TD XX

EXPRESSIONS ARITHMÉTIQUES

1 Arbre d'une expression arithmétique

Une expression arithmétique est fondamentalement un arbre :

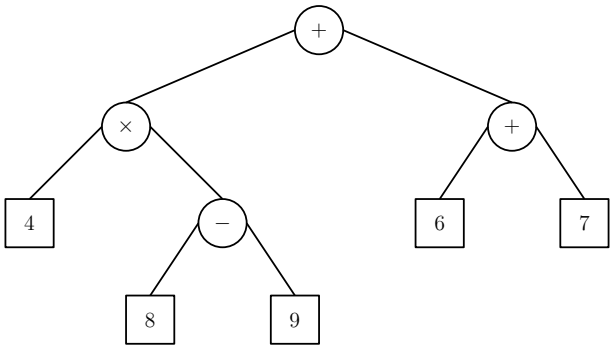

FIGURE XX.1 – Un exemple d'expression arithmétique

On choisit de représenter une expression en utilisant le type suivant :

```
type op =
  | Plus
  | Fois
  | Moins

type expr =
  | C of int
  | N of op * expr * expr
```

Exercice XX.1 p. 5

1. Donner la définition en OCaml de l'arbre représenté ci-dessus.
2. Représenter graphiquement, et définir en OCaml, les arbres correspondant aux expressions `2 + 3 * 4` et `(2 + 3) * 4` (en appliquant les règles de priorité usuelles).

Exercice XX.2 – Évaluation d'une expression p. 5

1. Écrire une fonction `applique : op -> int -> int -> int` qui prend en entrée un opérateur binaire et deux opérandes et renvoie le résultat.

```
# applique Moins 2 5;;
- : int = -3
# applique Fois 4 8;;
- : int = 32
```

2. Écrire une fonction `eval : expr -> int` qui prend l'arbre d'une expression et l'évalue.

2 Différentes notations pour les expressions arithmétiques

En appliquant les différents parcours en profondeur que nous avons vus à l'arbre d'une expression, on obtient différentes représentations « à plat ».

- En notation *infixe*, c'est-à-dire en plaçant les opérateurs entre les opérandes. C'est la notation traditionnelle, mais elle a l'inconvénient de nécessiter des parenthèses.
  Exemples : `2 + (3 * 5) = 17` et `(2 + 3) * 5 = 25`.
  On fixe habituellement des règles de priorité qui permettent d'éliminer une partie des parenthèses (mais seulement une partie...).
- En notation *préfixe*, c'est-à-dire en plaçant les opérateurs avant les opérandes. Aucune parenthèse n'est nécessaire.
  Exemples : `+ 2 * 3 5 = 17` et `* + 2 3 5 = 25`.
- En notation *postfixe*, avec les opérateurs après les opérandes. Là non plus, aucune parenthèse n'est nécessaire.
  Exemples : `2 3 5 * + = 17` et `2 3 + 5 * = 25`.

Remarque
Le fait qu'on ait besoin de parenthèses uniquement dans le cas infixe vient du fait qu'il est possible de reconstruire un arbre à partir de son parcours préfixe ou suffixe (à condition qu'on distingue bien les feuilles des nœuds internes), mais pas à partir de son parcours infixe. Nous l'avons vu, mais pas encore prouvé, en cours.

Exercice XX.3 p. 5

Donner les trois notations possibles pour l'arbre dessiné au début du sujet.

Exercice XX.4 p. 5

1. Écrire les expressions suivantes en notation préfixe et en notation postfixe :
   a. `(3 - 2) * 4`
   b. `(2 + 3) * (1 + 8)`
   c. `(2 + (3 + 4)) * (5 - 6)`
2. Les expressions suivantes sont en notation préfixe. Les traduire en infixe.
   a. `- * 2 3 + 1 4`
   b. `+ * - 4 5 6 7`
3. Les expressions suivantes sont en notation postfixe. Les traduire en infixe.
   a. `2 3 + 4 5 * -`
   b. `1 2 3 4 + * 5 - *`

2.1 À partir de l'arbre

On définit le type suivant :

```
type lexeme = PO | PF | Op of op | Val of int
```

Les constructeurs `PO` et `PF` correspondent respectivement aux parenthèses ouvrantes et fermantes.

Exercice XX.5 p. 6

Écrire les fonctions suivantes (on ira au plus simple sans trop se préoccuper de la complexité) :

1. `prefixe : expr -> lexeme list` qui construit la représentation préfixe d'un arbre.
2. `postfixe` et `infixe` similaires. Pour `infixe`, on ne cherchera pas à éliminer les parenthèses inutiles.

2.2 À partir de la notation postfixe

Exercice XX.6 – Évaluation d'une expression postfixe p. 6

Il est très aisé d'évaluer une expression postfixe à l'aide d'une pile. Par exemple, à partir de l'expression `2 4 - 5 * 6 +`, on obtient (en écrivant la pile avec le sommet à gauche) :

Étape	Pile
2	`[2]`
4	`[4; 2]`
-	`[-2]`
5	`[5; -2]`
×	`[-10]`
6	`[6; -10]`
+	`[-4]`

1. Traiter de la même manière l'expression `1 2 3 4 5 + * - 6 * +`.
2. Que se passe-t-il pour `1 2 + - 3` ? et pour `1 2 3 +` ?
3. Si s est une expression postfixe, on note $\texttt{ent}_s(i)$ le nombre d'entiers apparaissant dans les i premiers éléments de s et $\texttt{op}_s(i)$ le nombre d'opérateurs dans les i premiers éléments. Donner à l'aide de ces fonctions une condition nécessaire et suffisante pour que s soit bien formée (on ne demande pas de démonstration).
4. Écrire une fonction `eval_post : lexeme list -> int` qui prend une liste de lexèmes et l'évalue en tant qu'expression postfixe. On pourra supposer que la liste ne contient pas de parenthèses et on lèvera une exception si l'expression n'est pas bien formée.

Exercice XX.7 p. 7

Écrire une fonction `arbre_of_post : lexeme list -> arbre` qui prend une expression postfixe en entrée et renvoie l'arbre correspondant.
Il suffit d'apporter quelques modifications à la fonction d'évaluation écrite à l'exercice précédent.

3 Expressions avec variables

On considère un ensemble infini dénombrable de variables entières $\mathcal{V} = \{x_0, x_1, \ldots\}$, et l'on étend le type des expressions :

```
type expr2 =
  | N of op * expr2 * expr2
  | C of int
  | V of int
```

Ici, une feuille `V i` ne représente pas l'entier `i` mais la variable x_I. Une feuille `C x`, en revanche, représente directement l'entier `x` (comme depuis le début du sujet).

Une *valuation* est une application de $\mathcal{V}$ dans $\mathbb{Z}$ (qui associe une valeur à chaque variable). En pratique, on n'aura besoin de spécifier les valeurs que d'un nombre fini de variables (celles apparaissant dans l'expression que l'on évalue). On définit donc le type suivant :

```
type valuation = int array
```

Si `t : valuation` est de longueur n, alors la valeur de x_i pour $0 \leqslant i < n$ est donnée par `t.(i)` (et les x_i avec $i \geqslant n$ ont des valeurs non spécifiées).

Exercice XX.8 p. 7

1. Écrire une fonction `max_var : expr2 -> int` qui renvoie le plus grand indice de variable apparaissant dans l'expression reçue en argument. Si l'expression ne contient aucune variable, la fonction pourra avoir un comportement quelconque.
2. Écrire une fonction `eval_contexte : expr2 -> valuation -> int` qui évalue une expression `e` étant donnée une valuation `v` de ses variables. On supposera (sans le vérifier) que `max_var e` est strictement inférieur à la longueur de `v`.

   Remarque

   La fonction `max_var` n'est donc pas utile.

Exercice XX.9 p. 7

On souhaite maintenant pouvoir évaluer *partiellement* une expression étant donnée une valuation qui ne recouvre pas nécessairement toutes les variables présentes dans l'expression. Le résultat de cette évaluation sera donc encore une expression, dans laquelle les calculs possibles ont été effectués.

1. Dans le cas où la valuation spécifie la valeur de toutes les variables de l'expression, quelle forme doit avoir l'expression renvoyée ?
2. Écrire la fonction `eval_partielle : expr2 -> valuation -> expr2`.

4 Forme normale pour l'associativité

Dans cette partie, on se limite pour simplifier à des expressions ne contenant que les opérateurs d'addition et de multiplication :

```
type op = Plus | Fois

type expr3 =
  | C of int
  | V of int
  | N of op * expr3 * expr3
```

Comme les opérateurs $\times$ et $+$ sont associatifs, les expressions `1 + (2 + 3)` et `(1 + 2) + 3` (par exemple) sont équivalentes : on préférerait les représenter toutes les deux par un même arbre.

Pour se faire, on introduit un nouveau type d'arbre pour les expressions :

```
(* Un arbre est soit une feuille, soit un nœud interne avec une
   liste d'enfants *)
type expr_naire =
  | Cn of int
  | Vn of int
  | Nn of op * expr_naire list
```

Une expression `t : expr_naire` sera dite *en forme normale* (pour l'associativité) si :

- chaque nœud interne a au moins deux fils (elle ne contient donc pas de nœud de la forme `N (op, [])` ou `N (op, [e])`);
- un nœud `Plus` (respectivement `Fois`) n'a jamais de fils `Plus` (respectivement `Fois`).

Exercice XX.10 p. 8

Écrire une fonction `normalise : expr3 -> expr_naire` qui prend en entrée une expression (sous forme d'arbre binaire) et renvoie une expression équivalente en forme normale.

Solutions

Correction de l'exercice XX.1 page 1

1.

```
let arbre_exemple =
  N (Plus,
     N (Fois,
        C 4,
        N (Moins,
           C 8,
           C 9)),
     N (Plus,
        C 6,
        C 7))
```

2.
- Pour `2 + 3 * 4`, l'arbre est `N (Plus, C 2, N (Fois, C 3, C 4))`.
- Pour `(2 + 3) * 4`, l'arbre est `N (Fois, N (Plus, C 2, C 3), C 4)`.

Correction de l'exercice XX.2 page 1

1.

```
let applique op x y =
  match op with
  | Plus -> x + y
  | Moins -> x - y
  | Fois -> x * y
```

2.

```
let rec eval expr =
  match expr with
  | C x -> x
  | N (op, g, d) -> applique op (eval g) (eval d)
```

Correction de l'exercice XX.3 page 2

Préfixe : `+ * 4 - 8 9 + 6 7`
Postfixe : `4 8 9 - * 6 7 + +`
Infixe : `(4 * (8 - 9)) + (6 + 7)`

Correction de l'exercice XX.4 page 2

1. **a.** Préfixe : `* - 3 2 4`
Postfixe : `3 2 - 4 *`

b. Préfixe : `* + 2 3 + 1 8`
Postfixe : `2 3 + 1 8 + *`

c. Préfixe : `* + 2 + 3 4 - 5 6`
Postfixe : `2 3 4 + + 5 6 - *`

2. **a.** `(2 * 3) - (1 + 4)`

b. `((4 - 5) * 6) + 7`

3. **a.** `(2 + 3) - (4 * 5)`

b. `1 * (2 * (3 + 4) - 5)`

Correction de l'exercice XX.5 page 2

Les implémentations proposées sont **très inefficaces** (complexité quadratique dans le pire cas). Nous verrons de meilleures manières de procéder, mais ce n'est pas l'objectif aujourd'hui.

```
let rec prefixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> Op op :: prefixe g @ prefixe d

let rec postfixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> postfixe g @ postfixe d @ [Op op]

let rec infixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> PO :: infixe g @ [Op op] @ infixe d @ [PF]
```

Correction de l'exercice XX.6 page 3

1. Laissé au lecteur... On doit obtenir -149.
2. Pour `1 2 + - 3` il n'y a que l'entier 3 sur la pile quand on veut appliquer le `-`, donc il y a une erreur. Pour `1 2 3 +`, il reste à la fin 1 et 5 sur la pile, alors qu'il ne devrait y avoir qu'un seul entier (le résultat final), donc c'est également une erreur.
3. Notons n le nombre de symboles (au total). Il faut, et il suffit, que :
   - $\mathrm{ent}_s(i) \geqslant \mathrm{op}_s(i) + 1$ pour $1 \leqslant i < n$;
   - $\mathrm{ent}_s(n) = 1 + \mathrm{op}_s(n)$.

   Nous ferons la démonstration en cours, dans un cadre plus général.
4. Tout le travail est fait par la fonction auxiliaire, qui prend en argument :
   - `expr` la liste des symboles qu'il faut encore lire ;
   - `pile` l'état actuel de la pile d'évaluation.

   Attention, `expr` est de type `lexeme list` alors que `pile` est de type `int list` (et l'on voit `expr` comme une liste alors que l'on voit `pile` comme une pile, mais c'est une différence plus philosophique qu'autre chose).

   ```
   let eval_post expr =
     let rec eval_aux expr pile =
       match expr, pile with
       | [], [x] -> x
       | Val x :: xs, _ -> eval_aux xs (x :: pile)
       | Op op :: xs, dr :: ga :: reste_pile ->
          eval_aux xs ((applique op ga dr) :: reste_pile)
       | _ -> failwith "expression incorrecte" in
     eval_aux expr []
   ```

   Remarque
   Le dernier cas regroupe en fait trois erreurs possibles :
   - on lit un lexème `PO` ou `PF` (qui n'a rien à faire dans une expression postfixe) ;
   - on lit un opérateur alors qu'il y a zéro ou un entier sur la pile ;
   - on arrive à la fin de l'expression et le nombre d'entiers sur la pile n'est pas exactement un.

Correction de l'exercice XX.7 page 3

On fait exactement la même chose, sauf que l'on remplace la pile d'entiers par une pile d'*arbres*.

```
let arbre_of_post expr =
  let rec aux expr pile =
    match expr, pile with
    | [], [x] -> x
    | Val x :: xs, _ -> aux xs (C x :: pile)
    | Op op :: xs, dr :: ga :: reste_pile ->
       aux xs (N (op, ga, dr) :: reste_pile)
    | _ -> failwith "expression incorrecte" in
  aux expr []
```

Correction de l'exercice XX.8 page 4

1. Renvoyer `min_int` quand il n'y a pas de variable permet de simplifier l'écriture de la fonction :

   ```
   let rec max_var = function
     | C _ -> min_int
     | V i -> i
     | N (_, g, d) -> max (max_var g) (max_var d)
   ```

   On peut bien sûr décider de lever une exception dans ce cas, mais il faut alors (absolument !) distinguer davantage de cas ensuite :

   ```
   let rec max_var = function
     | C _ -> failwith "max de l'ensemble vide"
     | V i -> i
     | N (_, C _, d) -> max_var d
     | N (_, g, C _) -> max_var g
     | N (_, g, d) -> max (max_var g) (max_var d)
   ```

2. On adapte simplement la fonction d'évaluation écrite au début du sujet :

   ```
   let rec eval_contexte e v =
     match e with
     | C x -> x
     | V i -> v.(i)
     | N (op, g, d) -> applique op (eval_contexte g v) (eval_contexte d v)
   ```

Correction de l'exercice XX.9 page 4

1. Dans le cas où toutes les valeurs des variables sont connues, il faut renvoyer une feuille `C x`.
2.

```
let rec eval_partielle e v =
  match e with
  | C x -> C x
  | V i when i < Array.length v -> C (v.(i))
  | V i -> V i
  | N (op, g, d) ->
     match eval_partielle g v, eval_partielle d v with
     | C xg, C xd -> C (applique op xg xd)
     | g', d' -> N (op, g', d')
```

On pourrait très facilement ajouter des règles de simplification : par exemple, dans le `match` interne, le cas `C 0, d` donnerait 0 si `op = Fois` et `d` si `op = Plus`... Cependant, ce n'était pas demandé ici.

Correction de l'exercice XX.10 page 4

Le code est très simple, ce qui ne veut pas forcément dire qu'il est simple à écrire.

```
let rec remonte op = function
  | Nn (op', enfants) when op = op' -> enfants
  | e -> [e]

let rec normalise = function
  | C x -> Cn x
  | V x -> Vn x
  | N (op, g, d) ->
     Nn (op, remonte op (normalise g) @ remonte op (normalise d))
```

Si vous êtes arrivé jusqu'ici, vous avez bien mérité vos vacances. Si vous n'êtes pas arrivé jusqu'ici, vous les avez méritées quand même, mais vous ne le saurez jamais...