Cours 4 : Définition de types, types récursifs généraux et arbres

Pour l'instant nous n'avons vu que des types prédéfinis (récursifs ou non). Ici, nous allons définir des nouveaux types. Un type (de données) se définit comme le regroupement d'un ensemble de situations/données différentes. Chaque situation sera distinguée par un identifiant spécial, un **constructeur**, qui indique dans quel sous-cas on se trouve. Ces cas peuvent être distingués par filtrage, comme pour les listes. La syntaxe des arguments (lorsqu'il y en a) du constructeur, dans un filtre ou une expression, est similaire à celle des arguments de fonctions. Un contructeur est ainsi appelé parce qu'il permet de construire une donnée "plus grande" en réunissant plusieurs sous-données. Cette vision se prête bien à l'analyse récursive des problèmes.

Le type `'a list` est un exemple de type récursif défini par constructeurs (`[]` et `::`, même si ceux-ci possèdent une syntaxe spécifique).

1 Définition de types

1.1 Alias de type

Si le type de données à définir ne comporte qu'un seul cas, alors l'usage des constructeurs peut être évité comme par exemple des files implantées par des paires de listes :

```
type 'a !!file!! = 'a list * 'a list
```

1.2 Constructeurs constants

```
type couleur = Bleu | Blanc | Vert
```

Attention : Bleu, Blanc et Vert deviennent des mots clés. Les constructeurs commencent par une majuscule, alors que les identificateurs commencent par une minuscule.

```
type jour =
Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche

(* Fonction qui v[U+FFFD]rifie si le jour est un vendredi 13 *)
let chance (j, d) =
  match j, d with
  | Vendredi, 13 -> true
  | _            -> false
(* val chance : jour * int -> bool *)

let chance (j, d) =
 j = Vendredi && d = 13
(* val chance : jour * int -> bool *)
```

1.3 Constructeurs avec arguments

```
# type num = Entier of int | Flottant of float;;

# type complexe = Polaire of (float * float) | Cartesien of (float * float);;
```

```
# Entier 1;;
- : num = Entier 1

# Polaire (1.0, atan 1.);;
- : complexe = Polaire (1.0, 0.78...)

# Cartesien (sqrt 2., sqrt 2.);;
- : complexe = Cartesien (1.41..., 1.41...)

(* conversion : complexe -> complexe *)
(* Conversion de complexe, de polaire vers cart[U+FFFD]sien *)
let conversion c =
 match c with
 | Polaire (//module//, angle) -> Cartesien (//module// *. cos angle, //module// *. sin
 | _                           -> c

(* egal_complexe : complexe -> complexe -> bool *)
(* Teste l'egalite entre deux complexes *)
let egal_complexe c1 c2 =
 conversion c1 = conversion c2
```

Exemple : le type option est une **structure de données** servant à spécifier un choix entre une situation normale (on a une valeur) et une situation exceptionnelle (on a un échec, une absence de solution, etc) sans arrêter l'exécution comme une exception peut le faire. On pourra donc savoir si un calcul a "échoué" ou non. On peut faire ce parallèle avec les autres structures de données :

- une paire encode 2 valeurs.
- un n-uplet encode n valeurs.
- une liste encode de 0 à n valeurs.
- une option encode 0 ou 1 valeur.

Syntaxe :

```
type 'a option =
| None
| Some of 'a
```

1.5 Types récursifs paramétrés

Les paramètres ne sont même pas nécessairement homogènes, i.e. on peut par exemple définir :

```
type 'a power_list = Nil | Cons of 'a * ('a * 'a) power_list
```

2 Arbre binaire

2.1 Spécification et implantation

Le type 'a power_list est intéressant, car il représente les arbres binaires équilibrés, mais délicat à utiliser en pratique, surtout avec des fonctions récursives, à cause de l'algorithme de typage de caml. Plus sagement, on peut écrire un type "arbre binaire" classique, qui contiendra des données :

- dans les nœuds.

— dans les branches.
— dans les feuilles.

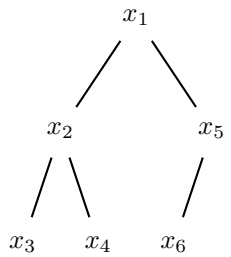
Tous les mélanges sont bien sûr possibles, avec des élément de types différents, etc.

Voici le type arbre binaire standard, où les **données sont dans les nœuds** de l'arbre :

```
type 'a standard_tree =
| Empty
| Node  of 'a * 'a standard_tree * 'a standard_tree
```

Exemple :

```
Node ("x1",
      Node ("x2",
            Node ("x3",Empty,Empty),
            Node ("x4",Empty,Empty)
           ),
      Node ("x5",
            Node ("x6",Empty,Empty),
            Empty
           )
      )
```

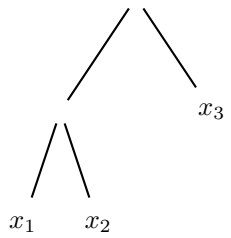


Voici le type arbre binaire où les **données sont dans les feuilles** de l'arbre :

```
type 'a leaf_tree =
| Leaf of 'a
| Node of 'a leaf_tree * 'a leaf_tree
```

Exemple :

```
Node (Node (Leaf "x1",
            Leaf "x2"
           ),
      Leaf "x3"
      )
```

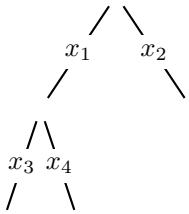


Voici le type arbre binaire où les **données sont dans les branches** de l'arbre :

```
type 'a edge_tree =
| Empty
| Edges of ('a * 'a edge_tree) * ('a * 'a edge_tree)
```

Exemple :

```
Edges (("x1",
        Edges(("x3",Empty),
              ("x4",Empty))),
       ("x2",
        Empty)
       )
```



2.3 Application

L'arbre binaire "standard" est le plus répandu, c'est la définition que l'on considère pour l'exemple suivant et les exercices (en fin de polycopié).

Exemple Nous souhaitons définir une fonction qui compte le nombre d'éléments d'un arbre.

La structure récursif du type 'a standard_tree donne la structure récursif de la fonction.

Question : Si je sais calculer la taille des deux fils de la racine de l'arbre, comment est-ce que je calcule la taille de l'arbre ?

Réponse : Il suffit d'ajouter les deux tailles, et d'incrémenter (pour la racine).

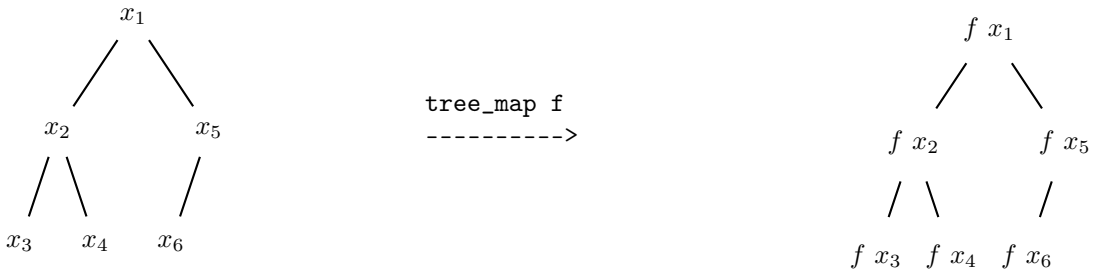
```
(* Cardinal : 'a standard_tree -> int *)
(* Renvoie le nombre d'elements d'un arbre *)
let rec cardinal arb =
 match arb with
  | Empty            -> 0
  | Node  (_, g, d) -> 1 + cardinal g + cardinal d
```

2.5 Itérateurs structurels

Comme pour les listes, on cherche des itérateurs structurels "universels". Un itérateur structurel se contente de remplacer les constructeurs par des appels de fonctions.

2.5.1 Map

Nous voulons l'équivalent de List.map pour les arbres binaires standard.



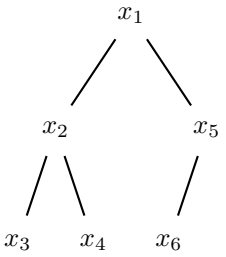

```
(* tree_map : ('a -> 'b) -> 'a standard_tree -> 'b standard_tree *)
let rec tree_map f arb =
```

```
match arb with
 | Empty          -> Empty
 | Node (n, g, d) -> Node (f n, tree_map f g, tree_map f d)
```

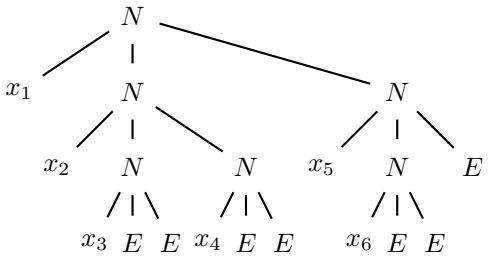
2.5.2 Fold

Nous voulons l'équivalent de List.fold_right pour les arbres binaires standard.

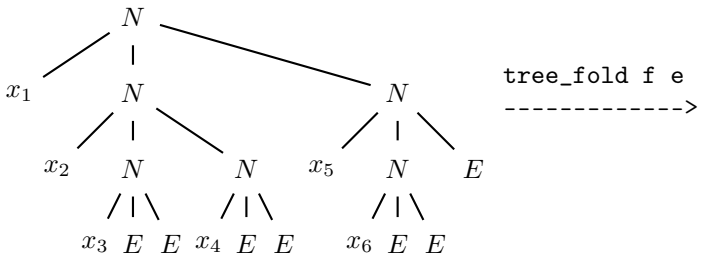
Notons que l'arbre :



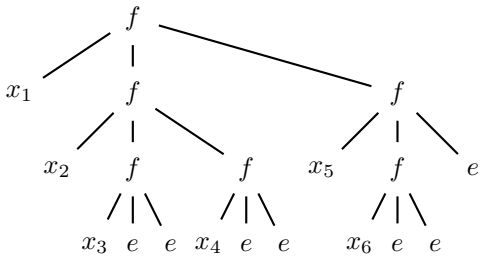
peut aussi être représenté par (où Node est abrégé en N et Empty en E) :



Tout comme List.fold_right "transformait" les constructeurs en appels de fonction, le comportement de tree_fold est :




```
tree_fold f e
------------->
```




```
(* tree_fold : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a standard_tree -> 'b *)
let rec tree_fold f e arb =
 match arb with
 | Empty          -> e
 | Node (n, g ,d) -> f n (tree_fold f e g) (tree_fold f e d)
```

La fonction f prend trois arguments qui correspondent :
- au contenu du nœud
- au fils gauche "déjà traité"
- au fils droit "déjà traité"

et renvoie le traitement sur l'arbre entier.

Généralisation Pour définir le type et le corps de la fonction `fold`, il faut "remplacer" les constructeurs du type par des appels de fonctions de même arité. Le nombre et le type des paramètres de le l'itérateur `fold` dépend donc du nombre et du type des constructeurs de la structure de données sur laquelle l'itérateur itère.

2.5.3 Applications

La taille d'un arbre binaire se définit à l'aide de l'itérateur fold :

```
(* cardinal : 'a standard_tree -> int *)
(* Renvoie le nombre d'elements d'un arbre *)
let cardinal arb =
 tree_fold (fun _ cardinal_g cardinal_d -> 1 + cardinal_g + cardinal_d) 0 arb
```

3 Structure de données "efficace" : Arbres Binaires à Gauche ("leftist binary trees")

3.1 Contrainte structurelle

La plupart des structures de données “efficaces” nécessitent la mise en place d'**invariants structurels**, i.e. des propriétés locales à chaque nœud qui ne peuvent être exprimées par le type seul et qui garantissent globalement l'efficacité des opérations. De telles propriétés sont par exemple :
- des propriétés numériques portant sur la taille, la profondeur, etc.
- des propriétés d'ordre entre éléments.
- des propriétés portant sur des données auxiliaires ajoutées (arbres colorés Rouge-Noir par exemple).

Lorsque les invariants sont suffisamment forts, nous pouvons obtenir une représentation canonique.

Enfin, ces invariants doivent obligatoirement être maintenus par les constructeurs abstraits (à supposer que leurs arguments les respectent : programmation offensive), si l'on veut que les structures créées soient toujours correctes. Les types doivent donc être abstraits/privés, afin que l'utilisateur ne puisse pas manipuler “à la main” les constructeurs “concrets” et casser les invariants.

3.2 Spécification

La structure des arbres binaires à gauche permet de rétablir un certain équilibre entre les branches d'un arbre binaire, et permet de réaliser les opérations d'insertion et même d'union de manière efficace. Par contre, seul le retrait de l'élément **minimum** est possible de manière efficace. La complexité du pire cas de ces opérations est logarithmique.

Les arbres binaires à gauche respectent le schéma de type standard des arbres binaires :

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

et reposent sur ces deux principes :
- **Invariant 1** : Les éléments de l'arbre sont ordonnés en tas ("heap ordered"), i.e. pour tout sous-arbre non vide, l'élément à sa racine est toujours inférieur à chaque élément présent dans ses fils gauche et droit.

— **Invariant 2** : Les branches penchent à gauche, i.e. pour tout sous-arbre non vide, son fils gauche est au moins aussi profond que son fils droit.

L'union entre deux arbres binaires à gauche non vides (seul cas non trivial) peut se décomposer en deux phases :

— Décomposition : on insère l'arbre de racine la plus grande dans le fils droit de l'autre arbre, afin de respecter l'invariant 1.

— Recomposition : on échange les fils gauche et droit du résultat si besoin est, afin de respecter l'invariant 2.

3.3 Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg

(* profondeur 'a abg -> int *)
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
(* On s'interesse a la branche la plus a gauche, car par definition c'est la plus profond
let rec profondeur a =
  match a with
  |Vide          -> 0
  |Noeud(g,r,d) -> 1 + profondeur g

(* noeud : 'a abg -> 'a -> 'a abg -> 'a abg *)
(* Constructeur abstrait qui construit un noeud en permutant les 2 branches si necessaire
(* afin de respecter l'invariant 2 *)
let noeud g r d =
  if profondeur g < profondeur d then Noeud (d, r, g) else Noeud (g, r, d)

(* union : 'a abg -> 'a abg -> 'a abg *)
(* Calcule l'union de deux arbres binaires a gauche *)
let rec union abr1 abr2 =
  match abr1, abr2 with
  |Vide           , _              -> abr2
  |_              , Vide           -> abr1
  |Noeud(g1,r1,d1), Noeud(g2,r2,d2) ->
    if (r1 > r2)
    then noeud g2 r2 (union abr1 d2)
    else noeud g1 r1 (union abr2 d1)

(* ajout : 'a -> 'a abg -> 'a abg *)
(* Ajoute un element a un arbre binaire a gauche *)
let ajout e arb =
  union (Noeud(Vide,e,Vide)) arb

(* minimum : 'a abg -> 'a *)
(* Renvoie le minimum d'un arbre binaire a gauche *)
(* ie la racine *)
(* Erreur si l'arbre est vide *)
let minimum abr =
  match abr with
  |Vide          -> failwith "Arbre vide!"
  |Noeud(g,r,d) -> r
```

```
(* retrait_min : 'a abg -> 'a abg *)
(* Retire son minimum a un arbre binaire a gauche *)
(* Erreur si l'arbre est vide *)
let retrait_min abr =
  match abr with
  |Vide          -> failwith "Arbre vide!"
  |Noeud (g,r,d) -> union g d
```

4 Parcours d'arbres binaires

On considère le schéma récursif standard.

```
type 'a standard_tree =
| Empty
| Node  of 'a * 'a standard_tree * 'a standard_tree
```

Un parcours des éléments d'un arbre consiste à présenter ses éléments séquentiellement (i.e. "linéariser") en vue d'itérer un traitement particulier sur cette séquence. Ce type d'algorithme correspond à une approche récursive terminale avec accumulateurs. Contrairement aux listes, où un seul type de parcours avec accumulateur existe (i.e. List.fold_left), le cas des arbres donne lieu à de multiples possibilités. On envisagera néanmoins uniquement les parcours de gauche à droite des éléments (sachant d'une part que c'est ce que l'on fait pour les listes, et d'autre part que calculer l'image miroir d'un arbre est simple, ce qui autorise alors les parcours de droite à gauche).

Pour simplifier le problème, on effectue une décomposition fonctionnelle en remarquant qu'un tel parcours peut se scinder en deux étapes de calcul :

1. construire la liste (séquence) des éléments, dans l'ordre où le traitement à itérer les trouverait.
2. appliquer itérativement ce traitement sur la liste obtenue (avec un List.fold_left par exemple).

On ne s'intéressera donc qu'à la construction de la dite liste.

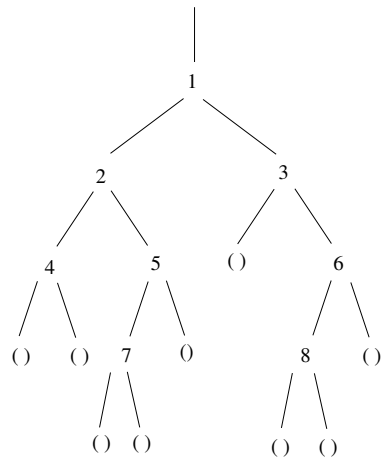
Pour parcourir les éléments situés dans les noeuds d'un arbre, on dispose principalement de deux types de parcours : en largeur et en profondeur.

4.1 Parcours en profondeur

Le parcours en profondeur consiste, pour un arbre non vide, à explorer la branche gauche complètement (depuis la racine jusqu'aux feuilles), puis à explorer la branche droite. Le traitement de l'élément à la racine de l'arbre donne lieu à trois possibilités selon que :

— on traite la racine avant ses fils. Parcours **préfixe**.
— on traite la racine après ses fils. Parcours **postfixe**.
— on traite la racine entre son fils gauche et son fils droit. Parcours **infixe**.

Exemple :



— Préfixe : `[1; 2; 4; 5; 7; 3; 6; 8]`
— Postfixe : `[4; 7; 5; 2; 8; 6; 3; 1]`
— Infixe : `[4; 2; 7; 5; 1; 3; 8; 6]`

4.3.1 Implantation

Un parcours en profondeur est "naturel", au sens où il correspond à la sémantique des appels récursifs. Il est possible de l'implanter avec une fonction récursive structurelle, sans autre argument que l'arbre.

```
let rec parcours_profondeur_prefixe a =
tree_fold (fun racine prof_prefix_g prof_prefix_d -> racine::(prof_prefix_g--@--prof_pre

let rec parcours_profondeur_postfixe a =
tree_fold (fun racine prof_postfix_g prof_postfix_d -> prof_postfix_g--@--(prof_postfix_

let rec parcours_profondeur_infixe a =
tree_fold (fun racine prof_infixe_g prof_infixe_d -> prof_infixe_g--@--(racine::prof_infi
```

Les appels récursifs utilisent implicitement la pile d'appels (qui contient les environnements successifs définissant les paramètres, ainsi que les résultats). Il est également possible d'utiliser une pile utilisateur plus simple pour réaliser un parcours en profondeur "à la main".

```
let parcours_prefixe arb =
let rec --parcours-- pile =
 match pile with
 | []                -> []
 | Empty::q          -> --parcours-- q
 | Node (n, g, d)::q -> n::--parcours-- (g::d::q)
 in --parcours-- [arb]
```

4.6 Parcours en largeur

Le parcours en largeur consiste à parcourir les noeuds de l'arbre, "ligne" par "ligne". Cela ne correspond pas du tout à la récursivité naturelle, et bien qu'il existe une solution récursive structurelle, elle est effroyablement compliquée et il vaut mieux utiliser une structure de données auxiliaire.

Avec l'arbre donné en exemple, on aura le parcours en largeur suivant : `[1;2;3;4;5;6;7;8]`. Par rapport à la pile explicite :

— la racine apparaît également dans la liste résultat avant ses fils.
— la racine du fils gauche apparaît également avant la racine du fils droit.
— par contre, les sous-noeuds du fils gauche apparaissent **après** la racine du fils droit.

Comment faire ? Étant donné que la pile contient Node(2,.,.); Node(3,.,.), après le traitement de la racine, il suffit d'insérer les fils du nœud gauche à la fin de la structure, **après le fils droit**. Cette règle est valable pour tous les nœuds. On obtient donc une **file** (retrait en tête, insertion en queue) à la place d'une pile.

```
let parcours_largeur arb =
let rec parcours file =
 match file with
 | []                -> []
 | Empty::q          -> --parcours-- q
 | Node (n, g, d)::q -> n::--parcours-- (q--@--[g; d])
 in --parcours-- [arb]
```

On utilise ici une version naïve et inefficace de la file par concaténation à droite.

5 Les arbres n-aires

On s'intéresse aux arbres n-aires, des arbres avec un nombre quelconque de fils. Nous cherchons une représentation **canonique**, i.e. telle que l'égalité entre objets représentés (ici les arbres n-aires au sens mathématique) se ramène à l'égalité structurelle sur la structure de données choisie (ici, le type arbre n-aire).

Nous choisissons une structure avec des données dans les nœuds, mais les variantes vues précédemment sur les arbres binaires (données dans les feuilles, données dans le branche, combinaison des ces trois possibilités) sont également possibles.

5.1 Spécification

La spécification d'un module `Arbre_naire`, contient :
- le type, les constructeurs et les sélecteurs
- les itérateurs map et fold, sur le modèle des itérateurs de listes

Les arbres n-aires ne peuvent pas être vides (sinon on pourrait avoir des sous-arbres vides dans la liste des fils, donc représentation non canonique). Si on veut des arbres vides, on peut utiliser le type arbre_naire option par exemple.

Le fichier `Arbre_naire.mli` contient donc :

```
(* un arbre dont le nombre de fils d'un noeud est quelconque : arbre n-aire *)
type 'a arbre_naire

(* constructeurs "abstraits" *)
val --cons-- : 'a -> 'a arbre_naire list -> 'a arbre_naire

(* s[U+FFFD]lecteurs *)
val --racine-- : 'a arbre_naire -> 'a

val --fils--   : 'a arbre_naire -> 'a arbre_naire list
```

L'itérateur fold peut être défini de différentes façons, selon que l'on parcoure également la liste des fils, ou que l'on laisse l'utilisateur s'en charger. Sachant qu'un itérateur doit fournir autant de paramètres que de constructeurs du type, la seconde version est plus simple (1 paramètre pour 1 constructeur) au lieu de 3 (on rajoute les 2 constructeurs du type liste) et donc préférable.

Le fichier `Arbre_naire.mli` contient donc :

```
(* it[U+FFFD]rateurs *)
val map : ('a -> 'b) -> 'a arbre_naire -> 'b arbre_naire

val fold : ('a -> 'b list -> 'b) -> 'a arbre_naire -> 'b
```

5.2 Implantation

L'implantation du module `Arbre_naire` est alors :

```
type 'a arbre_naire =
 | Noeud of 'a * 'a arbre_naire list
```

```
let --cons-- racine fils = Noeud (racine, fils)

let --racine-- (Noeud (racine, _)) = racine

let --fils-- (Noeud (_, fils)) = fils

(* principe : une fonction / constante pour chaque constructeur: Noeud *)
let rec fold fNoeud (Noeud (racine, fils)) =
 fNoeud racine (List.map (fold fNoeud) fils)

let rec map f (Noeud (r, fils)) =
 Noeud (f r, List.map (map f) fils)

(* ou bien *)
let map f arb =
 fold_arbre (fun r liste_map_fils -> Noeud (f r, liste_map_fils)) arb
```

Tous les itérateurs sont nécessairement en temps linéaire (sans compter l'effet de la fonction appliquée).

5.3 Application

La fonction qui renvoie le nombre d'éléments d'un arbre n-aire, s'écrit alors :

```
(* cardinal : 'a arbre_naire -> int *)
(* Renvoie le nombre d'elements d'un arbre *)
let cardinal arb = fold (fun _ liste_cardinal_fils -> 1 + List.fold_right (+) liste_card
```

6 Problème : écriture d'un évaluateur d'expression

Les arbres peuvent ne pas être utilisés sous leur forme générique mais "instanciés" pour représenter une structure arborescente avec plus de précision. Nous allons illustrer cela en écrivant en évaluateur d'une sous-partie des expressions OCaml.

6.1 Grammaire des expressions

La grammaire formelle (cf cours de modélisation de 1A) est de la forme :

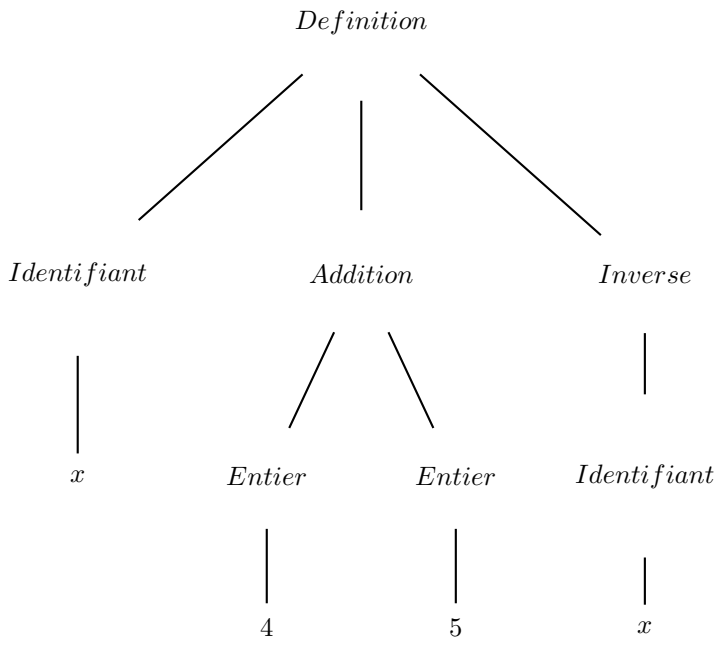
1. $E \rightarrow let\ id\ =\ E\ in\ E$
2. $E \rightarrow (E)$
3. $E \rightarrow E + E$
4. $E \rightarrow E - E$
5. $E \rightarrow -E$
6. $E \rightarrow id$
7. $E \rightarrow entier$

6.2 Définition du type OCaml

Les expressions se représentent naturellement sous forme d'arbre. Par exemple, l'expression :

```
let x = 4+5 in -x
```

peut être représentée par l'arbre :



Un type possible pour représenter les expressions est alors :

```
type expression =
| Definition of string * expression * expression
| Addition of expression * expression
| Soustraction of expression * expression
| Inverse of expression
| Identifiant of string
| Entier of int
```

6.3 Écriture de l'évaluateur

S'il n'y avait pas les définitions l' écriture de l'évaluateur serait triviale :

```
(* evalue : expression -> int *)
let rec evalue exp =
match exp with
| Addition (e1,e2) -> (evalue e1)+(evalue e2)
| Soustraction (e1,e2) -> (evalue e1)-(evalue e2)
| Inverse e -> -(evalue e)
| Entier i -> i

# evalue (Addition ((Entier 3),(Entier 4))) ;;
- : int = 7
```

La présence des définitions nécessite de gérer un environnement. L'environnement peut être représenté par une liste de couples *(identifiant,valeur)*.

```
(* evalue : expression -> int *)
let evalue exp=
  let rec --aux-- exp env =
    match exp with
    | Definition (i,def,e) -> --aux-- e ((i,--aux-- def env)::env)
    | Addition (e1,e2) -> (--aux-- e1 env)+(--aux-- e2 env)
    | Soustraction (e1,e2) -> (--aux-- e1 env)-(--aux-- e2 env)
    | Inverse e -> -(--aux-- e env)
    | Identifiant id -> List.assoc id env
    | Entier i -> i
  in --aux-- exp []

# evalue (Definition ("x",Addition ((Entier 4),(Entier 5)) , Inverse (Identifiant "x")))
- : int = -9
```

7 Exercices

7.1 Arbre binaire

▷ **Exercice 1**
- *Définir une fonction qui compte le nombre d'éléments d'un arbre.*
- *Définir une fonction qui cherche si un élément x apparaît dans un arbre.*
- *Définir une fonction qui calcule les profondeurs minimum et maximum d'un arbre.*
- *Définir une fonction qui remplace toutes les occurrences d'un élément dans un arbre par un autre élément.*

▷ **Exercice 2** *À l'aide des itérateurs structurels, définir :*
- *la taille d'un arbre binaire.*
- *la profondeur maximale d'un arbre binaire.*
- *le parcours préfixe d'un arbre binaire.*

7.5 Arbre n-aire

▷ **Exercice 3** *À l'aide des itérateurs, définir sur les arbres n-aires :*
- *la fonction qui renvoie le nombre d'éléments.*
- *la fonction qui teste l'appartenance d'un élément.*
- *la fonction qui renvoie la liste résultant d'un parcours en profondeur préfixe.*