SUP | UL_INF | UL_EGAL | UL_DIFF | UL_SUPEGAL | UL_INFEGAL as op) -> [Term (term_of_ul op)]
  | Op,ul -> raise(Pas_de_derivation(Op,ul));;

(* Analyse_caractere dérive un 'acarctère' (de type v) en lisant une liste d'unités lexicales, et
retourne un arbre concret et la liste d'unités lexicales restant à lire
Si le caractère est un terminal, on vérifie que l'unité lexicale en tête de la liste d'ul correspond à
ce caractère et on créer alors la feuille de l'arbre concret;
Si le caractère est un non-terminal, on créer le noeud ayant pour fils la liste d'arbres concrets
créée par analyse_mot *)
let rec
(analyse_caractere : v * (unite_lexicale list) -> arbre_concret * (unite_lexicale list)) = function
    (Term (_)),liste -> (ACT (hd liste), tl liste)
  | (Non_term (_ as nterm)),liste -> let listeTerm = ma_derivation(nterm, hd liste) in
                                     let (listeAC,listeUL) = analyse_mot(listeTerm,liste) in
                                       (ACNT(nterm,listeAC),listeUL)
(* Analyse_mot dérive un mot (sous forme de v list) en lisant une liste d'unités lexicales, et
retourne une liste d'arbres concrets et la liste d'unités lexicales restant à lire. On traite la liste
de la liste de type (v list) caractère par caractère. En d'autres termes, on utilise analyse_caractère
poour le premier élément de la liste et analyse mot pour le reste *)
and
(analyse_mot : (v list) * (unite_lexicale list) -> (arbre_concret list) * (unite_lexicale list)) =
function
    [],liste -> [],liste
  | listeTerm,listeUL -> let (ac,nouvelleListe) = analyse_caractere(hd listeTerm,listeUL) in
                         let (suiteListeAC,nouvelleListe) = analyse_mot(tl listeTerm,nouvelleListe) in
                           (ac::suiteListeAC,nouvelleListe);;

(* construit_arbre_abstrait transforme un arbre concret en arbre abstrait. A chaque constructeur du
type arbre_concret, on dessine l'arbre abstrait correspondant *)
let rec construit_arbre_abstrait = function ACNT(nt,l) ->
    match (nt,l) with
        (S,[a;ACT UL_EOF])
      | (Facteurb,[ACT UL_OUVR;a;ACT UL_FERM])
      | (Facteurb,[a])
      | (Termb,[a;ACNT(SuiteTermb,[])])
      | (Expr,[a;ACNT(SuiteExpr,[])]) -> construit_arbre_abstrait a
      | (Expr,[a;ACNT(SuiteExpr,[ACT UL_OU;b])]) -> Ou (construit_arbre_abstrait
a,construit_arbre_abstrait b)
      | (Termb,[a;ACNT(SuiteTermb,[ACT UL_ET;b])]) -> Et (construit_arbre_abstrait
a,construit_arbre_abstrait b)
      | (Relation,[ACT (UL_IDENT a);ACNT(Op,[ACT op]);ACT (UL_IDENT b)]) -> Comp(a,op,b)
      | (Facteurb,[ACT UL_SI;a;ACT UL_ALORS;b;ACT UL_SINON;c;ACT UL_FSI]) -> Cond
(construit_arbre_abstrait a,construit_arbre_abstrait b,construit_arbre_abstrait c);;
```