Option informatique MP
TP OCaml (Python)
Validité d'une formule logique par les arbres de Beth.

Antoine MOTEAU
antoine.moteau@wanadoo.fr
lundi 27 août 2018

A l'aide des arbres de Beth, on peut décider si une formule du calcul propositionnel est valide (une tautologie), une antilogie (une contradiction) ou une contigence et, dans ce dernier cas, on peut exhiber des valeurs de variables qui satisfient ou falsifient la formule.
Ce document est présenté sous forme d'un TP (initialement OCaml), utilisant des ressources pré-définies, incluses dans le texte, où on demande principalement l'écriture d'un algorithme réalisant la méthode de Beth.

- Le corrigé OCaml des algorithmes demandés est disponible après l'énoncé du TP.
- Une version Python (3), complète, du corrigé, est disponible en fin de document.

Table des matières

Validité d'une formule par les arbres de Beth
Application au calcul propositionnel
TP avec OCaml

On se propose de mettre en œuvre la méthode des arbres de Beth pour décider de la validité de formules du calcul propositionnel. A l'aide des arbres de Beth, on pourra décider si une formule du calcul propositionnel est une tautologie, une antilogie (contradiction) ou une contigence et, dans ce dernier cas, exhiber des valeurs de variables qui satisfient ou falsifient la formule.

1 Introduction : description de la méthode de Beth, exemples

Cette introduction est la copie quasi-conforme d'un texte trouvé à l'adresse
`http://pauillac.inria.fr/caml/polycopies/herbelin/beth.ps.gz`
mais ce n'est plus joignable et j'ai même perdu le document initial ...
(cela viendrait de : Université Paris VII - Licence d'informatique - Enseignement de logique 1-er semestre. 1995-96 ?).
Le document initial étant bien écrit, clair, je n'ai pas changé grand-chose...

Le principe de la méthode est le suivant : partant d'une formule F que l'on cherche à falsifier (ou à satisfaire) on se ramène, étape par étape, à devoir falsifier (ou à satisfaire) des sous-formules toujours plus petites de F. Lorsqu'on a ramené le problème à la falsifiabilité (ou la satisfiabilité) uniquement d'atomes (variables ou constantes), on s'arrête et on regarde s'il est possible d'attribuer à ces atomes des valeurs de vérités adéquates.

Les règles de falsification sont les suivantes :

- Pour falsifier $A \wedge B$ il faut soit falsifier A, soit falsifier B.
- Pour falsifier $A \vee B$ il faut falsifier A et falsifier B.
- Pour falsifier $A \Rightarrow B$ il faut satisfaire A et falsifier B.
- Pour falsifier $A \Leftrightarrow B$ il faut soit satisfaire A et falsifier B, soit falsifier A et satisfaire B
- Pour falsifier $\neg A$ il faut satisfaire A.

Les règles de satisfaction sont les suivantes :

- Pour satisfaire $A \wedge B$ il faut satisfaire A et satisfaire B.
- Pour satisfaire $A \vee B$ il faut soit satisfaire A, soit satisfaire B.
- Pour satisfaire $A \Rightarrow B$ il faut soit falsifier A, soit satisfaire B.
- Pour satisfaire $A \Leftrightarrow b$ il faut soit satisfaire A et B, soit falsifier A et B
- Pour satisfaire $\neg A$ il faut falsifier A.

A partir de ces règles de falsification et de satisfaction, et partant de F, à falsifier ou à satisfaire, on construit un arbre.

Supposons par exemple que l'on cherche à falsifier F.

S'il y a deux manières de falsifier F (par exemple si F a la forme $A \wedge B$) alors on construit un embranchement de l'arbre avec, notées sur chaque sous-arbre, les nouvelles formules à falsifier (ici, A sur un des sous-arbres, B sur l'autre). S'il n'y a qu'une manière de falsifier F (par exemple si F a la forme $A \vee B$), on construit un arbre avec juste un unique sous-arbre sur lequel on note les formules restant à falsifier (ici A et B). Et on recommence le procédé de falsification sur le (ou les) sous-arbre(s) que l'on vient de construire.

Lorsqu'une formule à falsifier a la forme $A \Rightarrow B$, on est amené à devoir, cette fois, satisfaire une formule (ici A) et à falsifier une autre (ici B).

Ainsi, on se retrouve, à chaque étape, à devoir falsifier certaines formules et à devoir satisfaire d'autres. Dans la suite, on associera aux nœuds de l'arbre des notations de la forme $A_1, \cdots, A_p$ `vrai` $\mid B_1, \cdots, B_q$ `faux`. Cela signifiera qu'il faut falsifier les formules $B_1, \cdots, B_q$ et satisfaire les formules $A_1, \cdots, A_p$.

Puisqu'à chaque étape de la méthode, on se retrouvera avec de nouveaux sous-arbres sur lesquels on aura indiqué zéro, une ou plusieurs formules à falsifier, et zéro, une ou plusieurs formules à satisfaire, il faudra faire un choix de la formule sur laquelle on va poursuivre l'algorithme. De même, lorsqu'il y aura des embranchements, il faudra faire le choix du sous-arbre que l'on regarde en premier. Mais, de toutes façons, puisqu'à chaque étape on décompose une nouvelle formule, au bout d'un temps fini, on n'aura plus que des atomes.

Que se passe-t'il si on débouche sur un sous-arbre n'ayant que des formules atomiques à satisfaire ou à falsifier ? Deux cas peuvent se présenter. Le premier cas arrive lorsqu'aucun des atomes n'est à la fois à falsifier et à satisfaire. Par conséquent, on est capable d'exhiber une assignation des atomes telle que la formule F de départ soit falsifiable. Dans le deuxième cas, au contraire, il y a un même atome à rendre à la fois vrai et à la fois faux. C'est un échec : la branche en question ne peut fournir d'assignation des atomes qui permette de falsifier la formule F de départ. Il reste à explorer les autres branches, voir si elles permettent de trouver une assignation falsifiante de F. Et si, en fin de compte, aucune des branches ne permet de trouver une assignation falsifiante de F, c'est que F est valide (est une tautologie).

Exemple 1 : la formule $(A \wedge B) \Rightarrow (B \wedge A)$ est-elle valide ? (ide, est-elle une tautologie ?)
Avec comme objectif de falsifier la formule, on construit de proche en proche un arbre de Beth :

a. Initialisation :

$$\overbrace{(A \wedge B) \Rightarrow (B \wedge A)\ \texttt{faux}}$$

qui signifie : il faut falsifier $(A \wedge B) \Rightarrow (B \wedge A)$.

b. Décomposition de $\Rightarrow$:

$$\begin{array}{c} \overbrace{A \wedge B\ \texttt{vrai} \quad | \quad B \wedge A\ \texttt{faux}} \\ \hline (A \wedge B) \Rightarrow (B \wedge A)\ \texttt{faux} \end{array}$$

qui signifie : il faut satisfaire $A \wedge B$ et falsifier $B \wedge A$.

c. On choisit par exemple de décomposer $B \wedge A$:

$$\begin{array}{c} \overbrace{A \wedge B\ \texttt{vrai} \quad | \quad B\ \texttt{faux}} \qquad \overbrace{A \wedge B\ \texttt{vrai} \quad | \quad A\ \texttt{faux}} \\ \hline A \wedge B\ \texttt{vrai} \quad | \quad B \wedge A\ \texttt{faux} \\ \hline (A \wedge B) \Rightarrow (B \wedge A)\ \texttt{faux} \end{array}$$

qui signifie : il faut soit continuer à satisfaire $A \wedge B$ tout en falsifiant B, soit continuer à satisfaire $A \wedge B$ tout en falsifiant A.

d. On choisit par exemple de décomposer $A \wedge B$ dans le nœud de gauche :

$$\begin{array}{c} \overbrace{A\,,\,B\ \texttt{vrai} \quad | \quad B\ \texttt{faux}} \\ \hline A \wedge B\ \texttt{vrai} \quad | \quad B\ \texttt{faux} \qquad A \wedge B\ \texttt{vrai} \quad | \quad A\ \texttt{faux} \\ \hline A \wedge B\ \texttt{vrai} \quad | \quad B \wedge A\ \texttt{faux} \\ \hline (A \wedge B) \Rightarrow (B \wedge A)\ \texttt{faux} \end{array}$$

qui signifie : il faut, sur cette branche, satisfaire A et B tout en falsifiant B. C'est évidemment impossible : la branche est un échec.

e. Il reste à essayer la décomposition de $A \wedge B$ dans le nœud de droite :

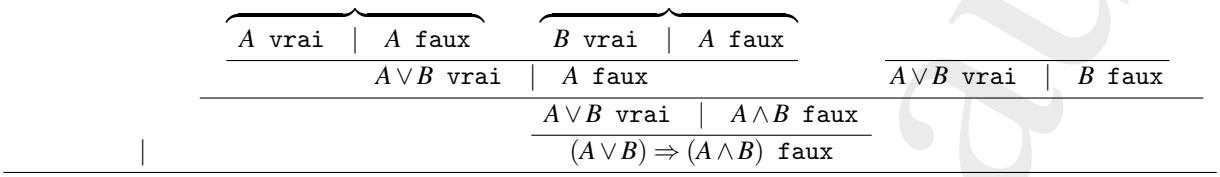
$$\begin{array}{c} \overline{\overline{A\,,\,B\ \texttt{vrai} \quad | \quad B\ \texttt{faux}}} \qquad \overbrace{A\,,\,B\ \texttt{vrai} \quad | \quad A\ \texttt{faux}} \\ \hline A \wedge B\ \texttt{vrai} \quad | \quad B\ \texttt{faux} \qquad A \wedge B\ \texttt{vrai} \quad | \quad A\ \texttt{faux} \\ \hline A \wedge B\ \texttt{vrai} \quad | \quad B \wedge A\ \texttt{faux} \\ \hline (A \wedge B) \Rightarrow (B \wedge A)\ \texttt{faux} \end{array}$$

là aussi, c'est un échec. Il n'y a alors plus de branches à explorer.

Aucune branche ne permet donc de falsifier la formule $(A \wedge B) \Rightarrow (B \wedge A)$. Cette dernière est donc une formule valide (une tautologie).

Exemple 2 : $(A \vee B) \Rightarrow (A \wedge B)$ est-elle valide ? (ide, est-elle une tautologie ?)

Avec comme objectif de falsifier la formule, après quelques étapes (et quelques choix, ici décomposition plutôt orientée vers les nœuds de gauche) on arrive à l'arbre (non complètement développé ici) :



La branche la plus à gauche, qui conduit au nœud annoté par A `vrai` | A `faux` est un échec, la branche suivante conduit au nœud annoté par B `vrai` | A `faux` et on obtient une assignation invalidante en assignant `vrai` à B et `faux` à A. Par construction de l'arbre, on déduit que cette même assignation falsifie la formule $(A \vee B) \Rightarrow (A \wedge B)$ qui n'est donc pas valide (pas une tautologie).

2 Environnement de programmation (OCaml)

2.1 Arbre formule propositionnelle

On utilise une représentation arborescente pour décrire les formules :

```
type formule =      (* arbre formule *)
    Var of string   (* nom de variable, ici réduit à 1 caractère, de a à z *)
  | Vrai | Faux
  | Non of formule
  | Et of formule * formule  | Ou of formule * formule
  | Imp of formule * formule | Equ of formule * formule
;;
```

2.2 Interface (construction, lecture d'un arbre formule)

2.2.1 Formules représentées par une chaîne de caractères

Une formule peut être écrite comme une chaîne de caractères :

- ELTP : Expression Logique Totalement Parenthésée ou quasi Totalement Parenthésée, sans règles de priorité.

  Par exemple `"(((e) Et (t)) => ((a) Ou (Non (b))))"` est une ELTP

  et `"((e Et t) => (a Ou Non b))"` est une formule quasi-ELTP (c'est plus léger).
- ELNTP : Expression Logique Non Totalement Parenthésée, qui utilise des règles de priorité ("usuelles" ?).

  Par exemple, `"e Ou t Ou u Et v => a Ou Non b"; "a => b => c"` seraient interprétées comme les quasi-ELTP `"((e Ou (t Ou (u Et v))) => (a Ou Non b))"; "(a => (b =>c))"`.

Ici, pour simplifier, **les noms de variables sont réduits à une seule lettre minuscule a . . . z**, non accentuée,
et on n'utilise que des formules **quasi**-ELTP (ELQTP), plus simples à gérer que les formules ELNTP :

- **avec ou sans parenthèses autour d'une variable a, d'un Non a** :
  `"a", "(a)", "Non a", "(Non a)", "Non ((a)+b)", "(Non((a)+b))"` sont acceptées,
- **avec parenthèses obligatoires autour d'une sous-expression binaire** `g op d` :
  `"(a+b)", "(a + (b+c))", "Non(a+b)` sont acceptées ; `"a+b", "(a+b+c)", "Non a + b"` sont rejetées.
- **avec ou sans parenthèses superflues** : `"((( a )))", ((a+b))", "((( ((a))+b )))"` sont acceptées.

Les formules ELQTP (quasi complètement parenthésées), seront conformes à la grammaire :

$$A ::= \underbrace{a|\ldots|z}_{\texttt{variable}} \mid \underbrace{0}_{\texttt{faux}} \mid \underbrace{1}_{\texttt{vrai}} \mid \underbrace{\neg A}_{\substack{\texttt{Non } A \\ \textit{avec N} \\ \sim A}} \mid \underbrace{(A \wedge A)}_{\substack{(A\ \texttt{Et}\ A) \\ (A\ \texttt{et}\ A) \\ (A\cdot A),(A*A)}} \mid \underbrace{(A \vee A)}_{\substack{(A\ \texttt{Ou}\ A) \\ (A\ \texttt{ou}\ A) \\ (A+A)}} \mid \underbrace{(A \Rightarrow A)}_{(A\ \texttt{=>}\ A)} \mid \underbrace{(A \Leftrightarrow A)}_{(A\ \texttt{<=>}\ A)} \mid (A)$$

les espaces entre constituants étant non significatifs :

`"(Non((aEtNonb)=>a)=>b)" , "  (~((a.~b)=>  a  ) =>b ) " sont acceptées,`
`"N on a" , "(Non((a Et Non b) => a) = >b)" , "(~((a.~b)=>a)= >b)" sont refusées.`

2.2.2 Arbre formule déduit d'une chaîne quasi-ELTP

On disposera de la fonction de construction de formule (voir préambule) :

`formule_of_string : string -> formule`

formule_of_string s renvoie la formule construite à partir de la formule quasi-ELTP contenue dans la chaîne de caractères s. Provoque les exceptions Stream.Failure ou Stream.Error en cas d'erreur d'analyse du flux (formule vide, contenant des caractères non reconnus, incorrecte, mal parenthésée, sur-parenthésée).

2.2.3 Chaîne (quasi-ELTP) déduite d'un arbre formule

La fonction réciproque de la précédente est plus facile à écrire : à l'aide d'un parcours en profondeur classique d'un arbre formule, on construit, par concaténation de chaînes, l'expression quasi-ELTP correspondant à la formule représentée par l'arbre.

▶ `string_of_formule: formule -> string` A ECRIRE

string_of_formule p renvoie la chaîne représentant, sous forme quasi-ELTP, la formule représentée par l'arbre p.

2.2.4 Représentation graphique d'un arbre formule

On disposera de la ressource de dessin d'arbre (voir préambule) utilisant la bibliothèque Graphics :

`dessine_formule : formule -> unit = <fun>`

dessine_formule p représente (tant bien que mal) l'arbre formule p (pas trop compliqué) sur la fenêtre graphique de Caml.

2.3 Programmation par exceptions (illustrations)

1. Traitement des exceptions renvoyées par l'analyse de flux : `Stream.Failure` et `Stream.Error`. Par exemple,

```
try let f = formule_of_string s  in
    ....
with Stream.Failure as ex -> print_string (s ^ " est incorrecte\n"); raise ex
   | Stream.Error e as ex -> print_string (s ^ " est incorrecte\n"); raise ex
```

2. Exceptions définies par l'utilisateur, provoquées, interceptées ou non :

```
exception Echec;;       (* exception définie par l'utilisateur *)
...
let machin = function
   | ... -> ...
   | ... -> raise Echec      (* exception provoquée *)
;;
...
begin try let u = machin f1 in ...
      with Echec -> try let u = machin f2 in ...
                    with Echec -> ...                (* 2-ième : interceptée *)
end
...
begin try let u = machin f1 in ...
      with Echec -> let u = machin f2 in ...         (* 2-ième : non interceptée *)
end
```

3 Le TP : la méthode de Beth (OCaml)

3.1 Interprétation d'une liste de formules

Une interprétation d'une liste de formules est une liste de couples (variable, valeur) qui pourra être

- falsifiante (les valeurs attribuées aux variables rendent les formules de la liste fausses)
- satisfiante (les valeurs attribuées aux variables rendent les formules de la liste vraies)

Remarque. Les formules de la liste vide sont à la fois vraies et fausses pour n'importe quelle interprétation.

Au cours de l'évaluation de l'arbre de Beth, les couples (variable, valeur) seront ajoutés un à un dans une interprétation initialement vide. Lorsque l'on aura à ajouter le couple (v, valeur) dans l'interprétation I,

- si I ne contient pas d'assignation à v, on ajoute l'assignation (v, valeur) à I,
- si I contient déjà l'assignation (v, valeur), I reste inchangé,
- si I contient déjà une assignation à v qui n'est pas (v, valeur), c'est contradictoire (`raise Echec`).

On peut, si on veut, définir un type pour les interprétations :

```
type interpretation = (char * bool) list;;
exception Echec;;     (* nom d'une exception privée, que l'on provoquera et traitera *)
```

Ecrire la fonction :

▶ `ajoute : char -> bool -> interpretation -> interpretation` `2 lignes A ECRIRE`

ajoute v valeur i ajoute le couple (v, valeur) à la liste i si celle-ci ne contient pas déjà un couple (v, valeur_b) et provoque l'erreur Echec si i contient un couple (v,valeur_b) avec valeur_b $\neq$ valeur.

Remarque. On pourra utiliser la fonction `List.assoc` décrite dans l'aide OCaml :

```
val assoc : 'a -> ('a * 'b) list -> 'b
```

assoc a l returns the value associated with key a in the list of pairs l. That is, assoc a [... ; (a,b) ; ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

3.2 Construction (implicite) et évaluation de l'arbre de Beth

Un nœud de l'arbre de Beth est un couple (g,d) où

- g est une liste de formules à satisfaire (affectée de la marque `vrai` dans les exemples ci-dessus),
- d est une liste de formules à falsifier (affectée de la marque `faux` dans les exemples ci-dessus) ;

et on cherche une interprétation (initialement vide) qui soit satisfiante pour g et falsifiante pour d.

Avec une interprétation i, on traite un couple (g,d), **de façon récursive**, en traitant d'abord g et puis d :

1. Traitement de g (liste des formules à satisfaire)
   - Lorsque g est vide, on passe au traitement de d
   - Lorsque la formule de tête de g est
     (a) une formule élémentaire
       - réduite à une variable p, on tente l'assignation (p, true) dans l'interprétation et, s'il n'y a pas eu d'échec, on continue avec le reste de g, c'est à dire (queue g), à satisfaire et d à falsifier ;
       - réduite à la constante vrai, on passe et on continue avec le reste de g, (queue g), à satisfaire et d à falsifier ;
       - réduite à la constante faux, c'est contradictoire (`raise Echec`) ;

     (b) de la forme $\neg f$, on continue avec (queue g) à satisfaire et $f :: d$ à falsifier ;

     (c) de la forme $f_1 \wedge f_2$, on continue avec $f_1 :: f_2 ::$ (queue g) à satisfaire et d à falsifier ;

     (d) de la forme $f_1 \vee f_2$,
     on essaie de satisfaire $f_1 ::$ (queue g) et de falsifier d ;
     en cas d'échec, en revenant à l'interprétation initiale (sauvegardée avant le premier essai (*)),
     on essaie de satisfaire $f_2 ::$ (queue g) et de falsifier d ;
     en cas d'échec c'est ... un échec (qui sera intercepté ou non quelque part ...).

     (e) de la forme $f_1 \Rightarrow f_2$, ...

     (f) de la forme $f_1 \Leftrightarrow f_2$, ...

     Ces différents cas peuvent aboutir à un échec, répercuté à l'appelant (qui peut être l'algorithme lui même) et intercepté ou non par celui ci.
2. Traitement de d (liste des formules à falsifier) :
   - Lorsque g et d sont devenus vide, c'est fini. Il n'y a pas eu d'échec et on renvoie l'interprétation où l'on a accumulé les assignations de variables qui ont satisfié les formules gauches et falsifié les formules droites rencontrées lors de l'algorithme.
   - Lorsque g n'est pas vide, on revient au traitement de g
   - Sinon, raisonnement analogue à ce qui a été fait pour g .....

On en déduit la fonction récursive :

▶ `beth : interpretation -> formule list * formule list -> interpretation` `A ECRIRE`

beth [ ] g , d renvoie une interprétation qui est satisfiante pour les formules de la liste g et falsifiante pour les formules de la liste d. **Provoque l'exception Echec s'il n'y a pas de telle interprétation**.

(*) Avec OCaml, l'interprétation n'aura pas été modifié en cas d'Echec (si ce paramètre est passé par valeur).
Ce n'est pas le cas avec Python, il faut faire une sauvegarde explicite (paramètres toujours passés par référence).

Les interprétations seront éditées par une fonction spécialisée (parcours usuel de liste) :

▶ `print_interpretation : interpretation -> unit` A ECRIRE

print_interpretation i édite sur le terminal, ligne par ligne, le contenu de l'interprétation i, nom de variable et valeur associée, le couple `(v, valeur)` où `valeur` $\in \{\texttt{vrai} = 1, \texttt{faux} = 0\}$, s'éditant par l'instruction :
`printf "%s=%d " v (if valeur then 1 else 0);`.

Remarques.

`printf` est une fonction de la bibliothèque `printf` de Caml : bibliothèque d'éditions formatées sur xx places, selon un modèle (voir aide Caml).

Le passage à la ligne peut se faire en incorporant un "\n" en fin de la chaîne de format : `"%s=\%d \n"`
ou par un `print_newline ()`.

3.3 Tautologie, contradiction ou contingence

Soit F la formule (arbre) construite d'après la chaîne s.

Remarque. S'il y a eu une erreur dans la construction (exceptions Stream.Failure ou Stream.Error), c'est que la formule saisie dans s est incorrecte.

On cherche une interprétation I_s qui satisfait $[F]$ (et qui falsifie $[\,]$)
En cas d'échec (exception Echec), F est une antilogie (ou contradiction)
Sinon, on cherche une interprétation I_f (qui satisfait $[\,]$ et) qui falsifie $[F]$
En cas d'échec (exception Echec), F est valide (une tautologie)
Sinon, F est contingente, I_s est une interprétation satisfiante de F
I_f est une interprétation falsifiante de F

On construit, selon cet algorithme, la fonction :

▶ `analyse : string ->  unit` A ECRIRE

analyse s édite sur le terminal les messages résultant de l'application de la méthode de Beth à la formule contenue dans la chaîne s, selon l'algorithme précédent, en produisant les messages :
"Formule incorrecte" (en cas d'erreur de construction), "Tautologie", "Antilogie" ou "Contingence".
Dans le cas d'une "Contingence", il y aura aussi édition d'une interprétation satisfiante et d'une interprétation falsifiante.

3.4 Préambule : éléments de programme pré-écrits

```
#load "graphics.cma";;
open Graphics;;
open Printf;;
#load "dynlink.cma" ;;  (* Dynamic loading of object files. *)
#load "camlp4o.cma" ;;  (* Pre-Processor-Pretty-Printer for OCaml, parsing and printing *)
open_graph " 1200x800+10+10";;
Graphics.set_font("9x15bold");;  (* + lisible *)

type formule =      (* arbre formule *)
    Var of string   (* nom de variable, ici réduit à 1 caractère, de a à z *)
  | Vrai | Faux
  | Non of formule
  | Et of formule * formule  | Ou of formule * formule
  | Imp of formule * formule | Equ of formule * formule
;;
```

(* Construction d'un arbre formule à partir d'un flux de char représentant une quasi-ELTP : *)

```
let rec fini = parser                          (* supprime "blancs" restants jusqu'à fin de flux attendue *)
     [< '( ' ' | '\t' | '\n' ); r = fini >] -> r
   | [<  r = Stream.empty >] -> r
and     parenthese_droite = parser                (* supprime "blancs" éventuels devant ) attendue *)
     [< '( ' ' | '\t' | '\n' ); r = parenthese_droite >] -> r
   | [< '')' >] -> ()
and     connecteur = parser
     [< '( ' ' | '\t' | '\n' ); r = connecteur >] -> r   (* suppression "blancs" de tête *)
   | [< '( 'e' | 'E' ); ''t' >]  -> (fun f1 f2 -> Et (f1,f2))   (* et, Et  *)
   | [< ''.' | '*' >]            -> (fun f1 f2 -> Et (f1,f2))   (* . * == Et *)
   | [< '( 'o' | 'O' ); ''u' >]  -> (fun f1 f2 -> Ou (f1,f2))   (* ou, Ou  *)
   | [< ''+' >]                  -> (fun f1 f2 -> Ou (f1,f2))   (* + == Ou *)
   | [< ''='; ''>' >]            -> (fun f1 f2 -> Imp (f1,f2))
   | [< ''<'; ''='; ''>' >]      -> (fun f1 f2 -> Equ (f1,f2))
and     construit = parser
     [< '( ' ' | '\t' | '\n' ); r = construit >] -> r   (* suppression "blancs" de tête *)
   | [< ''('; f1 = construit; r = suite f1 >] -> r
   | [< ''N' ; ''o'; ''n'; f = construit >]  -> Non f     (* N imposé: sinon pb, exemple: "n ou a" *)
   | [< ''~' ; f = construit >]              -> Non f     (* ~ (tilde) pour Non *)
   | [< ''1' >]           -> Vrai
   | [< ''0' >]           -> Faux
   | [< ''a'..'z' as p >] -> Var (String.make 1 p)   (* minuscule *)
and     suite f1 = parser
     [< '( ' ' | '\t' | '\n' ); r = suite f1 >] -> r   (* suppression "blancs" de tête *)
   | [< '')' >] -> f1
   | [<  c = connecteur; f2 = construit; r = parenthese_droite >] -> r;  c f1 f2
;;
(*  val fini : char Stream.t -> unit = <fun>
    val parenthese_droite : char Stream.t -> unit = <fun>
    val connecteur : char Stream.t -> formule -> formule -> formule = <fun>
    val construit : char Stream.t -> formule = <fun>
    val suite : formule -> char Stream.t -> formule = <fun>
*)

let formule_of_string s =
  let construction = parser
    [< f = construit; r = fini >] -> r; f
  in
  try construction (Stream.of_string s)
  with Stream.Failure as ex -> print_string (s ^ " n'est pas une formule correcte\n"); raise ex
     | Stream.Error e as ex -> print_string (s ^ " n'est pas une formule correcte\n"); raise ex
;;
(* val formule_of_string : string -> formule = <fun> *)
```

(* Eléments de représentation graphique, un peu primitif, d'un arbre formule : *)

```
let dessine_formule t =
  let graph_dx = (25 * fst (text_size "0") )/15  (* espacement en x *)
  and graph_dy = (3 * snd (text_size "0") ) in   (* espacement en y *)

  let rec ecart = function                       (* écartement variable entre noeuds *)
    | Var _ | Vrai | Faux -> 1
    | Non (p)   -> 1 + ecart p
    | Et (g,d)  -> (ecart g) + (ecart d)
    | Ou (g,d)  -> (ecart g) + (ecart d)
    | Imp (g,d) -> (ecart g) + (ecart d)
    | Equ (g,d) -> (ecart g) + (ecart d)

  and     dessine_noeud x y t =
    let tmp x y g d s =
         let zg = max 1 (ecart g) and zd = max 1 (ecart d ) in
         lineto x y; dessine_noeud (x - zd * graph_dx) (y - graph_dy) g;
         moveto x y; dessine_noeud (x + zg * graph_dx) (y - graph_dy) d;
         moveto x y; draw_circle x y 14;
                     set_color white; fill_circle  x y 13; set_color black;
                     let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;
    in
    match t with
      | Var s   -> lineto x y;
                   set_color white; fill_circle  x y 13; set_color black;
                   let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

      | Vrai    -> lineto x y;
                   set_color white; fill_circle  x y 13; set_color black;
                   let s = "1" in
                   let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

      | Faux    -> lineto x y;
                   set_color white; fill_circle  x y 13; set_color black;
                   let s = "0" in
                   let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

      | Non (p) -> lineto x y;
                   dessine_noeud x (y - graph_dy) p;
                   moveto x y; draw_circle x y 14;
                   set_color white; fill_circle  x y 13; set_color black;
                   let s = String.make 1 (char_of_int 172) in
                   let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

      | Et (g,d)  -> tmp x y g d "Et"
      | Ou (g,d)  -> tmp x y g d "Ou"
      | Imp (g,d) -> tmp x y g d "=>"
      | Equ (g,d) -> tmp x y g d "<=>"
  in
  let  x = (size_x () /2) and y = (size_y () - graph_dy) in
  clear_graph ();
  moveto x y; dessine_noeud x y t
;;
(* dessine_formule: formule -> unit = <fun> *)

let s = "(((a ou Non b) <=> ((a et d) => (((b + c) ou (e Ou f)) ou (g et h))))" ^
        " => ((a ou Non b) <=> ((a et Non ~ Non d) => (b ou c))) )";;
let p = formule_of_string s in dessine_formule p;;
```

3.5 Squelette (suite du préambule) : fonctions à écrire

Non exécutable !

```
(*-- Expression d'un arbre, sous forme de chaîne quasi-ELTP *)
let rec string_of_formule = function
   ..............                                                    7 lignes A ECRIRE
;;
(* string_of_formule : formule -> string = <fun> *)

type interpretation = (char * bool) list;;
exception Echec
;;
(*--- Ajout d'une interprétation  ----*)
let ajoute v valeur (i :interpretation) =
   ..............                                                    2 lignes A ECRIRE
;;
(* ajoute : char -> bool -> interpretation -> interpretation = <fun> *)

(*--- Construction et évaluation de l'arbre de Beth ---*)
let rec beth (i :interpretation) = function
   ..............                                                 2 * 11 lignes A ECRIRE
;;
(* beth : interpretation -> formule list * formule list -> interpretation =  <fun> *)

(*--- Tautologie, contradiction et contingence ---*)
let rec (print_interpretation : interpretation -> unit) = function
   ..............                                                  2 à 3 lignes A ECRIRE
;;
(* print_interpretation : interpretation -> unit = <fun> *)

let analyse s =                                                      9 lignes A ECRIRE
  try ........
      try ...........
          try ...........
          with Echec ->  ........
      with Echec ->  ........
  with Stream.Failure ->  ........
     | Stream.Error e ->  ........
;;
(* analyse : string -> unit = <fun> *)

(*--- Exemple de programme principal (simpliste) ---*)
let main () =
  print_string "Bonjour ... \n";
  print_string "Pour terminer, tapez $;; en guise de formule";

  try while true
      do print_string "\nEntrez une formule, terminée par \";;\": ";
         let s = ref (read_line ()) in
         begin
               try while !s.[String.length !s -1] = ';'
                   do s := String.sub !s 0 (String.length !s -1); done;
               with _ -> s := "$";
         end;
         if !s.[0] = '$' then failwith "Fini."
         else analyse !s
      done
  with Failure w -> print_string ("\n" ^ w ^ "Au revoir...\n")
      | _        -> print_string "\nErreur indéterminée ... abandon\n"
;;
(* main : unit -> unit = <fun> *)

main ();;
```

4 Corrigé TP Arbres de Beth, avec OCaml

(* Complément au préambule *)

```
type interpretation = (char * bool) list;;
exception Echec;;

let rec (print_interpretation : interpretation -> unit) = function
      []            -> print_newline ()
    | (v,valeur)::i -> printf "%s=%d " v (if valeur then 1 else 0);
                       print_interpretation i
;;
(* print_interpretation : interpretation -> unit = <fun> *)

let ajoute v valeur (i :interpretation) =
  try if (List.assoc v i) = valeur then i else raise Echec
  with Not_found -> (v,valeur)::i
;;
(* ajoute : char -> bool -> interpretation -> interpretation = <fun> *)

(*-------------------------------------------------------*)
(*--- Construction et évaluation de l'arbre de Beth ---*)
(*-------------------------------------------------------*)
let rec beth (i :interpretation) = function
   (* Must not be directly called without management of error! *)
    ([], [])              -> i
  | ([], (Var p)::r)      -> beth (ajoute p false i) ([], r)
  | ([], Vrai::r)         -> raise Echec
  | ([], Faux::r)         -> beth i ([], r)
  | ([], (Non f)::r)      -> beth i ([f], r)
  | ([], (Et(f1,f2))::r)  -> begin try beth i ([], f1::r)    with Echec -> beth i ([], f2::r) end    (* 1 *)
  | ([], (Ou(f1,f2))::r)  -> beth i ([], f1::f2::r)
  | ([], (Imp(f1,f2))::r) -> beth i ([f1], f2::r)
  | ([], (Equ(f1,f2))::r) -> begin try beth i ([f1], f2::r)  with Echec -> beth i ([f2], f1::r) end  (* 1 *)

  | ((Var p)::r, d)       -> beth (ajoute p true i) (r, d)
  | (Vrai::r, d)          -> beth i (r, d)
  | (Faux::r, d)          -> raise Echec
  | ((Non f)::r, d)       -> beth i (r, f::d)
  | ((Et(f1,f2))::r, d)   -> beth i (f1::f2::r, d)
  | ((Ou(f1,f2))::r, d)   -> begin try beth i (f1::r, d)     with Echec -> beth i (f2::r, d) end     (* 1 *)
  | ((Imp(f1,f2))::r, d)  -> begin try beth i (r, f1::d)     with Echec -> beth i (f1::f2::r, d) end (* 1 *)
  | ((Equ(f1,f2))::r, d)  -> begin try beth i (f1::f2::r, d) with Echec -> beth i (r, f1::f2::d) end (* 1 *)
;;
(* beth : interpretation -> formule list * formule list -> interpretation =  <fun> *)
```

(* (* 1 *) : en cas d'Echec, i est inchangé, il n'y a pas besoin de faire un copie préalable (ce ne sera pas le cas avec Python).*)

(* **Problème ; est-ce un comportement normal, pérenne, ou une particularité de la version de OCaml utilisée ?** *)

```
(*-------------------------------------------------*)
(*--- Tautologie, contradiction ou contingence ---*)
(*-------------------------------------------------*)
let analyse s =
  try let f = formule_of_string s in
      try let i_satisfie = beth [] ([f],[]) in
          try let i_falsifie = beth [] ([],[f]) in
              print_string (s^" est contingente\n");
              print_string ("Voici une interprétation satisfiante: "); print_interpretation i_satisfie;
              print_string ("Voici une interprétation falsifiante: "); print_interpretation i_falsifie
          with Echec -> print_string (s^" est une tautologie\n")
      with  Echec -> print_string (s^" est une antilogie\n")
   with  Stream.Failure -> print_string (s ^ " ne représente pas une formule correcte\n")
       | Stream.Error e -> print_string (s ^ " ne représente pas une formule correcte\n")
;;
(* analyse : string -> unit = <fun> *)
```

```
(*--------------------------------------------------------------------*)
(*-- Expression d'un arbre formule, sous forme de chaine quasi-ELTP *)
(*--------------------------------------------------------------------*)
let rec string_of_formule = function    (* ELQTP *)
    Var a     -> a
  | Vrai      -> "1"
  | Faux      -> "0"
  | Non f     -> "Non " ^ string_of_formule f
  | Et (g,d)  -> "(" ^ string_of_formule g ^ " Et "  ^ string_of_formule d ^ ")"
  | Ou (g,d)  -> "(" ^ string_of_formule g ^ " Ou "  ^ string_of_formule d ^ ")"
  | Imp (g,d) -> "(" ^ string_of_formule g ^ " => "  ^ string_of_formule d ^ ")"
  | Equ (g,d) -> "(" ^ string_of_formule g ^ " <=> " ^ string_of_formule d ^ ")"
;;
(* string_of_formule : formule -> string = <fun> *)

let p =  Imp (Et (Var "e", Var "t"), Imp (Var "a", Ou (Var "n", Var "e"))) in string_of_formule p;;
let s = "((a et Non b) => (a => (b Ou c)))" in let p = formule_of_string s in string_of_formule p;;
```

(* On peut même introduire certaines simplifications des arbres Formule, par exemple : *)

```
let rec reduit_unaire = function
  | Non f     -> let u = reduit_unaire f in
                 begin match u with
                     Vrai  -> Faux
                   | Faux  -> Vrai
                   | Non h -> h
                   | _     -> Non u
                 end
  | Equ (g,d) -> Equ (reduit_unaire g, reduit_unaire d)
  | Imp (g,d) -> Imp (reduit_unaire g, reduit_unaire d)
  | Et (g,d)  -> Et (reduit_unaire g, reduit_unaire d)
  | Ou (g,d)  -> Ou (reduit_unaire g, reduit_unaire d)
  | x -> x
;;
(* val reduit_unaire : formule -> formule = <fun> *)
```

(* On pourrait aussi réduire les "(a Et 1)", "(1 Et a)", "(0 => a)", ... *)

```
(*---------------------------------------*)
(*--- Programme principal (simpliste) ---*)
(*---------------------------------------*)
let main () =
  print_string "Bonjour ... \n";
  print_string "Pour terminer, tapez $;; en guise de formule";

  try while true
      do print_string "\n Entrez une formule, terminée par ;;: ";
         let s = ref (read_line ()) in
         begin
              try while !s.[String.length !s -1] = ';'
                  do s := String.sub !s 0 (String.length !s -1); done;
               with _ -> s := "$";
         end;
         if !s.[0] = '$' then failwith "Fini."
         else analyse !s
      done
  with Failure w -> print_string ("\n" ^ w ^ "Au revoir...\n")
      | _        -> print_string "\n Erreur indéterminée ... abandon\n"
;;
(* main : unit -> unit = <fun> *)

main ();;
```

5 Corrigé TP Arbres de Beth, avec Python

Rédigé par transformation en parallèle depuis le code OCaml.
Remarques. Cela ne respecte pas entièrement la PEP 8 ni la PEP 257 ; j'ai privilégié une rédaction qui garde les portions de code homogènes dans une seule page et les lignes font parfois plus de 79 caractères . . . **Scandale !** et il peut y avoir plusieurs instructions par ligne . . . **Re Scandale !** et il y en a d'autres . . .
La doc est réduite, mais j'ai utilisé des noms significatifs (franglais) et les annotations de fonctions (PEP 3107).

5.1 Arbre formule propositionnelle

```
class Lexeme(object):
    """Classe mère, non instanciée directement."""
    def __str__(self):
        return "{0}".format(self.value)

class Vrai(Lexeme):
    value = 1
class Faux(Lexeme):
    value = 0
class Variable(Lexeme):
    def __init__(self, value: str ):
        self.value = value              # nom de variable. Ici 1 char a..z

class Non(Lexeme):          # ~ (Non) unaire
    value = "~"             # symbole du Non: tilde

class Binaire(Lexeme):
    """Classe mère, non instanciée directement."""
class Et(Binaire):
    value = "*"
class Ou(Binaire):
    value = "+"
class Equ(Binaire):
    value = "<=>"
class Imp(Binaire):         # seul non associatif
    value = "=>"

class Formule:
    """Arbre de lexèmes."""
    def __init__(self, n: Lexeme = None, g: Lexeme = None, d: Lexeme = None, **kwargs: "not used"):
        """avec vérification de cohérence."""
        if isinstance(n, Lexeme):
            if isinstance(n, Binaire):
                if not isinstance(g, Formule) or not isinstance(d, Formule):
                    raise Exception("Invalid child.")
            elif isinstance(n, Non):
                if not isinstance(g, Formule):
                    raise Exception("Invalid child.")
                if not d is None:
                    raise Exception("No right child admitted here.")
            elif isinstance(n, (Vrai, Faux, Variable)):
                if (d is not None) or (g is not None):
                    raise Exception("Childs not admitted here.")
            else: raise Exception("Incorrect type for node.")
            self.node  = n
            self.gauche = g
            self.droit  = d
        else: raise Exception("Incorrect type for node")
```

5.2 Construction d'un arbre formule depuis une chaîne de caractères quasi-ELTP

```
def vide_tete(s: str) -> str:
    """Suppression des caractères "blancs" (espace, \t, \n) en tête de chaîne."""
    try:
        while s[0] in " \t\n": s = s[1:]
    except: pass
    return s

def fini(s: str) -> None:
    """Suppression des caractères "blancs" (espace, \t, \n) avant fin de chaîne attendue."""
    s = vide_tete(s)
    if len(s) > 0: raise ValueError("Too much chars")

def parenthese_droite(s: str) -> str:
    """Suppression des caractères "blancs" (espace, \t, \n) devant ) attendue."""
    s = vide_tete(s)
    if len(s) == 0: raise ValueError("Missing close braket )")
    else:
        if s[0] != ")": raise ValueError("Not a close bracket )")
        else: return s[1:]

def connecteur(s: str) -> Binaire:
    s = vide_tete(s)
    if len(s) == 0: raise ValueError("Connector not found")
    try:
        if s[0:3] == "<=>": return Equ(), s[3:]
        else: raise ValueError("Pb 1")
    except:
        try:
            if s[0:2] == "=>": return Imp(), s[2:]
            elif s[0:2].upper() == "ET": return Et(), s[2:]
            elif s[0:2].upper() == "OU": return Ou(), s[2:]
            else: raise ValueError("Pb 2")
        except:
            if   s[0] in "*.": return Et(), s[1:]
            elif s[0] == "+":  return Ou(), s[1:]
            else: raise ValueError("Not a connector")

def construit(s: str) -> (Formule, str):
    s = vide_tete(s)
    if len(s) == 0: raise ValueError("Empty formula")
    if   s[0] == "(": f1, s = construit(s[1:]); return suite(f1,s)
    elif s[0] == "~": f, s = construit(s[1:]); return Formule( Non(), f), s
    elif s[0] == "N":
        try:
            if s[1:3].upper() == "ON": f, s = construit(s[3:]); return Formule( Non(), f), s
            else: raise ValueError("Invalid char")
        except: raise ValueError("Invalid char")
    elif s[0] == "1": return Formule( Vrai() ), s[1:]
    elif s[0] == "0": return Formule( Faux() ), s[1:]
    elif s[0] in "abcdefghijklmnopqrstuvwxyz": return Formule( Variable(s[0]) ), s[1:]
    else: raise ValueError("Invalid char")

def suite(f1: Formule, s: str) -> (Formule, str):
    try: c, s = connecteur(s);
    except ValueError: s = parenthese_droite(s); return f1, s
    else: f2, s = construit(s); s = parenthese_droite(s); return Formule(c, f1, f2), s

def formule_of_string(s: str) -> Formule:
    """Depuis une chaîne de caractères quasi-ELTP-stricte."""
    f, s = construit(s);
    fini(s);
    return f
```

5.3 Repésentation graphique d'un arbre formule avec matplotlib (simpliste)

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(10,10))
ax = fig.gca()
plt.box("off"); plt.axis("off")  # "off": supp axis, grid. Sinon: auto-tracés
plt.tight_layout(0.1)
ax.axis("equal")
fig.set_facecolor("white");
#------------------------------ simulation du module graph de OCaml
xc,yc,cc = 0,0,"black"   # currents

def set_color(c: str):
    global cc
    cc = c
def moveto(x,y):
    global xc, yc
    xc, yc = x,y
def lineto(x,y):
    global xc, yc
    ax.plot([xc,x],[yc,y], color=cc, zorder=1)
    xc, yc = x,y
def draw_circle(x,y,r):
    acircle = plt.Circle( [x,y],r, ec=cc, fill=False, zorder=2)
    ax.add_patch(acircle)
    ax.autoscale_view()
def fill_circle(x,y,r):
    acircle = plt.Circle( [x,y],r, linewidth=1, ec="white", fc=cc, fill=True, zorder=2)
    ax.add_patch(acircle)
    ax.autoscale_view()
def draw_string(x,y, s: str):
    ax.text(x,y,s, ha="center", va="center", color=cc, zorder=3)

def dessine_formule(t: Formule):
    graph_dx, graph_dy, rayon = 0.35, 1.5, 0.45
    def ecart(m: Formule) -> int:
        if   isinstance(m.node, Non): return 1 + ecart(m.gauche)
        elif isinstance(m.node, Binaire): return ecart(m.gauche) + ecart(m.droit)
        else: return 1
    def dessine_noeud(x,y, t: Formule):
        if isinstance(t.node, (Vrai, Faux) ):
            lineto(x,y);
            set_color("white"); fill_circle(x,y,rayon); set_color("black")
            draw_string(x,y, str(t.node.value))
        elif isinstance(t.node, Variable):
            lineto(x,y);
            set_color("white"); fill_circle(x,y,rayon); set_color("black")
            draw_string(x,y, str(t.node.value))

        elif isinstance(t.node, Non):
            lineto(x,y);
            dessine_noeud( x, y-graph_dy, t.gauche)
            set_color("white"); fill_circle(x,y,rayon); set_color("black")
            moveto(x,y); draw_circle(x,y, rayon); draw_string(x,y, str(t.node.value))
        elif isinstance(t.node, Binaire):
            zg, zd = max(1, ecart(t.gauche)), max(1, ecart(t.droit))
            lineto(x,y); dessine_noeud(x - zd*graph_dx, y - graph_dy, t.gauche)
            moveto(x,y); dessine_noeud(x + zg*graph_dx, y - graph_dy, t.droit)
            set_color( "white"); fill_circle(x,y,rayon); set_color("black")
            moveto(x,y); draw_circle(x,y,rayon); draw_string(x,y, str(t.node.value))
        else: raise Exception("Unknown")

    x,y = 100, 100
    moveto(x,y)
    dessine_noeud(x,y,t)
```

5.4 Méthode de Beth

```
def ajoute(v: str, valeur: int, dico: dict) -> dict:
    """add (v,valeur=0/1) to dico if not exist, raise "Echec" if exist key with different value."""
    try:
        if dico[v] == valeur: return dico
        else: raise ValueError("Echec")
    except KeyError:
        dico[v] = valeur
        return dico

def __beth(i: dict, lg: list, ld: list) -> dict:
    """Where lg and ld are lists of Formule. Must not be directly called without management of error!"""
    if (lg == []) and (ld == []): return i

    elif lg == []:
        n, f1, f2, r = ld[0].node, ld[0].gauche, ld[0].droit, ld[1:]

        if   isinstance(n, Variable): return __beth( ajoute( n.value, False, i), [], r )
        elif isinstance(n, Vrai):     raise ValueError("Echec")
        elif isinstance(n, Faux):     return __beth( i, [], r )
        elif isinstance(n, Non):      return __beth( i, [f1], r )
        elif isinstance(n, Et):
            try:                      i0 = dict(i); return __beth( i, [], [f1]+r )        # (1)
            except ValueError:        return __beth( i0, [], [f2]+r )
        elif isinstance(n, Ou):       return __beth( i, [], [f1, f2]+r )
        elif isinstance(n, Imp):      return __beth( i, [f1], [f2]+r )
        elif isinstance(n, Equ):
            try:                      i0 = dict(i); return __beth( i, [f1], [f2]+r )      # (1)
            except ValueError:        return __beth( i0, [f2], [f1]+r )
    else:
        n, f1, f2, r = lg[0].node, lg[0].gauche, lg[0].droit, lg[1:]

        if   isinstance(n, Variable): return __beth( ajoute( n.value, True, i), r, ld )
        elif isinstance(n, Vrai):     return __beth( i, r, ld )
        elif isinstance(n, Faux):     raise ValueError("Echec")
        elif isinstance(n, Non):      return __beth( i, r, [f1]+ld )
        elif isinstance(n, Et):       return __beth( i, [f1,f2]+r, ld )
        elif isinstance(n, Ou):
            try:                      i0 = dict(i); return __beth( i, [f1]+r, ld )        # (1)
            except ValueError:        return __beth( i0, [f2]+r, ld )
        elif isinstance(n, Imp):
            try:                      i0 = dict(i); return __beth( i, r, [f1]+ld )        # (1)
            except ValueError:        return __beth( i0, [f1,f2]+r, ld )
        elif isinstance(n, Equ):
            try:                      i0 = dict(i); return __beth( i, [f1,f2]+r, ld )     # (1)
            except ValueError:        return __beth( i0, r, [f1,f2]+ld )
```

(1) : en cas d'Echec, i peut avoir été altéré et on repartira avec le contenu précédent i0.

```
def analyse(s: str):
    """string to formule to Beth and Beth."""
    try:
        f = formule_of_string(s)
        try:
            i_satisfie = __beth( {}, [f],[] )
            try:
                i_falsifie = __beth( {}, [], [f] )
                print(s + " est contingente")
                print("Voici une interprétation satisfiante: ", i_satisfie)
                print("Voici une interprétation falsifiante: ", i_falsifie)
            except: print(s + " est une tautologie")
        except: print(s + " est une antilogie" )
    except: print( s + " ne représente pas une formule correcte")
```

Expression d'un arbre formule, sous forme de chaine quasi-ELTP :

```
def string_of_formule(f: Formule) -> str:
    """Construction d"une chaîne expression logique ELQTP."""
    def chn(a):
        if a is None: s = ""
        else:
            e = a.node; w = str(e.value)
            if isinstance(e, (Vrai, Faux, Variable) ): s = w
            elif isinstance(e, Non): s = w + chn(a.gauche)
            elif isinstance(e, Binaire): s = "(" + chn(a.gauche) + w + chn(a.droit) + ")"
            else: raise Exception("Invalid Lexeme in tree")
        return s
    return chn(f)
```

On peut aussi introduire certaines simplifications des arbres Formule, par exemple :

```
def reduit_unaire(f: Formule) -> Formule:
    """renvoie une formule où les "Non" en cascade ont été réduits"""
    if f is not None:
        if isinstance(f.node, Non):
            u = reduit_unaire(f.gauche)
            if isinstance(u.node, Non): return u.gauche
            elif isinstance(u.node, Vrai): return Formule( Faux() )
            elif isinstance(u.node, Faux): return Formule( Vrai() )
            else: f.gauche = u; return f
        elif isinstance(f.node, Binaire):
            f.gauche = reduit_unaire(f.gauche)
            f.droit  = reduit_unaire(f.droit)
            return f
        else: return f
    else: return f
```

On pourrait aussi réduire les "(a Et 1)", "(1 Et a)", "(0 => a)", ...

5.5 Exemples

```
if __name__ == "__main__":

    s = " ( ((a et Non ~ c) => (b oud ) )  => ( 1 . ~~~~0 ) ) "
    print("s = ", s)
    f0 = formule_of_string(s)
    print( string_of_formule(f0) )
    f = reduit_unaire(f0)
    print( string_of_formule(f) )
    analyse(s)
    dessine_formule(f)
    plt.draw()                  # draw tout seul n"affiche pas! draw + wait -> affichage !
    print("Click inside figure to continue..")
    plt.waitforbuttonpress()  # cf doc: .. it throws a deprecated warning ..

    s = "(((a ou Non b) <=> ((a et d) => (((b ou c) ou (e ou f)) ou (g et h))))" +\
        " => ((a ou Non b) <=> ((a . d) => (b ou c))))"
    print("s = ", s)
    f = reduit_unaire( formule_of_string(s)  )
    print( string_of_formule(f) )
    analyse(s)
    plt.cla()                   # clear, mais il faut re-supprimer axis et grid
    plt.axis("off")             # "off": supp axis, grid. Sinon: auto-tracés
    dessine_formule(f)
    plt.draw()
    print("Click inside figure to continue..")
    plt.waitforbuttonpress()

#=== Fin ===
```