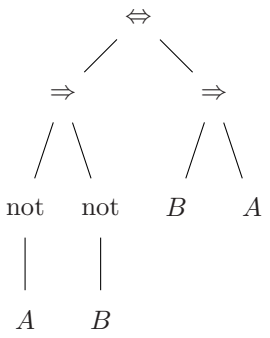
Le but de ce TP est de comprendre l'intérêt des arbres syntaxiques et leur interprétation.
Le cadre est le suivant : on dispose d'un ensemble de variables V, et d'un ensemble de symboles fonctionnels Σ. Un symbole $f \in \Sigma$ est caractérisé par son arité $\alpha(f) \in \mathbb{N}$, que l'on peut voir comme le "nombre d'arguments" de f. Un symbole $f \in \Sigma$ d'arité 0 est appelé symbole constant.

Une *expression* sera un arbre étiqueté par $\Sigma \cup V$ tel que :
- les feuilles sont étiquetées par des variables ou des symboles constants ;
- les noeuds ayant n fils sont étiquetées par des symboles d'arité n.

Exemple Pour représenter des expressions logiques en les variables A et B, on prend :
- Symboles d'arité 0 : $\{\text{Vrai}, \text{Faux}\}$;
- Symboles d'arité 1 : $\{\text{not}\}$;
- Symboles d'arité 2 : $\{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$;
- Variables : $\{A, B\}$

Ainsi, l'arbre suivant est une expression logique :

Quelle formule logique voudriez-vous qu'elle représente ?



1 Calcul formel

On va utiliser ces arbres pour faire du calcul formel. Pour simplifier, on se limite aux symboles suivants :
- Symboles d'arité 0 : les flottants ;
- Symboles d'arité 1 : $\{\text{exp}\}$;
- Symboles d'arité 2 : $\{+, *\}$;
- Variables : les caractères.

Pour manipuler ces arbres, on définit le type suivant :

```
type expr = Var of char
          | Const of float
          | Exp of expr
          | Somme of expr * expr
          | Produit of expr * expr ;;
```

Question 1. Écrire une fonction `calcule : expr -> float` qui prend en argument une expression et renvoie sa valeur. On déclenchera une exception lorsqu'une variable apparaît dans l'expression.

Question 2. Écrire une fonction `evalue : expr -> (char -> float) -> float` qui prend non seulement une expression, mais aussi une fonction qui dit ce que valent les variables, et renvoie la valeur associée à l'expression.

Question 3. Écrire une fonction `derive : char -> expr -> expr` qui dérive une expression par rapport à la variable donnée en argument.

Question 4. Que renvoie `derive 'x' (Exp (Somme (Var 'x',Const 1.)))` ?
Vous pouvez écrire une fonction `simplifie : expr -> expr` qui effectue quelques simplifications sur une expression.

Dans cette partie, on est passé de la *syntaxe* (façon d'organiser et de représenter les objets) à la *sémantique* (liée au sens, interprétation de cette écriture). Les objets syntaxiques comme le symbole `Exp` pourraient représenter n'importe quoi ; ici, on l'a *interprété* comme la fonction exponentielle des réels.
Par exemple, les expressions `Exp(Const(0.))` et `Const(1.)` sont deux objets syntaxiques différents, mais leur interprétation sera la même.
Dans la partie suivante, on va faire quelque chose de purement syntaxique.

2 Syntaxe concrète

Vous avez dû vous rendre compte que les arbres syntaxiques sont bien utiles pour le programmeur, mais pas très pratiques pour l'utilisateur qui n'a pas envie d'écrire tout les temps des choses du style `Exp (Somme (Var 'x',Const 1.))`. On va ici voir comment passer d'écritures plus naturelles à la structure d'arbres.

On se placera dans le cas général suivant : on travaille avec le type

```
type expr = C of string | U of string * expr | B of string * expr * expr ;;
```

où les `C` sont les constantes, les `U` (unaire) sont les symboles d'arité 1, et les `B` (binaire) les symboles d'arité 2. Le `string` sert simplement à donner un nom à ces différents symboles, par exemple, on écrira `U("exp", B("+", C "x", C "1") )`. Remarquez qu'on n'a pas différencié les constantes des variables, ce qui ne posera pas de problème dans ce qu'on va faire.

2.1 Forme postfixe

On appelle *écriture postfixe* d'une expression t le mot de Σ^* défini inductivement par :

$$\text{Post}(t) = \begin{cases} t & \text{si } t \text{ est constant} \\ \text{Post}(t_1)\dots\text{Post}(t_n)f & \text{si } t = f(t_1,\dots,t_n) \end{cases}$$

Question 5. Écrire sous forme postfixe l'exemple d'arbre syntaxique précédent.

Question 6. Écrire une fonction `ecritpost : expr -> unit` qui imprime l'écriture postfixe d'une expression quelconque.

On va maintenant s'intéresser au problème inverse : étant donné une écriture postfixe, comment retrouver l'arbre syntaxique associé ?

Une méthode possible est la suivante : on utilise une pile p initialement vide. On lit l'un après l'autre les symboles du mot :

- à chaque symbole constant, on l'empile ;
- à chaque symbole f d'arité $n \geq 1$, on dépile successivement $t_n, \ldots, t_1$ (attention à l'ordre !), puis on empile $f(t_1, \ldots, t_n)$.

Il est possible de montrer que si le mot correspondait à $\text{Post}(t)$, alors l'algorithme ne déclenche pas d'erreur et qu'à la fin il ne reste dans la pile qu'un seul élément, l'expression t. Réciproquement, si l'algorithme appliqué à un mot u ne déclenche pas d'erreur et finit avec un seul élément élément dans la pile, alors u était bien une écriture postfixe.

Question 7. On représente le mot par une `int*string list` : les éléments de cette liste sont les symboles dans Σ, représentés sous forme de couples où l'entier indique leur arité (0,1 ou 2) et la chaîne de caractère leur nom.
Par exemple, `[(0,"1");(0,"x");(2,"+")]` représente le mot `1 x +`.

Écrire une fonction `arbre de liste : int*string list -> expr` qui calcule l'expression associée à cette écriture, grâce à l'algorithme précédent.

Question 8. Cette écriture sous forme de liste n'est pas non plus très satisfaisante ; on aimerait bien pouvoir écrire carrément des chaînes de caractères. Le problème qui consiste à transformer une chaîne de caractère en, mettons, la liste précédente, est un problème de *parsing*.
Dans cette petite introduction, nous utiliserons la convention suivante : les noms des constantes seront introduits par la lettre `C`, les fonctions unaires par la lettre `U` et les fonctions binaires par la lettre `B`. Ces trois lettres seront interdites dans les noms des fonctions et constantes.
Par exemple, la chaîne `"C1CxB+"` correspond à l'exemple de la question précédente.

Écrire une fonction `parse : string -> int*string list` qui transforme cette chaîne de caractère en la liste associée (on ne gardera pas les `C`, `U` et `B` dans les noms), puis une fonction `arbre de chaine : string -> expr` qui transforme une telle écriture en son arbre syntaxique.

Exemple.
```
#arbre_de_chaine"CxC1B+Uexp";;
- : expr = U ("exp", B ("+", C "x", C "1"))
```

2.2 Forme préfixe

L'écriture préfixe d'une expression t est définie inductivement par :

$$\text{Pre}(t) = \begin{cases} t & \text{si } t \text{ est constant} \\ \text{Pre}(t_1)\ldots\text{Pre}(t_n)f & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

Question 9. La forme préfixe est-elle l'image miroir de la forme postfixe ?

Question 10. Refaire la question 7 dans le cas où la liste représente un mot sous forme préfixe.
Indication : on pourra écrire une fonction récursive qui prend en argument une `int*char list` et renvoie l'arbre syntaxique associé au début de la liste, ainsi que le reste de la liste.

3 Expressions rationnelles sur un alphabet X

Retour à l'interprétation sémantique, et à des choses que vous connaissez : on veut manipuler des expressions rationnelles. On va donc utiliser les symboles suivants :

- Symboles d'arité 0 : $X \cup \{\varepsilon, \varnothing\}$;
- Symboles d'arité 1 : $\{*\}$;
- Symboles d'arité 2 : $\{., +\}$;
- Pas de variables.

Question 11. Écrivez un type `regexp` pour les expressions rationnelles.
On prendra `char` pour l'ensemble X des lettres.

Question 12. Écrire une fonction `avide : regexp -> bool` qui détermine si ε appartient au langage.

Question 13. Écrire une fonction `arden : regexp -> regexp -> regexp` qui étant donné deux expressions A et B, avec $\varepsilon \notin A$, résout l'équation $X = A.X + B$ Si $\varepsilon \in A$, on renverra une erreur.

Question 14. On considère le système d'équations suivant, d'inconnues les langages $X_1, \ldots, X_n$:

$$\begin{cases} X_1 &= A_{1,1}X_1 + A_{1,2}X_2 + \ldots + A_{1,n}X_n + B_1 \\ X_2 &= A_{2,1}X_1 + A_{2,2}X_2 + \ldots + A_{2,n}X_n + B_2 \\ \vdots & \\ X_n &= A_{n,1}X_1 + A_{n,2}X_2 + \ldots + A_{n,n}X_n + B_n \end{cases}$$

où l'on suppose que pour tous i, j, $\varepsilon \notin A_{i,j}$.

Écrire une fonction `syst : regexp array array -> regexp array -> regexp array` qui prend en argument la matrice des $A_{i,j}$ et le vecteur des B_i, et résout le système.

Question 15. Comment la fonction précédente permet-elle de calculer le langage reconnu par un automate ?

Cela prouve algorithmiquement la seconde inclusion du théorème de Kleene : tout langage reconnaissable par automate fini est un langage rationnel.