

## TME – Sémantique d'un langage d'expressions fonctionnelles

*Remarque* : Les personnes qui suivent le module de Compilation Avancée sont invitées à implanter un analyseur syntaxique (avec les outils CamlLex et CamlYacc) pour les expressions et les programmes manipulés dans ce TME.

**Expressions arithmétiques simples** On considère ici les expressions arithmétiques simples, c'est à dire les expressions construites à partir d'entiers relatifs, de sommes, de différences, de produits et de quotients d'expressions arithmétiques. De plus, on autorise la présence de variables dans les expressions. Plus formellement :

- tout entier relatif  $n$  est une expression de  $L$
- si  $e_1$  et  $e_2$  sont des expressions de  $L$  alors  $e_1 + e_2$ ,  $e_1 \times e_2$ ,  $e_1 - e_2$  et  $e_1/e_2$  sont des expressions de  $L$
- tout nom de variable est une expression de  $L$  (les noms de variable sont représentés par des chaînes de caractères)

On introduit donc le type des expressions de  $L$  comme suit :

```
type arbre_eval =
  Cste of int
  | Plus of arbre_eval*arbre_eval
  | Mult of arbre_eval*arbre_eval
  | Moins of arbre_eval*arbre_eval
  | Div of arbre_eval*arbre_eval
  | Nom of string ;;
```

Pour pouvoir évaluer de telles expressions, il faut disposer de la notion d'environnement qui permet d'associer une valeur à un nom de variable. Pour le moment, les seules valeurs dont nous avons besoin pour pouvoir évaluer une expression de  $L$  sont les entiers relatifs. Toutefois, nous verrons par la suite que les expressions de  $L$  pourront prendre d'autres valeurs. On introduit donc un type somme correspondant au type des valeurs que pourront prendre les expressions de  $L$  – ce type n'a pour le moment qu'un seul constructeur mais sera enrichi par la suite.

```
type valeur = Vint of int;;
```

On définit une notion d'environnement d'évaluation permettant d'associer à certaines variables une valeur. Un environnement d'évaluation  $E$  est une liste de paires  $(x, v)$  où  $x$  est un nom de variable et  $v$  une valeur de type `valeur` (correspondant à la valeur associée à la variable  $x$ ) –  $E$  est donc de type `(string * valeur) list`. La valeur d'une variable  $x$  dans un environnement d'évaluation  $E$  sera obtenue en considérant la première occurrence de la paire  $(x, v)$  apparaissant dans la liste représentant  $E$ . Par exemple, si l'on considère l'environnement :

```
let env = [("x", (Vint 8)); ("y", (Vint 2)); ("z", (Vint 3)); ("x", (Vint 2))];;
```

on aura :

```
# valeur_de "x" env;;
- : valeur = (Vint 8)
```

1. Définir une fonction `valeur_de` qui, étant donné une variable  $x$  et un environnement d'évaluation  $E$  retourne la valeur associée à  $x$  par  $E$ . Si aucune paire  $(x, v)$  n'apparaît dans  $E$ , cette fonction lèvera l'exception définie par :

```
exception Bad_arg of string;;
```

avec la variable  $x$  pour argument.

L'évaluation des expressions de  $L$  se fait simplement :

- Tout entier relatif s'évalue en lui même.
- Pour évaluer une expression obtenue par application d'un opérateur sur deux expressions  $e_1$  et  $e_2$ , il suffit d'évaluer  $e_1$  en  $n_1$  et  $e_2$  en  $n_2$  puis d'appliquer cet opérateur sur  $n_1$  et  $n_2$  (on prendra soin de n'effectuer cette opération que lorsque  $n_1$  et  $n_2$  sont des "entiers" c'est à dire des objets de type `valeur` obtenus à partir du constructeur `Vint`). Enfin, une exception sera levée dans le cas d'une division par zéro.
- Pour évaluer une variable, il suffit, à l'aide de la fonction `valeur_de`, d'aller chercher sa valeur dans l'environnement d'évaluation.

On notera  $E(x)$  la valeur associée à la variable  $x$  dans l'environnement  $E$  et on écrira  $E \vdash e \rightsquigarrow v$  pour exprimer que l'expression  $e$  s'évalue à la valeur  $v$  dans l'environnement d'évaluation  $E$ . Avec cette notation, on utilisera des règles d'inférence pour spécifier l'évaluation des expressions. On a donc les 6 règles suivantes :

$$(R_1) : \frac{(n \in \mathbb{Z})}{E \vdash (\text{Cste } n) \rightsquigarrow (\text{Vint } n)}$$

$$(R_2) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2)}{E \vdash (\text{Plus}(e_1, e_2)) \rightsquigarrow (\text{Vint } (n_1 + n_2))}$$

$$(R_3) : \frac{}{E \vdash (\text{Nom } x) \rightsquigarrow E(x)}$$

$$(R_4) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2)}{E \vdash (\text{Moins}(e_1, e_2)) \rightsquigarrow (\text{Vint } (n_1 - n_2))}$$

$$(R_5) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2)}{E \vdash (\text{Mult}(e_1, e_2)) \rightsquigarrow (\text{Vint } (n_1 \times n_2))}$$

$$(R_6) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2) \quad n_2 \neq 0}{E \vdash (\text{Div}(e_1, e_2)) \rightsquigarrow (\text{Vint } (n_1/n_2))}$$

2. Définir une fonction `eval_all` qui étant donné un environnement  $E$  et une expression  $e$  de type `arbre_eval` calcule la valeur de l'expression représentée par  $e$ . Si  $e$  contient une occurrence de variable (libre) qui n'est associée à aucune valeur dans l'environnement  $E$  (c'est à dire si pour cette variable  $x$  il n'existe pas de paire  $(x, v)$  dans  $E$ ), alors cette fonction retournera un message d'erreur adéquat en rattrapant l'exception levée par la fonction `valeur_de` et en provoquant une erreur avec `failwith`.

**Exemple.** On considère l'environnement suivant (qui servira aussi d'environnement d'évaluation dans les exemples des questions suivantes) :

```
let env = [("x", (Vint 8)); ("y", (Vint 2)); ("z", (Vint 3))];;
```

Pour évaluer l'expression  $(3 + 2x) + y$ , on écrira :

```
let e1 = Plus(Plus((Cste 3),
                  (Mult((Cste 2),
                        (Nom "x")))),
            (Nom "y"));;
```

```
# eval_all env e1;;
- : valeur = Vint 21
```

**Variables locales** On étend à présent le langage des expressions arithmétiques de manière à autoriser la présence de variables locales dans les expressions. On ajoute donc au type somme `arbre_eval` le constructeur suivant :

```
| Decl of string*arbre_eval*arbre_eval
```

On enrichit donc le langage  $L$  en permettant la nouvelle construction :

- si  $s$  est un nom de variable et  $e_1$  et  $e_2$  sont des expressions de  $L$  alors la construction  $\text{Decl}(s, e_1, e_2)$  est une expression de  $L$ , il s'agit de la définition locale de  $s$  en  $e_1$  dans l'expression  $e_2$  (le `let x = e1 in e2` de Caml)

Ainsi si l'on veut définir une expression `exp`, représentant l'expression `let x=2+x in x+y`, on écrira :

```
let e2 = Decl("x",
             (Plus((Cste 2),
                  (Nom "x"))),
             (Plus((Nom "x"),
                  (Nom "y"))));;
```

De telles expressions s'évaluent comme suit :

- Si  $e = \text{Decl}(x, e_1, e_2)$ , alors il suffit d'ajouter en tête (i.e., au début) de la liste représentant l'environnement  $E$  la paire  $(x, v)$ , où  $v$  est le résultat de l'évaluation de l'expression  $e_1$  dans l'environnement  $E$ , pour obtenir ainsi l'environnement  $E'$ , puis il reste alors à évaluer l'expression  $e_2$  dans l'environnement  $E'$ .

$$(R_7) : \frac{E \vdash e_1 \rightsquigarrow v \quad (e_1, v) \oplus E \vdash e_2 \rightsquigarrow v'}{E \vdash \text{Decl}(x, e_1, e_2) \rightsquigarrow v'}$$

3. Modifier la fonction `eval_all` afin de prendre en compte les nouvelles expressions.

**Exemple.** L'évaluation de l'expression `e2` dans l'environnement `env` donne le résultat suivant :

```
# eval_all env e2;;
- : valeur = Vint 12
```

**Expressions booléennes** On veut à présent pouvoir comparer le résultat de l'évaluation de deux expressions arithmétiques et manipuler des expressions booléennes. On enrichit donc le langage  $L$  :

- si  $e_1$  et  $e_2$  sont des expressions de  $L$  alors  $\text{Inf}(e_1, e_2)$  et  $\text{Eq}(e_1, e_2)$  sont des expressions de  $L$ , correspondant respectivement aux expressions  $e_1 \leq e_2$  et  $e_1 = e_2$
- si  $e$  est une expression de  $L$ , alors  $\text{Non}(e)$  est une expression de  $L$ .
- si  $e_1$  et  $e_2$  sont des expressions de  $L$  alors  $\text{Et}(e_1, e_2)$  et  $\text{Ou}(e_1, e_2)$  sont des expressions de  $L$

Pour prendre en compte cette nouvelle construction, on introduit des nouveaux constructeurs au type somme `arbre_eval` :

```
| Inf of arbre_eval*arbre_eval
| Eq of arbre_eval*arbre_eval
| Non of arbre_eval
| Et of arbre_eval*arbre_eval
| Ou of arbre_eval*arbre_eval
```

Evidemment, il faut aussi ajouter un constructeur au type `valeur` puisqu'une expression de la forme `Inf(e1, e2)` s'évalue en une valeur booléenne (et non en un entier). On ajoute donc le constructeur :

```
| Vbool of bool
```

au type `valeur`. L'évaluation de ces nouvelles expressions nécessite d'introduire une exception correspondant à une erreur de type (quand on tentera, par exemple, d'évaluer le résultat de l'application d'une opération arithmétique sur des expressions qui s'évaluent en des valeurs booléennes). L'évaluation de ces nouvelles constructions s'obtient facilement comme suit :

- Si  $e = \text{Inf}(e_1, e_2)$ , alors il suffit d'évaluer  $e_1$  et  $e_2$  dans l'environnement  $E$ , puis de vérifier que les résultats de ces évaluations sont bien des objets de type `valeur` obtenus à partir du constructeur `Vint`, et enfin d'effectuer la comparaison (le résultat de l'évaluation sera un objet de type `valeur` obtenu à partir du constructeur `Vbool`).
- Si  $e = \text{Eq}(e_1, e_2)$ , alors il suffit d'évaluer  $e_1$  et  $e_2$  dans l'environnement  $E$ , puis de vérifier que les résultats de ces évaluations sont bien des objets de type `valeur` obtenus à partir du même constructeur `Vint` ou `Vbool`, et enfin d'effectuer la comparaison.
- Si  $e = \text{Not}(e_1)$ , alors il suffit d'évaluer  $e_1$  dans l'environnement  $E$ , puis de vérifier que le résultat de cette évaluation est bien un objet de type `valeur` obtenu à partir du constructeur `Vbool`, et enfin d'effectuer la négation ( $\neg$ ).
- Si  $e = \text{Et}(e_1, e_2)$  (resp.  $e = \text{Ou}(e_1, e_2)$ ), alors il suffit d'évaluer  $e_1$  et  $e_2$  dans l'environnement  $E$ , puis de vérifier que les résultats de ces évaluations sont bien des objets de type `valeur` obtenus à partir du constructeur `Vbool`, et enfin d'effectuer la conjonction  $\wedge$  (resp. la disjonction  $\vee$ ).

$$(R_8) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2)}{E \vdash \text{Inf}(e_1, e_2) \rightsquigarrow (\text{Vbool } (n_1 \leq n_2))} \quad (R_9) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vint } n_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vint } n_2)}{E \vdash \text{Eq}(e_1, e_2) \rightsquigarrow (\text{Vbool } (n_1 = n_2))}$$

$$(R_{10}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool } b_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vbool } b_2)}{E \vdash \text{Eq}(e_1, e_2) \rightsquigarrow (\text{Vbool } (b_1 = b_2))} \quad (R_{11}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool } b_1)}{E \vdash \text{Not}(e_1) \rightsquigarrow (\text{Vbool } (\neg b_1))}$$

$$(R_{12}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool } b_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vbool } b_2)}{E \vdash \text{Et}(e_1, e_2) \rightsquigarrow (\text{Vbool } (b_1 \wedge b_2))} \quad (R_{13}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool } b_1) \quad E \vdash e_2 \rightsquigarrow (\text{Vbool } b_2)}{E \vdash \text{Ou}(e_1, e_2) \rightsquigarrow (\text{Vbool } (b_1 \vee b_2))}$$

4. Modifier la fonction `eval_all` afin de prendre en compte les nouvelles expressions.

**Exemple.** Pour représenter et évaluer l'expression  $e_1 \leq e_2$  dans  $L$  on écrira :

```
let e3 = Inf(e1, e2);;

# eval_all env e3;;
- : valeur = Vbool true
```

**Conditionnelles** On souhaite à présent enrichir  $L$  de manière à pouvoir exprimer des conditionnelles. On ajoute donc au type `arbre_eval` le constructeur suivant :

```
| Altern of arbre_eval*arbre_eval*arbre_eval;;
```

- si  $e_1$ ,  $e_2$  et  $e_3$  sont des expressions de  $L$  alors `Altern(e1, e2, e3)` est une expression de  $L$ , correspondant au `if e1 then e2 else e3` de Caml

L'évaluation des conditionnelles se fait comme suit :

- pour évaluer  $\text{Altern}(e_1, e_2, e_3)$ , il suffit d'évaluer  $e_1$ , de s'assurer que le résultat de cette évaluation est bien un objet de type `valeur` obtenu à partir du constructeur `Vbool` appliqué à un booléen  $b$ , puis selon la valeur de  $b$  d'évaluer  $e_1$  ou  $e_2$

$$(R_{14}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool true}) \quad E \vdash e_2 \rightsquigarrow v}{E \vdash \text{Altern}(e_1, e_2, e_3) \rightsquigarrow v} \quad (R_{15}) : \frac{E \vdash e_1 \rightsquigarrow (\text{Vbool false}) \quad E \vdash e_3 \rightsquigarrow v}{E \vdash \text{Altern}(e_1, e_2, e_3) \rightsquigarrow v}$$

5. Modifier la fonction `eval_all` afin de prendre en compte les nouvelles expressions.

**Exemple.** Pour représenter et évaluer l'expression `if e3 then e1 else e2` dans  $L$  on écrira :

```
let e4 = Altern(e3,e1,e2);;

# eval_all env e4;;
- : valeur = Vint 21
```

**Fonctions et Applications de fonctions** On introduit à présent les expressions fonctionnelles. Une fonction est définie à partir d'un nom de variable – correspondant au paramètre formel de cette fonction, c'est à dire à son argument – et à partir d'une expression – correspondant au corps de cette fonction :

- si  $s$  est un nom de variable et  $e$  est une expression de  $L$  alors  $\text{Fct}(s, e)$  est une expression de  $L$ , correspondant la construction `function s -> e` de Caml

On ajoute donc au type `arbre_eval` le constructeur suivant :

```
| Fct of string*arbre_eval
```

Bien sûr, autoriser la présence de fonction dans un langage sans considérer l'application d'une fonction à un argument présente peu d'intérêt. On introduit donc aussi la construction :

- si  $e_1$  et  $e_2$  sont des expressions de  $L$ , alors  $\text{Appl}(e_1, e_2)$  est une expression de  $L$ , correspondant la construction `(e1 e2)` de Caml

On ajoute donc au type `arbre_eval` le constructeur suivant :

```
| Appl of arbre_eval*arbre_eval
```

Nous allons voir que les fonctions (i.e, les expressions de type `arbre_eval` obtenues à partir du constructeur `Fct`) s'évaluent en des objets, appelés fermetures, obtenus à partir du nom du paramètre formel de la fonction considérée, de son corps, et de l'environnement d'évaluation dans lequel elle a été introduite. On ajoute donc au type `valeur` le constructeur suivant :

```
| Vf of string*arbre_eval*((string*valeur) list)
```

L'évaluation des deux nouvelles constructions s'effectue comme suit :

- dans un environnement d'évaluation  $E$ , une fonction  $\text{Fct}(x, e)$  s'évalue en une fermeture  $\langle x, e, E \rangle$

$$(R_{16}) : \frac{}{E \vdash \text{Fct}(x, e) \rightsquigarrow \text{Vf}(x, e, E)}$$

- dans un environnement d'évaluation  $E$ , pour évaluer une expression  $\text{Appl}(e_1, e_2)$  (correspondant à l'application de la fonction  $e_1$  à l'argument  $e_2$ ), il suffit d'évaluer  $e_1$ , de s'assurer que le résultat de cette évaluation est un objet de type `valeur` de la forme  $\text{Vf}(x, e, EV)$ , puis d'évaluer  $e_2$  en une certaine valeur  $v$  (quelconque) et enfin d'évaluer le corps de la fonction, c'est à dire  $e$ , dans l'environnement  $EV$  dans lequel la valeur du paramètre formel  $x$  est  $v$

$$(R_{17}) : \frac{E \vdash e_1 \rightsquigarrow \text{Vf}(x, e, EV) \quad E \vdash e_2 \rightsquigarrow v \quad (x, v) \oplus EV \vdash e \rightsquigarrow r}{E \vdash \text{Appl}(e_1, e_2) \rightsquigarrow r}$$

6. Modifier la fonction `eval_all` afin de prendre en compte les nouvelles expressions.

*Exemple.* Pour représenter et évaluer l'expression fonction  $x \rightarrow x+1$  dans  $L$  on écrira :

```
let e5 = Fct("x", Plus((Nom "x"), (Cste 1)));;

# eval_all env e5;;
- : valeur =
Vf ("x", Plus (Nom "x", Cste 1),
  [("x", Vint 8); ("y", Vint 2); ("z", Vint 3)])
```

On peut à présent appliquer cette fonction `e5` à un argument, `e2` par exemple :

```
let e6 = Appl(e5, e2);;

# eval_all env e6;;
- : valeur = Vint 13
```

L'argument d'une fonction peut aussi être une fonction. Par exemple, l'expression fonction  $f \rightarrow$  fonction  $x \rightarrow (f x) + 1$  correspondant à la fonction qui étant donnée une fonction  $f$  renvoie une fonction qui étant donné  $x$  renvoie  $(f x) + 1$  peut s'écrire :

```
let e7 = Fct("f",
            (Fct("x", Plus(Appl((Nom "f"), (Nom "x")),
                          (Cste 1)))));;
```

On peut alors appliquer cette fonction `e7` à un argument qui est aussi une fonction, `e6` par exemple :

```
let e8 = Appl(e7, e6);;

# eval_all env e8;;
- : valeur =
Vf ("x", Plus (Appl (Nom "f", Nom "x"), Cste 1),
  [("f",
    Vf ("x", Plus (Nom "x", Cste 1),
      [("x", Vint 8); ("y", Vint 2); ("z", Vint 3)])]);
  ("x", Vint 8); ("y", Vint 2); ("z", Vint 3)])
```

On obtient ainsi une fonction qui peut elle aussi être appliquée :

```
let e9 = Appl(e8, e2);;

# eval_all env e9;;
- : valeur = Vint 14
```

**Le cas des fonctions récursives** La récursivité est aux langages fonctionnels ce que la boucle est aux langages impératifs ... on ajoute donc au langage  $L$  la possibilité de définir des fonctions récursives. Pour cela, il est nécessaire de pouvoir donner un nom à une fonction afin de pouvoir exprimer les appels récursifs dans le corps de la fonction :

- si  $f$  et  $s$  sont des noms de variable et  $e$  est une expression de  $L$  alors  $\text{Fctrec}(f, s, e)$  est une expression de  $L$ , correspondant la construction `rec f s = e` de Caml

On ajoute donc au type `arbre_eval` le constructeur suivant :

```
| Fctrec of string*string*arbre_eval
```

Comme pour les fonctions, on utilise la notion de fermeture pour donner une valeur aux fonctions récursives. Toutefois, la forme des fermetures des fonctions récursives est légèrement différente puisqu'il faut y faire figurer le nom de la fonction récursive. On ajoute donc au type `valeur` le constructeur suivant :

```
| Vfrec of string*string*arbre_eval*((string*valeur) list)
```

- dans un environnement d'évaluation  $E$ , une expression de la forme  $\text{Fctrec}(f, s, e)$  s'évalue en une fermeture  $\langle f, s, e, E \rangle$

$$(R_{18}) : \frac{}{E \vdash \text{Fctrec}(f, s, e) \rightsquigarrow \text{Vfrec}(f, s, e, E)}$$

Tout comme les fonctions, les fonctions récursives ont pour vocation d'être appliquées à des arguments, et l'évaluation des expressions de la forme  $\text{App1}(e_1, e_2)$  doit maintenant prendre en compte cette nouvelle construction :

- dans un environnement d'évaluation  $E$ , pour évaluer une expression  $\text{App1}(e_1, e_2)$  (correspondant à l'application de la fonction récursive  $e_1$  à l'argument  $e_2$ ), il suffit d'évaluer  $e_1$ , de s'assurer que le résultat de cette évaluation est un objet de type `valeur` de la forme  $\text{Vfrec}(f, x, e, EV)$ , puis d'évaluer  $e_2$  en une certaine valeur  $v$  (quelconque) et enfin d'évaluer le corps de la fonction, c'est à dire  $e$ , dans l'environnement  $EV$  dans lequel de la valeur de  $x$  est  $v$  et la valeur de  $f$  est  $\text{Vfrec}(f, x, e, EV)$

$$(R_{19}) : \frac{\begin{array}{l} E \vdash e_1 \rightsquigarrow \text{Vfrec}(f, x, e, EV) \\ E \vdash e_2 \rightsquigarrow v \end{array} \quad (x, v) \oplus (f, \text{Vfrec}(f, x, e, EV)) \oplus EV \vdash e \rightsquigarrow r}{E \vdash \text{App1}(e_1, e_2) \rightsquigarrow r}$$

Il existe donc à présent deux règles qui permettent d'évaluer les expressions de la forme  $\text{App1}(e_1, e_2)$ .

7. Modifier la fonction `eval_all` afin de prendre en compte les nouvelles expressions.

**Exemple.** L'exemple de la fonction factorielle est aux langages fonctionnels ce que le "hello world" est aux langages impératifs, elle peut se définir dans  $L$  comme suit :

```
let e10 = Fctrec("fact", "n",
  (Altern((Inf((Nom "n"),
    (Cste 1))),
    (Cste 1),
    (Mult((Nom "n"),
      (App1((Nom "fact"),
        (Moins((Nom "n"),
```

```
(Cste 1))))))));;
```

```
# eval_all env e10;;  
- : valeur =  
Vfrec ("fact", "n",  
  Altern (Inf (Nom "n", Cste 1), Cste 1,  
    Mult (Nom "n", Appl (Nom "fact", Moins (Nom "n", Cste 1))),  
    [("x", Vint 8); ("y", Vint 2); ("z", Vint 3)])  
  
# eval_all env (Appl(e10,Appl(e6,(Nom "y"))));;  
- : valeur = Vint 6
```

Bien sûr, l'application des fonctions récursives qui ne terminent pas conduit à une évaluation qui ne termine pas :

```
let e11 = Fctrec("foo",  
  "n",  
  (Appl((Nom "foo"),  
    (Appl((Nom "foo"),Plus((Cste 1),(Nom "n"))))))));;  
  
# eval_all env (Appl(e11,(Cste 3))));;  
Stack overflow during evaluation (looping recursion?).
```

**Variables libres d'une expression** L'évaluation d'une expression  $e$  ne peut aboutir que si elle s'effectue dans un environnement qui permet de connaître la valeur de toutes les variables libres de  $e$ . L'ensemble  $\mathcal{F}(e)$  des variables libres d'une expression  $e$  est défini par :

$$\mathcal{F}(e) = \begin{cases} \emptyset & \text{si } e = n \text{ (} n \text{ est un entier)} \\ \{x\} & \text{si } e = x \text{ (} x \text{ est un nom de variable)} \\ \mathcal{F}(e_1) \cup \mathcal{F}(e_2) & \text{si } e = e_1 + e_2 \text{ ou } e = e_1 \times e_2 \text{ ou } e = e_1 - e_2 \\ & \text{ou } e = e_1/e_2 \text{ ou } e = (e_1 \ e_2) \text{ ou } e = e_1 \leq e_2 \\ & \text{ou } e = (e_1 = e_2) \text{ ou } e = e_1 \wedge e_2 \text{ ou } e = e_1 \vee e_2 \\ \mathcal{F}(e_1) & \text{si } e = \neg e_1 \\ \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \cup \mathcal{F}(e_3) & \text{si } e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\ \mathcal{F}(e_1) \cup (\mathcal{F}(e_2) \setminus \{x\}) & \text{si } e = \text{let } x = e_1 \text{ in } e_2 \\ \mathcal{F}(e') \setminus \{x\} & \text{si } e = \text{function } x \rightarrow e' \\ \mathcal{F}(e') \setminus \{f, x\} & \text{si } e = \text{rec } f \ x = e' \end{cases}$$

8. Définir une fonction `list_var_libres` qui étant donnée une expression  $e$  de type `arbre_eval` construit la liste des variables libres apparaissant dans  $e$ .