Mots_bien_formes

May 21, 2019

1 Table of Contents

1 Texte d'oral de modélisation - Agrégation Option Informatique
1.1 Préparation à l'agrégation - ENS de Rennes, 2016-17
1.2 À propos de ce document
1.3 Question de programmation
1.3.1 Théorème 3
1.3.2 Choix d'implémentation
1.4 Réponse à l'exercice requis
1.4.1 Symboles et arités
1.4.2 Vérification de l'écriture préfixe
1.4.3 Vérification et localisation de la première erreur
1.4.4 Exemples
1.5 Bonus : évaluation d'un terme
1.5.1 Manipulation basique sur une pile
1.5.2 Interprétation des symboles
1.5.3 Évaluation d'un terme
1.5.4 Un exemple d'évaluation d'un terme du calcul propositionnel
1.5.5 Avec une autre interprétation
1.6 Bonus : un autre exemple
1.6.1 Symboles et arités
1.6.2 Formules de l'arithmétique de Presburger
1.6.3 Quelques exemples de formules de Presburger
1.6.4 Vérification de l'écriture préfixe pour des formules de Presburger
1.7 Conclusion

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2016-17

- *Date* : 09 mai 2017
- *Auteur* : Lilian Besson
- *Texte*: Annale 2012, "Mots bien formés"

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un texte d'annale de l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [1]: Sys.command "ocaml -version";;

The OCaml toplevel, version 4.04.2


Out[1]: - : int = 0
```

2.3 Question de programmation

La question de programmation pour ce texte était donnée à la fin, en page 7 :

> ń Programmer la reconnaissance des mots bien formés en écriture préfixe sur la signature de l'exemple du texte à l'aide du critère fourni par le théorème 3 page 5. Il est conseillé de représenter le mot à valider sous forme d'un tableau ou d'une liste de caractères. ż

Mathématiquement, l'énoncé à implémenter sous forme d'un critère est le suivant :

2.3.1 Théorème 3

> Pour que la suite de symboles $s_1, \dots, s_n \in \Omega$ soit l'écriture *préfixe* d'un terme, il faut et il suffit que les $h_p := \sum_{i=1}^{p} (1 - \alpha(s_i))$ vérifient :
>
> $$h_1, \dots, h_{n-1} \leq 0 \text{ et } h_n = 1.$$

2.3.2 Choix d'implémentation

- Ce critère numérique va être très simple à implémenter.
- Le choix des structures de données à utiliser n'est cependant pas évident.
  - Le sujet suggérait d'utiliser un tableau ou une liste de caractères pour représenter la "suite de symboles" $s_1, \dots, s_n \in \Omega$
  - On va définir un type formel pour l'alphabet Ω, avec l'exemple du texte, $\Omega = \{V, F, \text{Non}, \text{Ou}, \text{Et}, \text{ITE}\}$.
  - On doit ensuite représenter l'arité, sous forme de tableau, table d'association ou de hashage, ou une fonction, qui permette d'associer un entier $\alpha(s)$ à chaque symbole s. Par simplicité, on choisit d'utiliser une fonction : $\alpha : \Omega \to \mathbb{N}, s \mapsto \alpha(s)$.
  - Enfin, la fonction demandée sera récursive, et très rapide à écrire.

2.4 Réponse à l'exercice requis

2.4.1 Symboles et arités

On définit : - les symboles du calcul propositionnel par un type énumération (abstrait) :

$$\Omega = \{V, F, \text{Non}, \text{Ou}, \text{Et}, \text{ITE}\}$$

ITE correspond au "If then else" ternaire (`a ? b : c` en C, `if a then b else c` en OCaml, ou `b if a else c` en Python). - Et les formules du calcul propositionnel comme une *liste* de symboles.

```
In [2]: type symbole_calcul_prop = F | V | Ou | Et | Non | ITE ;;

        type formule_calcul_prop = symbole_calcul_prop list ;;
```

```
Out[2]: type symbole_calcul_prop = F | V | Ou | Et | Non | ITE


Out[2]: type formule_calcul_prop = symbole_calcul_prop list
```

Quelques exemples de formules du calcul propositionnel, écrites sous forme préfixes, bien formées ou non :

- "Ou Non Ou F V Ou Non V F" $\equiv (\neg(F \vee V)) \vee ((\neg V) \vee F)$ est bien formé,
- "Ou Non F Non" est mal formé,
- "ITE V F V" un exemple de "If then else" bien formé,
- " " la formule vide.

```
In [3]: let ex_correct = [Ou; Non; Ou; F; V; Ou; Non; V; F];;
        let ex_incorrect = [Ou; Non; F; Non];;
        let ex_ite = [ITE; V; F; V];;
        let ex_vide = [];;
```

```
Out[3]: val ex_correct : symbole_calcul_prop list =
          [Ou; Non; Ou; F; V; Ou; Non; V; F]


Out[3]: val ex_incorrect : symbole_calcul_prop list = [Ou; Non; F; Non]


Out[3]: val ex_ite : symbole_calcul_prop list = [ITE; V; F; V]


Out[3]: val ex_vide : 'a list = []
```

Cette fonction donne l'arité du symbole du calcul propositionnel pris en argument.

- V et F ont arités $\alpha = 0$,
- Non a arité $\alpha = 1$,

- Et et Ou ont arités $\alpha = 2$,
- ITE a arité $\alpha = 3$.

Autrement dit, avec les notations du texte :

$$S = \Omega = S_0 \sqcup S_1 \sqcup S_2 \sqcup S_3 \begin{cases} S_0 &= \{V, F\}, \\ S_1 &= \{\text{Non}\}, \\ S_2 &= \{\text{Et}, \text{Ou}\}, \\ S_3 &= \{\text{ITE}\}. \end{cases}$$

Notez qu'on peut supposer que l'appel à cette fonction d'arité se fait en $\mathcal{O}(1)$ dans les calculs de complexité, puisqu'après compilation, cette fonction sera une simple lecture d'un tableau.

```
In [4]: let arite_calcul_prop (s : symbole_calcul_prop) : int =
            match s with
            | F -> 0
            | V -> 0
            | Non -> 1
            | Ou -> 2
            | Et -> 2
            | ITE -> 3
        ;;
```

```
Out[4]: val arite_calcul_prop : symbole_calcul_prop -> int = <fun>
```

2.4.2 Vérification de l'écriture préfixe

Cette fonction prend en argument une liste de symboles `l` et une fonction d'arité sur ces symboles et renvoie "vrai" (`true`) si `l` est l'écriture préfixe d'un terme et "faux" sinon (`false`).

Notez que le "et" en Caml est paresseux, i.e., `x && y` n'évalue pas `y` si `x` est `false`. Donc cette fonction peut s'arrêter avant d'avoir parcouru tous les symboles, dès qu'elle trouve un symbole qui contredit le critère sur les hauteurs h_p.

Et cela permet aussi d'obtenir une indication sur le premier symbole à contredire le critère, comme implémenté ensuite.

```
In [5]: let ecriture_prefixe_valide (l : 'a list) (arite : 'a -> int) : bool =
            let rec aux (l : 'a list) (h : int) : bool =
            match l with
            | [] -> false
            | [t] -> ((h + 1 - (arite t)) = 1)
            | t :: q  ->
                let h_suivant = (h + 1 - (arite t)) in
                (h_suivant <= 0) && (aux q h_suivant)
            in aux l 0
        ;;
```

```
Out[5]: val ecriture_prefixe_valide : 'a list -> ('a -> int) -> bool = <fun>
```

On vérifie tout de suite sur les 4 exemples définis ci-dessus :

```
In [6]: let _ = ecriture_prefixe_valide ex_correct   arite_calcul_prop;; (* true *)
        let _ = ecriture_prefixe_valide ex_incorrect arite_calcul_prop;; (* false *)
        let _ = ecriture_prefixe_valide ex_ite       arite_calcul_prop;; (* true *)
        let _ = ecriture_prefixe_valide ex_vide      arite_calcul_prop;; (* false *)

Out[6]: - : bool = true


Out[6]: - : bool = false


Out[6]: - : bool = true


Out[6]: - : bool = false
```

2.4.3 Vérification et localisation de la première erreur

Avec la remarque précédente, on peut écrire une fonction très similaire, mais qui donnera une indication sur la position (i.e., l'indice) du premier symbole qui fait que le mot n'est pas bien équilibré, si le mot n'est pas bien formé. Si le mot est bien formé, `None` est renvoyé.

```
In [7]: let ecriture_prefixe_valide_info (l : 'a list) (arite : 'a -> int) : int option =
            let rec aux (l : 'a list) (compteur : int) (h : int) : int option =
            match l with
            | [] -> Some compteur
            | [t] ->
                if (h + 1 - (arite t)) = 1 (* h = arite(t) *)
                then None
                else Some compteur
            | t :: q  ->
                (* h est l'accumulateur qui contient la somme des 1 - arite(t) *)
                let h_suivant = (h + 1 - (arite t)) in
                if h_suivant > 0 then
                    Some compteur
                else
                    aux q (compteur + 1) h_suivant
            in
            aux l 0 0
        ;;

Out[7]: val ecriture_prefixe_valide_info : 'a list -> ('a -> int) -> int option =
          <fun>
```

On vérifie tout de suite sur les 4 exemples définis ci-dessus :

```
In [8]: let _ = ecriture_prefixe_valide_info ex_correct   arite_calcul_prop;; (* None *)
        let _ = ecriture_prefixe_valide_info ex_incorrect arite_calcul_prop;; (* Some 3 *)
        let _ = ecriture_prefixe_valide_info ex_ite       arite_calcul_prop;; (* None *)
        let _ = ecriture_prefixe_valide_info ex_vide      arite_calcul_prop;; (* Some 0 *)

Out[8]: - : int option = None


Out[8]: - : int option = Some 3


Out[8]: - : int option = None


Out[8]: - : int option = Some 0
```

Cela permet de voir que sur la formule "Ou Non F Non", le premier symbole à poser problème est le dernier symbole. Et effectivement, le critère est vérifié jusqu'au dernier symbole Non.

2.4.4 Exemples

Avec les mêmes exemples :

```
In [9]: let _ = ecriture_prefixe_valide ex_correct   arite_calcul_prop;; (* true *)
        let _ = ecriture_prefixe_valide ex_incorrect arite_calcul_prop;; (* false *)
        let _ = ecriture_prefixe_valide ex_ite       arite_calcul_prop;; (* true *)
        let _ = ecriture_prefixe_valide ex_vide      arite_calcul_prop;; (* false *)

Out[9]: - : bool = true


Out[9]: - : bool = false


Out[9]: - : bool = true


Out[9]: - : bool = false
```

On peut aussi transformer un peu la deuxième formule pour la rendre valide.

```
In [10]: let ex_correct_2 = [Ou; Non; F; V];;
         let _ = ecriture_prefixe_valide ex_correct_2   arite_calcul_prop;; (* true *)

Out[10]: val ex_correct_2 : symbole_calcul_prop list = [Ou; Non; F; V]


Out[10]: - : bool = true
```

```
tant que possible
   (lire un symbole omega; soit k = alpha(omega) dans
    si k = 0
       alors empiler valeur(omega)
       sinon (pour i de 1 a k (depiler x ; v[i] := x);
              empiler omegabarre(v[k],...,v[1]))) ;
retourner le sommet de pile ;;
```

images/algorithme_evaluation_terme.png

2.5 Bonus : évaluation d'un terme

L'objectif est de construire la fonction d'évaluation d'un terme en écriture postfixe présentée dans le texte (Algorithme 2, page 4) :

- La pile utilisée dans l'algorithme sera implémentée par une simple liste.
- Ce que le texte appelle "valeur" et "omegabarre" sont regroupés dans une liste de couples tels que le premier élément du couple est un symbole `s` et le deuxième la fonction `f_s` permettant d'interpréter ce symbole ; on considère que les constante sont des fonctions d'arité 0. Cette fonction `f_s` prend en arguments une liste d'éléments du domaine et renvoie un élément du domaine.

2.5.1 Manipulation basique sur une pile

On a besoin de savoir dépiler plus d'une valeur, pour récupérer les k valeurs successives à donner à l'interprétation d'un symbole d'arité $k \geq 1$.

Cette fonction renvoie un couple de listes : - la liste des `k` éléments en sommet de la pile `p` de sorte que le sommet de la pile `p` se trouve en dernière position dans cette liste, - et la pile `p` une fois qu'on a dépilé ses `k` éléments du sommet.

```
In [11]: let depile (k : int) (p : 'a list) : ('a list * 'a list) =
             let rec aux (k : int) (p : 'a list)  (sommet_pile : 'a list) : ('a list * 'a list
             match k with
             | 0 -> (sommet_pile, p)
             | _ when p = [] ->
                 failwith "Liste vide"
             | i ->
                 let sommet_modif = (List.hd p) :: sommet_pile in
                 let p_modif = List.tl p in
                 aux (i - 1) p_modif sommet_modif
             in
             aux k p []
         ;;

Out[11]: val depile : int -> 'a list -> 'a list * 'a list = <fun>
```

Il est absolument crucial de faire *au moins un test* à ce moment là :

```
In [12]: depile 0 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile 1 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile 3 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile 8 [0; 1; 2; 3; 4; 5; 6; 7];;

Out[12]: - : int list * int list = ([], [0; 1; 2; 3; 4; 5; 6; 7])


Out[12]: - : int list * int list = ([0], [1; 2; 3; 4; 5; 6; 7])


Out[12]: - : int list * int list = ([2; 1; 0], [3; 4; 5; 6; 7])


Out[12]: - : int list * int list = ([7; 6; 5; 4; 3; 2; 1; 0], [])
```

Ça semble bien fonctionner. Notez que la première liste a été retournée (“renversée”), puisque les valeurs ont été empilées dans le sens inverse lors de leurs lectures.

On peut aussi proposer une implémentation alternative, moins élégante mais plus rapide à écrire, avec des tableaux, et deux appels à `Array.sub` pour découper le tableau, et `Array.of_list` et `Array.to_list` pour passer d'une liste à un tableau puis de deux tableaux à deux listes.

```
In [13]: let depile_2 (k : int) (p : 'a list) : ('a list * 'a list) =
             let pa = Array.of_list p in
             let debut = Array.sub pa 0 k in
             let fin   = Array.sub pa k ((Array.length pa) - k) in
             (List.rev (Array.to_list debut)), (Array.to_list fin)
         ;;

Out[13]: val depile_2 : int -> 'a list -> 'a list * 'a list = <fun>


In [14]: depile_2 0 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile_2 1 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile_2 3 [0; 1; 2; 3; 4; 5; 6; 7];;
         depile_2 8 [0; 1; 2; 3; 4; 5; 6; 7];;

Out[14]: - : int list * int list = ([], [0; 1; 2; 3; 4; 5; 6; 7])


Out[14]: - : int list * int list = ([0], [1; 2; 3; 4; 5; 6; 7])


Out[14]: - : int list * int list = ([2; 1; 0], [3; 4; 5; 6; 7])


Out[14]: - : int list * int list = ([7; 6; 5; 4; 3; 2; 1; 0], [])
```

2.5.2 Interprétation des symboles

Cette fonction prend en entrée une liste `interpretation` de couples (symbole, fonction interprétant ce symbole), un symbole `t` et une liste d'arguments `arg` et renvoie `f_t(arg)`, où `f_t` est la fonction associée au sympole `t` via `interpretation`.

Avec les notations mathématiques du texte, le symbole `t` est ω et `f_t` est $\overline{\omega}$ ("omegabarre" dans l'algorithme).

```
In [15]: let interprete (interpretation : ('a * ('d list -> 'd)) list) (t : 'a) (arg : 'd list)
             (List.assoc t interpretation) arg;;

Out[15]: val interprete : ('a * ('d list -> 'd)) list -> 'a -> 'd list -> 'd = <fun>
```

L'interprétation choisie consiste à évaluer la valeur de vérité d'une formule du calcul propositionnel, avec l'interprétation classique.

Notez que pour faciliter le typage, ces fonctions recoivent la pile, i.e., une *liste* d'arguments (de taille arbitraire) et s'occupent elles-mêmes de récupérer le sommet de pile et les valeurs suivantes.

Les fonctions échouent si la pile donnée n'est pas assez profonde, bien évidemment.

```
In [16]: let interp_V   _  = true;;
         let interp_F   _  = false;;
         let interp_Non l  = not (List.hd l);;
         let interp_Ou  l  = (List.hd l) || (List.hd (List.tl l));;
         let interp_Et  l  = (List.hd l) && (List.hd (List.tl l));;
         let interp_ITE l  = if (List.hd l) then (List.hd (List.tl l)) else (List.hd (List.tl

Out[16]: val interp_V : 'a -> bool = <fun>

Out[16]: val interp_F : 'a -> bool = <fun>

Out[16]: val interp_Non : bool list -> bool = <fun>

Out[16]: val interp_Ou : bool list -> bool = <fun>

Out[16]: val interp_Et : bool list -> bool = <fun>

Out[16]: val interp_ITE : bool list -> bool = <fun>
```

Ensuite, on crée cette liste d'association, qui permet d'associer à un symbole ω sa fonction $\overline{\omega}$:

```
In [17]: let interp_calcul_prop = [
             (V, interp_V); (F, interp_F);       (* Arité 0 *)
             (Non, interp_Non);                  (* Arité 1 *)
             (Ou, interp_Ou); (Et, interp_Et);   (* Arité 2 *)
             (ITE, interp_ITE)                   (* Arité 3 *)
         ];;

Out[17]: val interp_calcul_prop : (symbole_calcul_prop * (bool list -> bool)) list =
           [(V, <fun>); (F, <fun>); (Non, <fun>); (Ou, <fun>); (Et, <fun>);
            (ITE, <fun>)]
```

2.5.3 Évaluation d'un terme

Cette fonction prend une liste `l`, de symboles que l'on suppose correspondre à l'écriture postfixe *correcte* d'un terme, une fonction `arite : symbole -> entier` et une liste `interpretation` d'interprétation des symboles.

Elle renvoie le résultat de l'évaluation du terme `l` pour l'interpretation `interpretation`.

L'algorithme est annoncé correct par le théorème 2, non prouvé dans le texte (ça peut être une idée de développement à faire au tableau).

```
In [18]: let evalue (l : 'a list) (arite : 'a -> int) (interpretation : ('a * ('d list -> 'd))
             let rec aux (l : 'a list) (p : 'd list) : 'd  =
                 match l with
                 | [] -> List.hd p
                 | t :: q  ->
                     let k = arite t in
                     let (arguments, p_temp) = depile k p in
                     let valeur = interprete interpretation t arguments in
                     let p_fin = valeur :: p_temp
                 in (aux q p_fin)
             in
             aux l [];;

Out[18]: val evalue : 'a list -> ('a -> int) -> ('a * ('d list -> 'd)) list -> 'd =
           <fun>
```

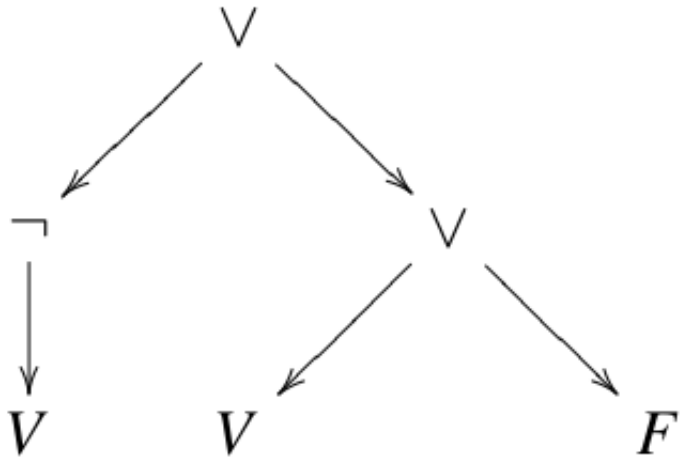
Si la lecture de l'arité d'un symbole `t`, `k = arite t`, est en $\mathcal{O}(1)$, alors l'algorithme `evalue` a une complexité en $\mathcal{O}(n)$ où n est le nombre de symbole du terme donné en entrée.

En d'autres termes, l'évaluation d'un terme postfixe par la méthode naïve, utilisée pour la notation polonaise inversée, est *linéaire* dans la taille du terme. Chouette !

2.5.4 Un exemple d'évaluation d'un terme du calcul propositionnel

On considère le terme "Ou Non V Ou V F" $= \vee\neg V \vee VF \equiv (\neg V) \vee (V \vee F)$ suivant :

```
In [19]: let ex1 = List.rev [ Ou; Non; V; Ou; V; F ];; (* écriture préfixe *)
         let _ = evalue ex1 arite_calcul_prop interp_calcul_prop;;
```



images/exemple_terme.png

```
Out[19]: val ex1 : symbole_calcul_prop list = [F; V; Ou; V; Non; Ou]


Out[19]: - : bool = true
```

On vient de passer de l'écriture préfixe à postfixe, simplement en inversant l'ordre des termes (`List.rev`).

Attention, cela ne fonctionne que si tous les symboles sont symmétriques !

Avec un autre exemple, directement écrit en postfixe, correspondant au terme suivant, qui s'interprête à "faux" (`F = false`) :

$$(F \vee \neg V) \vee (\neg(V \vee F)) \quad \underset{\text{Interpretation}}{\longrightarrow} \quad F$$

```
In [20]: let ex2 = [F; V; Ou; Non; V; Non; F; Ou; Ou];;
         let _ = evalue ex2 arite_calcul_prop interp_calcul_prop;;

Out[20]: val ex2 : symbole_calcul_prop list = [F; V; Ou; Non; V; Non; F; Ou; Ou]


Out[20]: - : bool = false
```

2.5.5 Avec une autre interprétation

On considère un second exemple : l'interprétation choisie consiste à construire **l'arbre syntaxique** d'une formule du calcul propositionnel.

```
In [21]: type arbre =
             | FeuilleV | FeuilleF                          (* Arité 0 *)
```

```
            | NNon of arbre                                  (* Arité 1 *)
            | NEt of arbre * arbre | NOu of arbre * arbre   (* Arité 2 *)
            | NITE of arbre * arbre * arbre                 (* Arité 3 *)
        ;;
```

```
Out[21]: type arbre =
             FeuilleV
           | FeuilleF
           | NNon of arbre
           | NEt of arbre * arbre
           | NOu of arbre * arbre
           | NITE of arbre * arbre * arbre
```

```
In [22]: let interp_V_a _    = FeuilleV;;
         let interp_F_a _    = FeuilleF;;
         let interp_Non_a l  = NNon (List.hd l);;
         let interp_Ou_a l   = NOu ( (List.hd l), (List.hd (List.tl l) ) );;
         let interp_Et_a l   = NEt ( (List.hd l), (List.hd (List.tl l) ) );;
         let interp_ITE_a l  = NITE ( (List.hd l), (List.hd (List.tl l) ), (List.hd (List.tl (
```

```
Out[22]: val interp_V_a : 'a -> arbre = <fun>
```

```
Out[22]: val interp_F_a : 'a -> arbre = <fun>
```

```
Out[22]: val interp_Non_a : arbre list -> arbre = <fun>
```

```
Out[22]: val interp_Ou_a : arbre list -> arbre = <fun>
```

```
Out[22]: val interp_Et_a : arbre list -> arbre = <fun>
```

```
Out[22]: val interp_ITE_a : arbre list -> arbre = <fun>
```

```
In [23]: let interp_calcul_prop_a = [
             (V, interp_V_a); (F, interp_F_a);      (* Arité 0 *)
             (Non, interp_Non_a);                   (* Arité 1 *)
             (Ou, interp_Ou_a); (Et, interp_Et_a);  (* Arité 2 *)
             (ITE, interp_ITE_a);                   (* Arité 3 *)
         ];;
```

```
Out[23]: val interp_calcul_prop_a : (symbole_calcul_prop * (arbre list -> arbre)) list =
           [(V, <fun>); (F, <fun>); (Non, <fun>); (Ou, <fun>); (Et, <fun>);
            (ITE, <fun>)]
```

```
In [24]: let _ = ex2;;
         let _ = evalue ex2 arite_calcul_prop interp_calcul_prop_a;;

Out[24]: - : symbole_calcul_prop list = [F; V; Ou; Non; V; Non; F; Ou; Ou]


Out[24]: - : arbre =
         NOu (NNon (NOu (FeuilleF, FeuilleV)), NOu (NNon FeuilleV, FeuilleF))
```

Un bonus rapide va être de jouer avec l'API du notebook Jupyter, accessible depuis OCaml via le kernel OCaml-Jupyter, pour afficher joliment le terme en LaTeX.

```
In [25]: #thread;;

/home/lilian/.opam/4.04.2/lib/ocaml/threads: added to search path
/home/lilian/.opam/4.04.2/lib/ocaml/unix.cma: loaded
/home/lilian/.opam/4.04.2/lib/ocaml/threads/threads.cma: loaded


In [26]: #require "jupyter";;
         #require "jupyter.notebook";;

/home/lilian/.opam/4.04.2/lib/easy-format: added to search path
/home/lilian/.opam/4.04.2/lib/easy-format/easy_format.cma: loaded
/home/lilian/.opam/4.04.2/lib/biniou: added to search path
/home/lilian/.opam/4.04.2/lib/biniou/biniou.cma: loaded
/home/lilian/.opam/4.04.2/lib/bytes: added to search path
/home/lilian/.opam/4.04.2/lib/result: added to search path
/home/lilian/.opam/4.04.2/lib/result/result.cma: loaded
/home/lilian/.opam/4.04.2/lib/ppx_deriving: added to search path
/home/lilian/.opam/4.04.2/lib/ppx_deriving/ppx_deriving_runtime.cma: loaded
/home/lilian/.opam/4.04.2/lib/yojson: added to search path
/home/lilian/.opam/4.04.2/lib/yojson/yojson.cma: loaded
/home/lilian/.opam/4.04.2/lib/ppx_deriving_yojson: added to search path
/home/lilian/.opam/4.04.2/lib/ppx_deriving_yojson/ppx_deriving_yojson_runtime.cma: loaded
/home/lilian/.opam/4.04.2/lib/uuidm: added to search path
/home/lilian/.opam/4.04.2/lib/uuidm/uuidm.cma: loaded
/home/lilian/.opam/4.04.2/lib/jupyter: added to search path
/home/lilian/.opam/4.04.2/lib/jupyter/jupyter.cma: loaded
/home/lilian/.opam/4.04.2/lib/base64: added to search path
/home/lilian/.opam/4.04.2/lib/base64/base64.cma: loaded
/home/lilian/.opam/4.04.2/lib/ocaml/compiler-libs: added to search path
/home/lilian/.opam/4.04.2/lib/ocaml/compiler-libs/ocamlcommon.cma: loaded
/home/lilian/.opam/4.04.2/lib/jupyter/notebook: added to search path
/home/lilian/.opam/4.04.2/lib/jupyter/notebook/jupyter_notebook.cma: loaded


In [27]: let print_latex (s : string) = Jupyter_notebook.display "text/html" ("$$" ^ s ^ "$$")
```

```
Out[27]: val print_latex : string -> Jupyter_notebook.display_id = <fun>
```

```
In [28]: print_latex "\\cos(x)";;
```

```
Out[28]: - : Jupyter_notebook.display_id = <abstr>
```

```
In [29]: let symbole_to_latex (sym : symbole_calcul_prop) =
             match sym with
             | ITE -> "\\implies"
             | Ou -> "\\vee"
             | Et -> "\\wedge"
             | Non -> "\\neg"
             | V -> "V" | F -> "F"
         ;;
```

```
Out[29]: val symbole_to_latex : symbole_calcul_prop -> string = <fun>
```

```
In [30]: let formule_to_latex (form : symbole_calcul_prop list) =
             String.concat " " (List.map symbole_to_latex form)
         ;;
```

```
Out[30]: val formule_to_latex : symbole_calcul_prop list -> string = <fun>
```

```
In [31]: print_latex (formule_to_latex ex2);;
```

```
Out[31]: - : Jupyter_notebook.display_id = <abstr>
```

Sans prendre en compte la structure d'arbre, c'est très moche !

```
In [32]: let rec arbre_to_latex (arb : arbre) =
             match arb with
             | FeuilleV -> "V"
             | FeuilleF -> "F"
             | NNon(a) -> "(\\neg " ^ (arbre_to_latex a) ^ ")"
             | NEt(a, b) -> "(" ^ (arbre_to_latex a) ^ " \\wedge " ^ (arbre_to_latex b) ^ ")"
             | NOu(a, b) -> "(" ^ (arbre_to_latex a) ^ " \\vee " ^ (arbre_to_latex b) ^ ")"
             | NITE(a, b, c) -> "(" ^ (arbre_to_latex a) ^ " ? " ^ (arbre_to_latex b) ^ " : " ^
         ;;

         let formule_to_latex2 (form : symbole_calcul_prop list) =
             let arb = evalue form arite_calcul_prop interp_calcul_prop_a in
             arbre_to_latex arb
         ;;
```

```
Out[32]: val arbre_to_latex : arbre -> string = <fun>
```

```
Out[32]: val formule_to_latex2 : symbole_calcul_prop list -> string = <fun>
```

```
In [33]: let _ = evalue ex2 arite_calcul_prop interp_calcul_prop_a
```

```
Out[33]: - : arbre =
         NOu (NNon (NOu (FeuilleF, FeuilleV)), NOu (NNon FeuilleV, FeuilleF))
```

```
In [34]: print_latex (formule_to_latex2 ex2);;
```

```
Out[34]: - : Jupyter_notebook.display_id = <abstr>
```

2.6 Bonus : un autre exemple

On va considérer ici un second exemple, celui de l'arithmétique de Presburger.
En gros, c'est l'arithmétique avec :

- des constantes (`Cst`), dans les entiers positifs $\mathbb{N}$,
- des variables (`Let`), sous formes de lettres ici (un seul `char`),
- le test d'*égalité binaire* entre variables et constantes (`Eq`), de la forme $x = y$, où x et y sont des constantes ou des variables,
- le test d'*égalité sur des sommes*, de la forme $x + y = z$, où x, y et z sont des constantes ou des variables,
- le *ou binaire* sur des formules, de la forme $\phi \vee \phi'$,
- le *et binaire* sur des formules, de la forme $\phi \wedge \phi'$,
- le *non unaire* sur une formule, de la forme $\neg\phi$,
- et le *test existenciel*, de la forme $\exists x, \phi$.

2.6.1 Symboles et arités

```
In [35]: type symbole_presburger =
             Cst | Let | Eq | PEq | O | A | N | Ex
         ;;
```

```
Out[35]: type symbole_presburger = Cst | Let | Eq | PEq | O | A | N | Ex
```

Avec ces symboles, on définit facilement leur arités.

```
In [36]: let arite_presburger (s : symbole_presburger) : int =
             match s with
             | Cst | Let       -> 0
             | N               -> 1
             | Eq | O | A | Ex -> 2
             | PEq             -> 3
         ;;
```

```
Out[36]: val arite_presburger : symbole_presburger -> int = <fun>
```

2.6.2 Formules de l'arithmétique de Presburger

Les formules suivent cette grammaire :

$$\phi, \phi' := (x = y) | (x + y = z) | \phi \vee \phi' | \phi \wedge \phi' | \neg\phi | \exists x, \phi$$

Les symboles sont donc :

- tous les entiers, et toutes les lettres d'arités 0,
- $\neg$, noté `Not`, d'arité 1,
- $\vee$, noté `Or`, $\wedge$, noté `And`, $=$, noté `Equal`, et $\exists$, noté `Exists`, d'arités 2,
- "$+=$", noté `PlusEqual`.

A noter que cet exemple nécessite des signatures non homogènes :

- $=$ et $+=$ exigent deux arguments qui soient des entiers ou des lettres,
- $\exists$ exige un premier argument qui soit une lettre, un second qui soit une formule,
- $\vee$ et $\neg$ n'acceptent que des arguments qui soient des formules.

Plutôt que de travailler avec des listes de symboles, on définit une structure arborescente pour les formules de l'arithmétique de Presburger.

```
In [37]: type entier = int ;;
         type lettre = char ;;
         type cst = I of entier | L of lettre ;;

Out[37]: type entier = int


Out[37]: type lettre = char


Out[37]: type cst = I of entier | L of lettre


In [38]: type formule_presburger =
             | Equal of cst * cst
             | PlusEqual of cst * cst * cst
             | Or of formule_presburger * formule_presburger
             | And of formule_presburger * formule_presburger
             | Not of formule_presburger
             | Exists of cst * formule_presburger

Out[38]: type formule_presburger =
             Equal of cst * cst
           | PlusEqual of cst * cst * cst
           | Or of formule_presburger * formule_presburger
           | And of formule_presburger * formule_presburger
           | Not of formule_presburger
           | Exists of cst * formule_presburger
```

Ces formules peuvent facilement s'écrire comme un terme en notation préfixes :

```
In [39]: let rec formule_vers_symboles form =
             match form with
             | Equal(L(_), L(_)) -> [Eq; Let; Let]
             | Equal(I(_), L(_)) -> [Eq; Cst; Let]
             | Equal(L(_), I(_)) -> [Eq; Let; Cst]
             | Equal(I(_), I(_)) -> [Eq; Cst; Cst]
             | PlusEqual(L(_), L(_), L(_)) -> [PEq; Let; Let; Let]
             | PlusEqual(I(_), L(_), L(_)) -> [PEq; Cst; Let; Let]
             | PlusEqual(L(_), I(_), L(_)) -> [PEq; Let; Cst; Let]
             | PlusEqual(I(_), I(_), L(_)) -> [PEq; Cst; Cst; Let]
             | PlusEqual(L(_), L(_), I(_)) -> [PEq; Let; Let; Cst]
             | PlusEqual(I(_), L(_), I(_)) -> [PEq; Cst; Let; Cst]
             | PlusEqual(L(_), I(_), I(_)) -> [PEq; Let; Cst; Cst]
             | PlusEqual(I(_), I(_), I(_)) -> [PEq; Cst; Cst; Cst]
             | Or(a, b) -> O :: (formule_vers_symboles a) @ (formule_vers_symboles b)
             | And(a, b) -> A :: (formule_vers_symboles a) @ (formule_vers_symboles b)
             | Not(a) -> N :: (formule_vers_symboles a)
             | Exists(L(_), a) -> [Ex; Let] @ (formule_vers_symboles a)
         ;;

File "[39]", line 2, characters 4-916:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Exists (I _, _)


Out[39]: val formule_vers_symboles : formule_presburger -> symbole_presburger list =
           <fun>
```

Cet avertissement est normal, ici on impose que dans `Exists(u, a)`, `u` soit nécessairement de la forme `L(x)`, i.e., une variable et non une constante.

On peut aussi afficher une formule, en la convertissant vers une chaîne de caractère :

```
In [40]: let i = string_of_int ;;
         let c car = String.make 1 car ;;

Out[40]: val i : int -> string = <fun>


Out[40]: val c : char -> string = <fun>


In [41]: let rec formule_vers_chaine form =
             match form with
             | Equal(L(x), L(y)) -> (c x) ^ "=" ^ (c y)
```

```
            | Equal(I(x), L(y)) -> (i x) ^ "=" ^ (c y)
            | Equal(L(x), I(y)) -> (c x) ^ "=" ^ (i y)
            | Equal(I(x), I(y)) -> (i x) ^ "=" ^ (i y)
            | PlusEqual(L(x), L(y), L(z)) -> (c x) ^ "+" ^ (c y) ^ "=" ^ (c z)
            | PlusEqual(I(x), L(y), L(z)) -> (i x) ^ "+" ^ (c y) ^ "=" ^ (c z)
            | PlusEqual(L(x), I(y), L(z)) -> (c x) ^ "+" ^ (i y) ^ "=" ^ (c z)
            | PlusEqual(I(x), I(y), L(z)) -> (i x) ^ "+" ^ (i y) ^ "=" ^ (c z)
            | PlusEqual(L(x), L(y), I(z)) -> (c x) ^ "+" ^ (c y) ^ "=" ^ (i z)
            | PlusEqual(I(x), L(y), I(z)) -> (i x) ^ "+" ^ (c y) ^ "=" ^ (i z)
            | PlusEqual(L(x), I(y), I(z)) -> (c x) ^ "+" ^ (i y) ^ "=" ^ (i z)
            | PlusEqual(I(x), I(y), I(z)) -> (i x) ^ "+" ^ (i y) ^ "=" ^ (i z)
            | Or(a, b) -> (formule_vers_chaine a) ^ "v" ^ (formule_vers_chaine b)
            | And(a, b) -> (formule_vers_chaine a) ^ "^" ^ (formule_vers_chaine b)
            | Not(a) -> "~" ^ (formule_vers_chaine a)
            | Exists(L(x), a) -> "E" ^ (c x) ^ ", " ^ (formule_vers_chaine a)
        ;;
```

```
File "[41]", line 2, characters 4-1040:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Exists (I _, _)
```

```
Out[41]: val formule_vers_chaine : formule_presburger -> string = <fun>
```

Et en bonus, on affiche aussi avec une chaîne en LaTeX :

```
In [42]: let rec formule_vers_latex form =
            match form with
            | Equal(L(x), L(y)) -> (c x) ^ "=" ^ (c y)
            | Equal(I(x), L(y)) -> (i x) ^ "=" ^ (c y)
            | Equal(L(x), I(y)) -> (c x) ^ "=" ^ (i y)
            | Equal(I(x), I(y)) -> (i x) ^ "=" ^ (i y)
            | PlusEqual(L(x), L(y), L(z)) -> (c x) ^ "+" ^ (c y) ^ "=" ^ (c z)
            | PlusEqual(I(x), L(y), L(z)) -> (i x) ^ "+" ^ (c y) ^ "=" ^ (c z)
            | PlusEqual(L(x), I(y), L(z)) -> (c x) ^ "+" ^ (i y) ^ "=" ^ (c z)
            | PlusEqual(I(x), I(y), L(z)) -> (i x) ^ "+" ^ (i y) ^ "=" ^ (c z)
            | PlusEqual(L(x), L(y), I(z)) -> (c x) ^ "+" ^ (c y) ^ "=" ^ (i z)
            | PlusEqual(I(x), L(y), I(z)) -> (i x) ^ "+" ^ (c y) ^ "=" ^ (i z)
            | PlusEqual(L(x), I(y), I(z)) -> (c x) ^ "+" ^ (i y) ^ "=" ^ (i z)
            | PlusEqual(I(x), I(y), I(z)) -> (i x) ^ "+" ^ (i y) ^ "=" ^ (i z)
            | Or(a, b) -> (formule_vers_latex a) ^ "\\vee" ^ (formule_vers_latex b)
            | And(a, b) -> (formule_vers_latex a) ^ "\\wedge" ^ (formule_vers_latex b)
            | Not(a) -> "\\neg" ^ (formule_vers_latex a)
            | Exists(L(x), a) -> "\\exists " ^ (c x) ^ ", " ^ (formule_vers_latex a)
        ;;
```

```
File "[42]", line 2, characters 4-1056:
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
Exists (I _, _)
```

```
Out[42]: val formule_vers_latex : formule_presburger -> string = <fun>
```

2.6.3 Quelques exemples de formules de Presburger

On peut prendre quelques exemples de formules, et les convertir en liste de symboles. Notez qu'on perd l'information des constantes et des lettres !

Des formules bien formées :

- $\phi_1 = \exists x, x = 3$, (vraie).
- $\phi_2 = \exists x, \exists y, x + y = 10$, (vraie).
- $\phi_3 = \exists x, x + 1 = 0$ (fausse).

```
In [43]: let formule_1 = Exists(L('x'), Equal(L('x'), I(3)));;
         let formule_2 = Exists(L('x'), Exists(L('y'), PlusEqual(L('x'), L('y'), I(10))));;
         let formule_3 = Exists(L('x'), PlusEqual(L('x'), I(1), I(0)));;
```

```
Out[43]: val formule_1 : formule_presburger = Exists (L 'x', Equal (L 'x', I 3))
```

```
Out[43]: val formule_2 : formule_presburger =
           Exists (L 'x', Exists (L 'y', PlusEqual (L 'x', L 'y', I 10)))
```

```
Out[43]: val formule_3 : formule_presburger =
           Exists (L 'x', PlusEqual (L 'x', I 1, I 0))
```

```
In [44]: print_endline (formule_vers_chaine formule_1);;
         print_endline (formule_vers_chaine formule_2);;
         print_endline (formule_vers_chaine formule_3);;
```

```
Ex, x=3
```

```
Out[44]: - : unit = ()
```

```
Ex, Ey, x+y=10
```

```
Out[44]: - : unit = ()
```

```
Ex, x+1=0
```

```
Out[44]: - : unit = ()
```

```
In [45]: print_latex (formule_vers_latex formule_1);;
         print_latex (formule_vers_latex formule_2);;
         print_latex (formule_vers_latex formule_3);;
```

```
Out[45]: - : Jupyter_notebook.display_id = <abstr>
```

```
Out[45]: - : Jupyter_notebook.display_id = <abstr>
```

```
Out[45]: - : Jupyter_notebook.display_id = <abstr>
```

2.6.4 Vérification de l'écriture préfixe pour des formules de Presburger

```
In [46]: let sy1 = formule_vers_symboles formule_1;;
         let sy2 = formule_vers_symboles formule_2;;
         let sy3 = formule_vers_symboles formule_3;;
```

```
Out[46]: val sy1 : symbole_presburger list = [Ex; Let; Eq; Let; Cst]
```

```
Out[46]: val sy2 : symbole_presburger list = [Ex; Let; Ex; Let; PEq; Let; Let; Cst]
```

```
Out[46]: val sy3 : symbole_presburger list = [Ex; Let; PEq; Let; Cst; Cst]
```

Elles sont évidemment bien formées.

```
In [47]: let _ = ecriture_prefixe_valide sy1 arite_presburger;; (* true *)
         let _ = ecriture_prefixe_valide sy2 arite_presburger;; (* true *)
         let _ = ecriture_prefixe_valide sy3 arite_presburger;; (* true *)
```

```
Out[47]: - : bool = true
```

```
Out[47]: - : bool = true
```

```
Out[47]: - : bool = true
```

On peut regarder d'autres suites de symboles qui ne sont pas valides.

```
In [48]: let sy4 = [Ex; Let; Eq; Let; Eq];;
         let sy5 = [Ex; Let; Ex; Let; Eq; Let; Let; Cst];;
         let sy6 = [Ex; Let; PEq; Let; Eq; Cst];;
```

```
Out[48]: val sy4 : symbole_presburger list = [Ex; Let; Eq; Let; Eq]


Out[48]: val sy5 : symbole_presburger list = [Ex; Let; Ex; Let; Eq; Let; Let; Cst]


Out[48]: val sy6 : symbole_presburger list = [Ex; Let; PEq; Let; Eq; Cst]


In [49]: let _ = ecriture_prefixe_valide_info sy4 arite_presburger;; (* Some 4 *)
         let _ = ecriture_prefixe_valide_info sy5 arite_presburger;; (* Some 6 *)
         let _ = ecriture_prefixe_valide_info sy6 arite_presburger;; (* Some 5 *)

Out[49]: - : int option = Some 4


Out[49]: - : int option = Some 6


Out[49]: - : int option = Some 5
```

Ça suffit pour cet exemple, on voulait juste montrer une autre utilisation de cette fonction `ecriture_prefixe_valide`.

Il serait difficile d'interpréter ces termes, par contre, à cause du prédicat $\exists$…

Je n'ai pas essayé d'en faire plus ici, inutile.

2.7 Conclusion

Voilà pour la question obligatoire de programmation :

- on a préféré être prudent, en testant avec l'exemple du texte (calcul propositionnel) mais on a essayé un autre exemple,
- on a fait des exemples et *on les garde* dans ce qu'on présente au jury.

Et on a essayé de faire *un peu plus*, en implémentant l'algorithme d'évaluation des termes.

Bien-sûr, ce petit notebook ne se prétend pas être une solution optimale, ni exhaustive.

Merci à Aude et Vlad pour leur implémentation, sur laquelle ce document est principalement basé.