public2018-D1

December 6, 2017

1 Table of Contents

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2017-18

- *Date* : 6 décembre 2017
- *Auteur* : Lilian Besson
- *Texte*: Annale 2018, "Expressions arithmétiques" (public2018-D1)

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";;
        print_endline Sys.ocaml_version;;

The OCaml toplevel, version 4.04.2


Out[1]: - : int = 0


4.04.2


Out[1]: - : unit = ()
```

2.3 Question de programmation

La question de programmation pour ce texte était donnée en fin de page 6 :

2.3.1 Modélisation

On est libre de choisir l'implémentation qui nous convient pour les expressions arithmétiques sous forme arborescente.

Je choisis de ne considérer que les variables et pas les valeurs (on aura des expressions en OCaml comme `F("x")` pour la variable x), et uniquement des arbres binaires. Les noeuds `N(t1, op, t2)` sont étiquetées par un opérateur binaire `op`, dont on fournit à l'avance une liste fixée et finie, et les feuilles `F(s)` sont étiquetées par une variable `s`.

2.3.2 Exercice

> ń Écrire une fonction qui reçoit en argument une expression algébrique donnée sous forme arborescente et décore cette expression en calculant pour chaque nud interne quelle est la valeur du paramètre et quelle branche doit être évaluée en premier selon l'algorithme d'Ershov. ż

2.4 Solution

On va essayer d'être rapide et de faire simple, donc on choisit une algèbre de termes particulière, mais l'algorithme d'Ershov sera implémenté de façon générique (polymorphique, peu importe la valeur de `op`).

2.4.1 Types et représentations

```
In [2]: type operateur = Plus | Moins | MoinsDroite | Mul | Div | DivDroite | Modulo | Expo ;;
        (* On utilisera MoinsDroite et DivDroite pour la compilation avec la méthode d'Ershov

Out[2]: type operateur =
            Plus
          | Moins
          | MoinsDroite
          | Mul
          | Div
          | DivDroite
          | Modulo
          | Expo


In [3]: type ('a, 'b) arbre_binaire = N of (('a,'b) arbre_binaire) * 'b * (('a,'b) arbre_binai

Out[3]: type ('a, 'b) arbre_binaire =
            N of ('a, 'b) arbre_binaire * 'b * ('a, 'b) arbre_binaire
          | F of 'a
```

Par exemple pour l'expression $\frac{x-yz}{u-vw}$, c'est-à-dire `(x - y*z)/(u - v*w)` :

```
In [4]: (* exp1 = (x - y*z) *)
        let exp1 =
        N(
            F("x"),
            Moins,
            N(
                F("y"),
                Mul,
                F("z")
            )
        )
        ;;

Out[4]: val exp1 : (string, operateur) arbre_binaire =
          N (F "x", Moins, N (F "y", Mul, F "z"))


In [5]: (* exp2 = (u - v*w) *)
        let exp2 =
        N(
            F("u"),
            Moins,
            N(
                F("v"),
```

```
            Mul,
            F("w")
        )
    )
    ;;
```

```
Out[5]: val exp2 : (string, operateur) arbre_binaire =
          N (F "u", Moins, N (F "v", Mul, F "w"))
```

```
In [6]: (* exp3 = (x - y*z)/(u - v*w) *)
        let exp3 =
        N(
            exp1,
            Div,
            exp2
        )
```

```
Out[6]: val exp3 : (string, operateur) arbre_binaire =
          N (N (F "x", Moins, N (F "y", Mul, F "z")), Div,
           N (F "u", Moins, N (F "v", Mul, F "w")))
```

2.4.2 Calcul récursif du nombre ρ

C'est assez immédiat, en suivant la définition récursive : $\rho(F) = 0$ et $\rho(N(t_1, t_2)) = \rho(t_1) + 1$ si $\rho(t_1) = \rho(t_2)$ et $\max(\rho(t_1), \rho(t_2))$ si $\rho(t_1) \neq \rho(t_2)$.

```
In [7]: let rec nombre_rho (expr : ('a, 'b) arbre_binaire) : int =
            match expr with
            | F _ -> 0
            | N(t1, _, t2) ->
                let d1, d2 = nombre_rho t1, nombre_rho t2 in
                if d1 = d2 then
                    d1 + 1
                else
                    max d1 d2
        ;;
```

```
Out[7]: val nombre_rho : ('a, 'b) arbre_binaire -> int = <fun>
```

Pour comparer avec le calcul, plus simple, de la hauteur de l'arbre :

```
In [8]: let rec hauteur (expr : ('a, 'b) arbre_binaire) : int =
            match expr with
            | F _ -> 0
            | N(t1, _, t2) ->
                let d1, d2 = hauteur t1, hauteur t2 in
                1 + (max d1 d2)
        ;;
```

```
Out[8]: val hauteur : ('a, 'b) arbre_binaire -> int = <fun>
```

Exemples qui concordent avec le texte :

```
In [9]: let _ = hauteur exp1;;
        let _ = nombre_rho exp1;;

Out[9]: - : int = 2


Out[9]: - : int = 1


In [10]: let _ = hauteur exp2;;
         let _ = nombre_rho exp2;;

Out[10]: - : int = 2


Out[10]: - : int = 1


In [11]: let _ = hauteur exp3;;
         let _ = nombre_rho exp3;;

Out[11]: - : int = 3


Out[11]: - : int = 2
```

2.4.3 Algorithme demandé pour décorer l'arbre

On choisit d'ajouter une *décoration* de type `'c` :

```
In [12]: type ('a, 'b, 'c) arbre_binaire_decore = N2 of ('c * (('a, 'b, 'c) arbre_binaire_deco

Out[12]: type ('a, 'b, 'c) arbre_binaire_decore =
             N2 of
               ('c * ('a, 'b, 'c) arbre_binaire_decore * 'b *
                ('a, 'b, 'c) arbre_binaire_decore)
           | F2 of 'a
```

On a besoin d'attacher à chaque noeud son paramètre ρ et un drapeau binaire permettant de savoir si l'algorithme d'Ershov indique d'évaluer en premier le sous-arbre gauche (`premier_gauche = true`) ou droite (`= false`).

```
In [13]: type decoration = {
             rho : int;
             premier_gauche : bool;
         };;

Out[13]: type decoration = { rho : int; premier_gauche : bool; }
```

```
In [14]: let rec decore (expr : (('a, 'b) arbre_binaire)) : (('a, 'b, decoration) arbre_binaire
             match expr with
             | F v -> F2 v
             | N (t1, o, t2) ->
                 let d1, d2 = nombre_rho t1, nombre_rho t2 in
                 let d = if d1 = d2 then d1 + 1 else max d1 d2 in
                 N2({rho = d; premier_gauche = (d2<= d1)}, (decore t1), o, (decore t2))
         ;;

Out[14]: val decore :
           ('a, 'b) arbre_binaire -> ('a, 'b, decoration) arbre_binaire_decore = <fun>
```

Dans nos exemples, on voit que l'évaluation favorise en premier (avec des `premier_gauche = false`) les expressions les plus profondes (à droite) au sens du paramètre ρ :

```
In [15]: decore exp1;;

Out[15]: - : (string, operateur, decoration) arbre_binaire_decore =
         N2
          ({rho = 1; premier_gauche = false}, F2 "x", Moins,
           N2 ({rho = 1; premier_gauche = true}, F2 "y", Mul, F2 "z"))
```

```
In [16]: decore exp2;;

Out[16]: - : (string, operateur, decoration) arbre_binaire_decore =
         N2
          ({rho = 1; premier_gauche = false}, F2 "u", Moins,
           N2 ({rho = 1; premier_gauche = true}, F2 "v", Mul, F2 "w"))
```

```
In [17]: decore exp3;;

Out[17]: - : (string, operateur, decoration) arbre_binaire_decore =
         N2
          ({rho = 2; premier_gauche = true},
           N2
            ({rho = 1; premier_gauche = false}, F2 "x", Moins,
             N2 ({rho = 1; premier_gauche = true}, F2 "y", Mul, F2 "z")),
           Div,
           N2
            ({rho = 1; premier_gauche = false}, F2 "u", Moins,
             N2 ({rho = 1; premier_gauche = true}, F2 "v", Mul, F2 "w")))
```

2.5 Complexités

2.5.1 En espace

Les deux fonctions présentées ci-dessus n'utilisent pas d'autre espace que l'arbre décoré, et la pile d'appel récursif.

- Le calcul récursif de $\rho(t)$ prend donc un espace proportionnel au nombre d'appel récursif imbriqué, qui est borné par la taille du terme t (définie comme le nombre de noeuds et de feuilles), donc est **linéaire**,
- Le calcul de la méthode d'Ershov est elle aussi **linéaire** puisque l'arbre décoré est de même taille que l'arbre initial.

2.5.2 En temps

Les deux fonctions présentées ci-dessus sont **linéaires** dans la taille de l'arbre.

2.6 Implémentations supplémentaires

On peut essayer d'implémenter une fonction qui afficherait ceci pour la méthode d'évaluation naturelle :

Et ceci pour la méthode d'Ershov :

Ce n'est pas trop difficile, mais ça prend un peu de temps :

- on va d'abord montrer comment évaluer les expressions, en lisant l'arbre et en effectuant des appels récursifs,
- puis on fera un parcours postfix de l'arbre afin d'utiliser l'évaluation avec une pile, avec la stratégie naïve,
- et enfin la méthode d'Ershov permettra de réduire la hauteur maximale de la pile, en passant de $h(t)$ (hauteur de l'arbre) à $\rho(t)$,
- en bonus, on affichera les instructions style "compilateur à registre", pour visualiser le gain apporté par la méthode d'Ershov.

  Bien sûr, tout cela fait beaucoup trop pour être envisagé le jour de l'oral ! Mais un des points aurait pû être implémenté rapidement.

2.6.1 Évaluation des expressions

Un premier objectif plus simple est d'évaluer les expressions, en fournissant un *contexte* (table associant une valeur à chaque variable).

```
In [18]: type ('a, 'b) contexte = ('a * 'b) list;;
         let valeur (ctx : ('a, 'b) contexte) (var : 'a) = List.assoc var ctx;;
         (* une Hashtbl peut etre utilisee si besoin de bonnes performances *)

Out[18]: type ('a, 'b) contexte = ('a * 'b) list


Out[18]: val valeur : ('a, 'b) contexte -> 'a -> 'b = <fun>
```

```
In [19]: let contexte1 : (string, int) contexte = [
             ("x", 1); ("y", 2); ("z", 3);
             ("u", 4); ("v", 5); ("w", 6)
         ];;

Out[19]: val contexte1 : (string, int) contexte =
           [("x", 1); ("y", 2); ("z", 3); ("u", 4); ("v", 5); ("w", 6)]


In [20]: let intop_of_op (op : operateur) : (int -> int -> int) =
             match op with
             | Plus -> ( + )
             | Moins -> ( - )
             | MoinsDroite -> (fun v1 -> fun v2 -> v2 - v1)
             | Mul -> ( * )
             | Div -> ( / )
             | DivDroite -> (fun v1 -> fun v2 -> v2 / v1)
             | Modulo -> ( mod )
             | Expo ->
                 (fun v1 -> fun v2 -> int_of_float ((float_of_int v1) ** (float_of_int v2)))
         ;;

Out[20]: val intop_of_op : operateur -> int -> int -> int = <fun>


In [21]: let rec eval_int (ctx : (string, int) contexte) (expr : (string, operateur) arbre_bina
             match expr with
             | F(s) -> valeur ctx s
             | N(t1, op, t2) ->
                 let v1, v2 = eval_int ctx t1, eval_int ctx t2 in
                 (intop_of_op op) v1 v2
         ;;

Out[21]: val eval_int :
           (string, int) contexte -> (string, operateur) arbre_binaire -> int = <fun>
```

Par exemple, x vaut 1 dans le contexte d'exemple, et $x + y$ vaut $1 + 2 = 3$:

```
In [22]: let _ = eval_int contexte1 (F("x"));;

Out[22]: - : int = 1


In [23]: let _ = eval_int contexte1 (N(F("x"), Plus, F("y")));;

Out[23]: - : int = 3


In [24]: let _ = eval_int contexte1 exp1;;
```

```
Out[24]: - : int = -5
```

```
In [25]: let _ = eval_int contexte1 exp2;;

Out[25]: - : int = -26
```

```
In [26]: let _ = eval_int contexte1 exp3;;

Out[26]: - : int = 0
```

On voit la faiblesse de l'interprétation avec des entiers, la division / est une division entière !
On peut aussi interpréter avec des flottants :

```
In [27]: let contexte2 : (string, float) contexte = [
             ("x", 1.); ("y", 2.); ("z", 3.);
             ("u", 4.); ("v", 5.); ("w", 6.)
         ];;

Out[27]: val contexte2 : (string, float) contexte =
           [("x", 1.); ("y", 2.); ("z", 3.); ("u", 4.); ("v", 5.); ("w", 6.)]
```

```
In [28]: let floatop_of_op (op : operateur) : (float -> float -> float) =
             match op with
             | Plus -> ( +. )
             | Moins -> ( -. )
             | MoinsDroite -> (fun v1 -> fun v2 -> v2 -. v1)
             | Mul -> ( *. )
             | Div -> ( /. )
             | DivDroite -> (fun v1 -> fun v2 -> v2 /. v1)
             | Modulo ->
                 (fun v1 -> fun v2 -> float_of_int ((int_of_float v1) mod (int_of_float v2)))
             | Expo -> ( ** )
         ;;

Out[28]: val floatop_of_op : operateur -> float -> float -> float = <fun>
```

```
In [29]: let rec eval_float (ctx : (string, float) contexte) (expr : (string, operateur) arbre
             match expr with
             | F(s) -> valeur ctx s
             | N(t1, op, t2) ->
                 let v1, v2 = eval_float ctx t1, eval_float ctx t2 in
                 (floatop_of_op op) v1 v2
         ;;
```

```
Out[29]: val eval_float :
           (string, float) contexte -> (string, operateur) arbre_binaire -> float =
           <fun>
```

Par exemple, x vaut 1 dans le contexte d'exemple, et $x + y$ vaut $1 + 2 = 3$:

```
In [30]: let _ = eval_float contexte2 (F("x"));;

Out[30]: - : float = 1.


In [31]: let _ = eval_float contexte2 (N(F("x"), Plus, F("y")));;

Out[31]: - : float = 3.


In [32]: let _ = eval_float contexte2 exp1;;

Out[32]: - : float = -5.


In [33]: let _ = eval_float contexte2 exp2;;

Out[33]: - : float = -26.


In [34]: let _ = eval_float contexte2 exp3;;

Out[34]: - : float = 0.192307692307692318
```

2.6.2 Evaluation par lecture postfix et pile

On va commencer par lire l'arbre en parcours postfix (cf. TP2 @ ENS Rennes 2017/18) et ensuite l'évaluer grâce à une pile.

```
In [35]: type ('a, 'b) lexem = O of 'b | V of 'a;;
         type ('a, 'b) parcours = (('a, 'b) lexem) list;;

Out[35]: type ('a, 'b) lexem = O of 'b | V of 'a


Out[35]: type ('a, 'b) parcours = ('a, 'b) lexem list


In [36]: let parcours_postfix (expr : ('a, 'b) arbre_binaire) : (('a, 'b) parcours) =
             let rec parcours vus expr =
                 match expr with
                 | F(s) -> V(s) :: vus
                 | N(t1, op, t2) -> O(op) :: (parcours (parcours vus t1) t2)
             in
             List.rev (parcours [] expr)
         ;;
```

```
Out[36]: val parcours_postfix : ('a, 'b) arbre_binaire -> ('a, 'b) parcours = <fun>
```

On le teste :

```
In [37]: parcours_postfix exp1;;

Out[37]: - : (string, operateur) parcours = [V "x"; V "y"; V "z"; O Mul; O Moins]
```

```
In [38]: parcours_postfix exp3;;

Out[38]: - : (string, operateur) parcours =
         [V "x"; V "y"; V "z"; O Mul; O Moins; V "u"; V "v"; V "w"; O Mul; O Moins;
          O Div]
```

```
In [39]: let eval_int_2 (ctx : (string, int) contexte) (expr : (string, operateur) arbre_binai
             let vus = parcours_postfix expr in
             let pile = Stack.create () in
             let aux lex =
                 match lex with
                 | V(s) -> Stack.push (valeur ctx s) pile;
                 | O(op) ->
                     let v1, v2 = Stack.pop pile, Stack.pop pile in
                     Stack.push ((intop_of_op op) v1 v2) pile;
             in
             List.iter aux vus;
             Stack.pop pile
         ;;

Out[39]: val eval_int_2 :
           (string, int) contexte -> (string, operateur) arbre_binaire -> int = <fun>
```

Par exemple, $x - y * z$ avec $x = 1, y = 2, z = 3$ vaut $1 - 2 * 3 = -5$:

```
In [40]: let _ = exp1 ;;
         let _ = eval_int_2 contexte1 exp1;;

Out[40]: - : (string, operateur) arbre_binaire =
         N (F "x", Moins, N (F "y", Mul, F "z"))
```

```
Out[40]: - : int = -5
```

```
In [41]: let _ = exp2;;
         let _ = eval_int_2 contexte1 exp2;;
```

```
Out[41]: - : (string, operateur) arbre_binaire =
         N (F "u", Moins, N (F "v", Mul, F "w"))


Out[41]: - : int = -26
```

On peut maintenant faire la même fonction mais qui va en plus afficher l'état successif de la pile (avec des valeurs uniquement).

```
In [42]: let print f =
             let r = Printf.printf f in
             flush_all();
             r
         ;;

Out[42]: val print : ('a, out_channel, unit) format -> 'a = <fun>


In [43]: let print_pile pile =
             print "\nPile : ";
             Stack.iter (print "%i; ") pile;
             print "."
         ;;

Out[43]: val print_pile : int Stack.t -> unit = <fun>


In [44]: let eval_int_3 (ctx : (string, int) contexte) (expr : (string, operateur) arbre_binai
             let vus = parcours_postfix expr in
             let pile = Stack.create () in
             let aux lex =
                 print_pile pile;
                 match lex with
                 | V(s) -> Stack.push (valeur ctx s) pile;
                 | O(op) ->
                     let v1, v2 = Stack.pop pile, Stack.pop pile in
                     Stack.push ((intop_of_op op) v1 v2) pile;
             in
             List.iter aux vus;
             Stack.pop pile
         ;;

Out[44]: val eval_int_3 :
           (string, int) contexte -> (string, operateur) arbre_binaire -> int = <fun>


In [45]: let _ = exp1 ;;
         let _ = eval_int_3 contexte1 exp1;;
```

```
Out[45]: - : (string, operateur) arbre_binaire =
         N (F "x", Moins, N (F "y", Mul, F "z"))
```

```
Pile : .
Pile : 1; .
Pile : 2; 1; .
Pile : 3; 2; 1; .
```

```
Out[45]: - : int = -5
```

```
In [46]: let _ = exp3;;
         let _ = eval_int_3 contexte1 exp3;;
```

```
Out[46]: - : (string, operateur) arbre_binaire =
         N (N (F "x", Moins, N (F "y", Mul, F "z")), Div,
          N (F "u", Moins, N (F "v", Mul, F "w")))
```

```
Pile : 6; 1; .
Pile : .
Pile : 1; .
Pile : 2; 1; .
Pile : 3; 2; 1; .
Pile : 6; 1; .
Pile : -5; .
Pile : 4; -5; .
Pile : 5; 4; -5; .
Pile : 6; 5; 4; -5; .
Pile : 30; 4; -5; .
```

```
Out[46]: - : int = 0
```

Il y a un soucis dans l'ordre d'affichage des lignes, dû à Jupyter et pas à notre fonction.

On vérifie qu'on utilise au plus 4 valeurs sur la pile, comme représenté dans la figure de l'énoncé :

2.6.3 Affichage dans ce langage de manipulation de registre

On ne va pas trop formaliser ça, mais juste les afficher...

```
In [47]: let print_aff (line : int) (i : int) (s : string) : unit =
             print "\n%02i: R[%d] := %s ;" line i s;
         ;;
```

```
Out[47]: val print_aff : int -> int -> string -> unit = <fun>


In [48]: let string_of_op (op : operateur) : string =
             match op with
             | Plus -> "+"
             | Moins | MoinsDroite -> "-"
             | Mul -> "*"
             | Div | DivDroite -> "/"
             | Modulo -> "%"
             | Expo -> "^"
         ;;

Out[48]: val string_of_op : operateur -> string = <fun>


In [49]: let print_op (line : int) (i : int) (j : int) (k : int) (op : operateur) : unit =
             match op with
             | MoinsDroite | DivDroite -> (* on gère ici les opérateurs "inverses" *)
                 print "\n%02i: R[%d] := R[%d] %s R[%d] ;" line i k (string_of_op op) j;
             | _ ->
                 print "\n%02i: R[%d] := R[%d] %s R[%d] ;" line i j (string_of_op op) k;
         ;;

Out[49]: val print_op : int -> int -> int -> int -> operateur -> unit = <fun>


In [50]: let eval_int_4 (ctx : (string, int) contexte) (expr : (string, operateur) arbre_binai
             let vus = parcours_postfix expr in
             let pile = Stack.create () in
             let ligne = ref 0 in
             let aux lex =
                 incr ligne;
                 match lex with
                 | V(s) ->
                     Stack.push (valeur ctx s) pile;
                     print_aff !ligne ((Stack.length pile) - 1) s;
                 | O(op) ->
                     let v1, v2 = Stack.pop pile, Stack.pop pile in
                     Stack.push ((intop_of_op op) v1 v2) pile;
                     print_op !ligne ((Stack.length pile) - 1) ((Stack.length pile) - 1) (Stac
             in
             List.iter aux vus;
             Stack.pop pile
         ;;

Out[50]: val eval_int_4 :
           (string, int) contexte -> (string, operateur) arbre_binaire -> int = <fun>
```

Essayons ça :

```
In [62]: let _ = exp1 ;;
         let _ = eval_int_4 contexte1 exp1;;

Out[62]: - : (string, operateur) arbre_binaire =
         N (F "x", Moins, N (F "y", Mul, F "z"))


04: R[1] := R[1] * R[2] ;
05: R[0] := R[0] - R[1] ;
01: R[0] := x ;
02: R[1] := y ;
03: R[2] := z ;


Out[62]: - : int = -5


In [66]: let _ = exp3;;
         let _ = eval_int_4 contexte1 exp3;;

Out[66]: - : (string, operateur) arbre_binaire =
         N (N (F "x", Moins, N (F "y", Mul, F "z")), Div,
          N (F "u", Moins, N (F "v", Mul, F "w")))


10: R[1] := R[1] - R[2] ;
11: R[0] := R[0] / R[1] ;
01: R[0] := x ;
02: R[1] := y ;
03: R[2] := z ;
04: R[1] := R[1] * R[2] ;
05: R[0] := R[0] - R[1] ;
06: R[1] := u ;
07: R[2] := v ;
08: R[3] := w ;
09: R[2] := R[2] * R[3] ;


Out[66]: - : int = 0
```

2.6.4 Méthode d'Ershov

Enfin, on va générer, en plus de l'évaluation, un affichage comme celui qu'on voulait, qui montre les affectations dans les registres, et permettra de visualiser que la méthode d'Ershov utilise un registre de moins sur le terme exemple qui calcule $(x - y * z)/(u - v * w)$.

On rappelle qu'on a obtenu un arbre binaire décoré, représenté comme tel :

```
In [53]: decore exp1;;

Out[53]: - : (string, operateur, decoration) arbre_binaire_decore =
         N2
          ({rho = 1; premier_gauche = false}, F2 "x", Moins,
           N2 ({rho = 1; premier_gauche = true}, F2 "y", Mul, F2 "z"))
```

On modifie notre parcours postfix pour prendre en compte la décoration et savoir si on calcul d'abord le sous-arbre gauche ou droit.

```
In [54]: let parcours_postfix_decore (expr : ('a, 'b, decoration) arbre_binaire_decore) : (('a
             let rec parcours vus expr =
                 match expr with
                 | F2(s) -> V(s) :: vus
                 | N2(dec, t1, Moins, t2) when dec.premier_gauche = false ->
                     O(MoinsDroite) :: (parcours (parcours vus t2) t1)
                 | N2(dec, t1, MoinsDroite, t2) when dec.premier_gauche = false ->
                     O(Moins) :: (parcours (parcours vus t2) t1)
                 | N2(dec, t1, Div, t2) when dec.premier_gauche = false ->
                     O(DivDroite) :: (parcours (parcours vus t2) t1)
                 | N2(dec, t1, DivDroite, t2) when dec.premier_gauche = false ->
                     O(Div) :: (parcours (parcours vus t2) t1)
                 | N2(dec, t1, op, t2) when dec.premier_gauche = false ->
                     O(op) :: (parcours (parcours vus t2) t1)
                 | N2(_, t1, op, t2) ->
                     O(op) :: (parcours (parcours vus t1) t2)
             in
             List.rev (parcours [] expr)
         ;;

Out[54]: val parcours_postfix_decore :
           ('a, operateur, decoration) arbre_binaire_decore ->
           ('a, operateur) parcours = <fun>


In [55]: let eval_int_ershov (ctx : (string, int) contexte) (expr : (string, operateur) arbre_
             let vus = parcours_postfix_decore (decore expr) in
             let pile = Stack.create () in
             let ligne = ref 0 in
             let aux lex =
                 incr ligne;
                 match lex with
                 | V(s) ->
                     Stack.push (valeur ctx s) pile;
                     print_aff !ligne ((Stack.length pile) - 1) s;
                 | O(op) ->
                     let v1, v2 = Stack.pop pile, Stack.pop pile in
                     Stack.push ((intop_of_op op) v1 v2) pile;
```

```
                print_op !ligne ((Stack.length pile) - 1) ((Stack.length pile) - 1) (Stack
        in
        List.iter aux vus;
        Stack.pop pile
    ;;

Out[55]: val eval_int_ershov :
           (string, int) contexte -> (string, operateur) arbre_binaire -> int = <fun>
```

Essayons ça :

```
In [56]: let _ = exp1 ;;
         let _ = eval_int_ershov contexte1 exp1;;

Out[56]: - : (string, operateur) arbre_binaire =
         N (F "x", Moins, N (F "y", Mul, F "z"))


10: R[1] := R[1] - R[2] ;
11: R[0] := R[0] / R[1] ;
01: R[0] := y ;
02: R[1] := z ;
03: R[0] := R[0] * R[1] ;


Out[56]: - : int = -5


In [68]: let _ = exp3;;
         let _ = eval_int_ershov contexte1 exp3;;

Out[68]: - : (string, operateur) arbre_binaire =
         N (N (F "x", Moins, N (F "y", Mul, F "z")), Div,
          N (F "u", Moins, N (F "v", Mul, F "w")))


10: R[1] := R[2] - R[1] ;
11: R[0] := R[0] / R[1] ;
01: R[0] := y ;
02: R[1] := z ;
03: R[0] := R[0] * R[1] ;
04: R[1] := x ;
05: R[0] := R[1] - R[0] ;
06: R[1] := v ;
07: R[2] := w ;
08: R[1] := R[1] * R[2] ;
09: R[2] := u ;
```

```
Out[68]: - : int = 0
```

Et voilà, ce n'était pas trop dur !

2.7 Conclusion

Voilà pour la question obligatoire de programmation, et un petit bonus.

2.7.1 Qualités

- On a décomposé le problème en sous-fonctions (d'abord le calcul de ρ puis la méthode d'Ershov),
- On a fait des exemples et *on les garde* dans ce qu'on présente au jury,
- On a testé la fonction exigée sur de petits exemples et sur un exemple de taille réelle (venant du texte).

2.7.2 Bonus

On a fait pas mal de bonus, en interprétant les termes, d'abord via l'arbre et des appels récursifs, ensuite par une lecture postfix et une pile, qui nous a permis de vérifier l'évolution de la pile et de sa hauteur (avec le même exemple que dans le texte), et ensuite avec une espèce de "compilation" en visualisant les affectations dans ces registres. La "compilation" n'est pas réelle, on a uniquement affiché des choses, mais elle permet de vérifier que la méthode de Ershov aide effectivement à réduire le nombre de registre requis.

2.7.3 Défauts

- ?

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

  Vous auriez pu choisir de modéliser le problème avec une autre approche, n'hésitez pas à me contacter svp.

  C'est tout pour aujourd'hui les amis, allez voir ici pour d'autres corrections, et que la force soit avec vous !