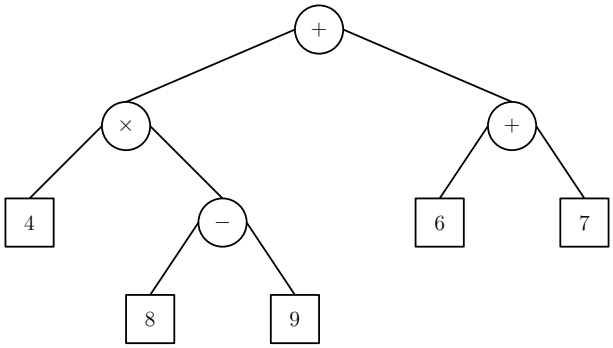
# TP : Expressions arithmétiques

## 1 Arbre d'une expression arithmétique

Une expression arithmétique est fondamentalement un arbre :




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
```

### 1.1 Exercice

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).

### 1.2 Évaluation d'une expression

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`.

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.

## 2.1 Exercice

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

## 2.2 Exercice

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 - *`

## 2.3 À partir de l'arbre

On définit le type suivant :

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

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

## 2.4 Exercice

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.

## 2.5 À partir de la notation postfixe

### 2.5.1 Évaluation d'une expression postfixe

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 $\text{ent}_s(i)$ le nombre d'entiers apparaissant dans les $i$ premiers éléments de $s$ et $\text{op}_s(i)$ 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.

### 2.5.2 Exercice

Écrire une fonction `arbre_of_post : lexeme list -> expr` 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 $\mathcal{V} = \{x_0, x_1, \dots\}$, 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).

## 3.1 Exercice

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

## 3.2 Exercice

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 noeud 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`).

## 4.1 Exercice

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.