Manipulation d'expressions et introduction à la compilation

1 Introduction

Dans cette séance de travaux pratiques, on s'intéressera à différentes manières de manipuler des expressions (principalement des expressions mathématiques). C'est une étape indispensable à la compilation, qui vise à transférer des données dans un format textuel en une suite d'instructions : il faut en effet reconstituer les expressions présentes dans le fichier source afin de pouvoir déterminer la meilleure façon de les évaluer. Durant la processus, le compilateur pourra être également amené à modifier les expressions afin de rendre leur évaluation plus efficace. Il faut donc un moyen de représenter algorithmiquement ces expressions.

Dans un premier temps, nous étudierons une représentation arborescente d'expressions (limitées aux flottants et aux quatre opérations courantes). Ces solutions présentant quelques difficultés au niveau de la gestion de la mémoire, nous étudierons une représentation alternative, sous forme de liste contenant la représentation postfixe de l'arbre (une solution utilisée par certaines machines et certains langages de programmation, dont le langage Postscript utilisé par les imprimantes) et verrons comment on peut passer d'une représentation à l'autre.

Enfin, nous regarderons comment passer d'une chaîne de caractères à une représentation machine de l'expression, première étape de la compilation, en effectuant, simultanément, une vérification de la syntaxe de l'expression.

2 Représentation arborescente

On envisage donc, tout d'abord, de représenter une expression mathématique sous la forme d'un arbre. On considère les quatre opérateurs courants (addition, soustraction, multiplication et division) et on définit un type permettant de désigner ces opérateurs :

```
type oper = Add | Sub | Mul | Div
```

Une expression mathématique peut être représentée par un arbre binaire dont le type serait

```
type expr = Leaf of float
          | Node of expr * oper * expr
```

1. Après avoir entré les définitions de type précédents, construire des arbres, que l'on désignera par les nom `a1`, `a2` et `a3`, correspondant aux expressions $2\times3+4$, $2\times(3+4)$ et $2-3-4$.

2. Écrire une fonction récursive `eval` de signature `expr -> float` qui prend en argument un arbre représentant une expression mathématique et retourne le résultat correspondant à son évaluation. On pourra envisager préalablement une fonction de signature `oper -> float -> float -> float` qui, à un opérateur, associe la fonction sur les flottants correspondante.

3. Vérifier que la fonction précédente donne les résultats attendus sur `a1`, `a2` et `a3`.

4. Proposer une fonction `print_expr` de signature `expr -> unit` qui prend en argument un arbre correspondant à une expression mathématique et affiche l'expression de façon lisible à l'écran. Par exemple, `a1` pourra donner `(2*(3+4))`. Peu importe s'il y a des parenthèses inutiles, mais l'expression affichée doit correspondre exactement à l'arbre ! On pourra commencer par écrire une fonction `print_oper` !

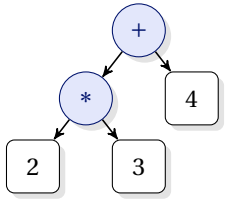
Pour illustrer les manipulations possibles sur les arbres, on se propose d'implémenter une règle : $a\times(b+c)=ab+ac$.

5. Proposer une fonction de signature `expr -> expr` qui identifie tous les endroits dans l'arbre où un nœud `Mul` a pour enfant à droite un nœud `Add`, et effectue le développement adéquat. L'arbre obtenu ne doit plus contenir de telle structure, mais l'expression doit, mathématiquement, être équivalente.

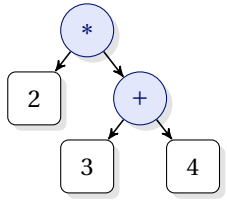
3 Représentation « polonaise inverse »

Une autre manière de représenter une expression mathématique est d'utiliser la représentation dite *polonaise inverse*, qui correspond à une écriture postfixe des arbres précédents (le nom « notation polonaise » est un hommage au logicien polonais Jan Łukasiewicz qui l'a présentée en 1924. on parle ici de notation inverse car la notation qu'il a initialement proposée était une notation *préfixe*).

Par exemple, l'expression mathématique $2\times3+4$, associé à l'arbre ci-dessous, correspond à la séquence en polonaise inverse `2 3 * 4 +`



Tandis que $2 \times (3+4)$, représentée dans l'arbre ci-dessus, est associée à la séquence `2 3 4 + *`.



On peut vouloir représenter en Caml une expression polonaise inverse sous la forme d'une liste. Seulement, on doit y trouver à la fois des flottants et des opérateurs, et les listes doivent contenir un unique type. Aussi définit-on un type `elem` de la sorte :

```
type elem = Val of float | Op of oper
```

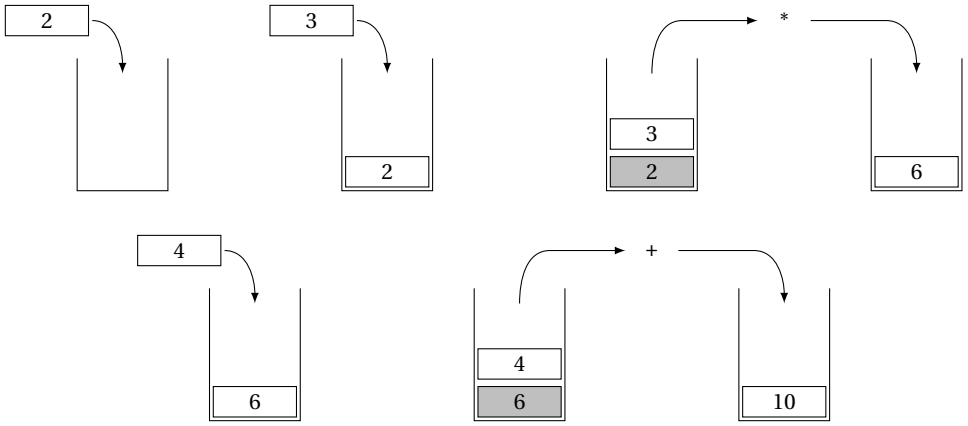
Par exemple, l'expression $2 \times 3 + 4$ pourra être représentée, en notation polonaise inverse, par la liste

```
[ Val 2.0; Val 3.0; Op Mul; Val 4.0; Op Add ]
```

Pour évaluer une expression en notation polonaise inverse, on se sert d'une pile. On prend les éléments de la liste un par un. Si c'est une valeur, on la place dans la pile. Si c'est un opérateur, on extrait les deux éléments au sommet de la pile, on applique l'opérateur à ces deux éléments, puis on replace le résultat dans la pile.

Une fois parvenu à la fin de la liste, l'expression est correcte si et seulement si il ne reste qu'un seul élément dans la pile. C'est le résultat de l'expression. Sinon, il y a une erreur de syntaxe dans l'expression.

Par exemple, voici comment est calculée l'expression `2 3 * 4 +` :



6. Écrire une fonction `eval_polinv` de signature `elem list -> float` qui prend en argument une liste représentant une expression mathématique en polonais inverse, et retourne son résultat (ou une erreur, peu importe laquelle, si l'expression n'est pas correcte).

On s'attachera suivre *scrupuleusement* la démarche présentée au-dessus, en utilisant les fonctions du module `Stack` ou, si vous le préférez, en utilisant une liste ou une référence vers une liste pour représenter la pile. On n'oubliera pas de vérifier qu'il n'y a effectivement plus qu'un élément dans la pile à la fin ! Par ailleurs, on rappelle que si l'on écrit

```
let a = Stack.pop pile and b = Stack.pop pile in ...
```

la syntaxe Caml ne garantit pas l'ordre dans lequel les définitions sont faites, et on ne sait donc pas si a était au-dessus de b dans la pile ou inversement ! On préférera donc écrire

```
let a = Stack.pop pile in
  let b = Stack.pop pile in ...
```

car dans ce cas, a est retiré *avant* b donc se trouvait *au-dessus* de b dans la pile.

7. Déterminer, grâce à la fonction `eval_polinv`, parmi les expressions suivantes, lesquelles sont correctes, et quel est le résultat de chacune :

- `2 3 4 + 5 * -`
- `2 3 4 5 * + -`
- `2 + 3 * 4 5 -`

Les réponses correctes sont, *dans le désordre*, `-21`, `expression incorrecte` et `-33`.

4 Changement de représentation

Nous allons à présent chercher à passer d'une représentation à l'autre.

8. Proposer une fonction `affiche_arbre_polinv` de signature `expr -> unit` qui prend en argument un arbre et *affiche à l'écran* la représentation polonaise inverse de l'expression.

9. En vous inspirant de la fonction précédente, écrire une fonction `polinv_of_arbre` de signature `expr -> elem list` qui prend en argument un arbre représentant une expression, et retourne une liste représentant cette même expression en polonais inverse. Par exemple, si l'on fournit `a1` en argument, le résultat devrait être

```
- : elem list = [Val 2.; Val 3.; Op Mul; Val 4.; Op Add]
```

On pourra, dans un premier temps, pour écrire la fonction `polinv_of_arbre`, utiliser l'opérateur de concaténation `@` si on le souhaite, puis chercher comment s'en passer afin de rendre la fonction plus efficace (on rappelle que la concaténation de deux listes a un coût proportionnel à leur longueur).

10. Proposer une fonction `arbre_of_polinv` de signature `elem list -> expr` qui prend en argument une liste représentant une expression mathématique en notation polonaise inverse et retourne l'arbre correspondant. On pourra s'inspirer de la méthode d'évaluation des expressions polonaises inverse et travailler là aussi avec une pile, en se posant la question des types d'objets à placerdans la pile.

5 Analyse d'une chaîne

Habituellement, lorsque l'on souhaite entrer une expression dans une calculatrice ou un ordinateur, on ne fournit pas un arbre ni [1] une séquence en polonaise inverse, mais plutôt une chaîne de caractères contenant une expression telle qu'on les manipule habituellement en mathématiques.

La première chose que devra effectuer une calculatrice ou un ordinateur est de transformer cette chaîne de caractère en une structure qu'elle pourra manipuler plus aisément, telle qu'un arbre (c'est ce que fait Caml lors de la compilation !).

Il existe des outils dédiés à cette transformation, mais dans le cas particulier qui nous intéresse, on peut envisager d'effectuer le travail « à la main ». Cette opération peut être décomposée en deux étapes :

- une première étape (*lexer*) vise à transformer la chaîne de caractère en la découpant et en la transformant en une liste de « jetons » (*tokens*) ; dans le cas qui nous intéresse, on obtiendra une séquence de flottants et d'opérateurs
- une seconde étape analyse, à l'aide d'une *grammaire* cette liste et la transforme, si elle représente une expression valide, en un objet informatique représentant l'expression (on travaillera ici avec la représentation arborescente).

Dans un premier temps, on ne considère que des chaînes de caractères contenant des expressions mathématiques constituées de flottants (constitués de chiffres, de la lettre `'e'` et du symbole `'.'`), supposés tous positifs ou nuls, et des quatre opérateurs [2] (`'+'`, `'-'`, `'*'` et `'/'`).

Pour représenter la liste intermédiaire, on pourra utiliser une `elem list`, comme précédemment (mais attention, la liste ne représente pas une expression en notation polonaise inverse !). On pourra considérer que toute séquence de caractères autres que `'+'`, `'-'`, `'*'` et `'/'` (lesquels seront directement convertis en un opérateur) correspond à l'écriture d'un unique flottant. On rappelle que l'on peut convertir une chaîne de caractères en flottant avec la fonction `float_of_string`.

1. excepté dans le cas particulier des calculatrices haut de gamme HP dans leur mode « original » de fonctionnement.
2. On travaille toujours avec des flottants, mais on n'utilisera pas les notations OCaml pour les opérateurs dans la chaîne, par souci de simplicité.

Ainsi, on veut :

- convertir `"4.5+6*5.2"` en `[ Val 4.5; Op Add; Val 6.0; Op Mul; Val 5.2 ]` ;
- convertir `"-4.5/3.7"` en `[ Op Sub; Val 4.5; Op Div; Val 3.7 ]` (cette expression ne sera cependant pas valable dans un premier temps car le `-` qui y figure n'est pas l'opérateur binaire de soustraction, cette erreur sera identifiée à la seconde étape) ;
- convertir `"4.5+*3.7"` en `[ Val 4.5; Op Add; Op Mul; Val 3.7 ]` (cette expression n'est pas non plus correcte, mais l'erreur de syntaxe ne sera identifiée qu'à l'étape suivante) ;
- lever une exception avec la chaîne `"4.5.7+2.5"`, car la séquence de caractères `"4.5.7` n'est pas un flottant valide.

11. Proposer une fonction `lexer` de signature `string -> elem list` réalisant cette première étape.

A présent, on cherche à transformer cette liste en un arbre représentant l'exception, et en identifiant les erreurs de syntaxe que l'on n'a pas encore détectées. Pour ce faire, on définit tout d'abord une *grammaire* qui décrit ce qu'est une expression.

Une *expression* peut être décrite comme un ensemble de *termes*, séparés par des `+` et des `-`. Un *terme* est un ensemble de *facteurs* séparés par des `*` et des `/`. Un facteur est, pour l'instant, simplement une valeur flottante. Cela peut se résumer par les règles suivantes :

```
# facteur    :: flottant
# terme      :: facteur [ (`*`,`/`) facteur ]*
# expression :: terme   [ (`+`,`-`) terme ]*
```

Les n-uplets entre parenthèses désignent ici une « union de possibilités », les crochets suivis de `*` « un nombre quelconque (pouvant être nul) de répétitions de ».

Par exemple, `expression` peut corresponsdre à `terme`, `terme` `` `+` `` `terme` ou bien encore à `terme` `` `+` `` `terme` `` `+` `` `terme` `` `-` `` `terme` `` `+` `` `terme`.

On propose une fonction `read_fact` de signature `elem list -> expr * elem_list` prenant en argument une (partie de la) liste d'éléments retournées par la première étape, listant un facteur au début de la liste, et retournant un objet de type `expr` correspondant au facteur lu, et le reste de la liste :

```
# exception SyntaxError

# let read_fact = function
    | Val x::q -> Leaf x, q
    | _ -> raise SyntaxError
```

12. Proposer une fonction `read_term` de signature `elem list -> expr * elem_list` lisant un terme au début de la liste. On utilisera la fonction `read_fact` pour lire un facteur.

Par exemple :

```
# read_term [ Val 1.; Op Add; Val 2.; Op Mul; Val 3.; Op Sub; Val 4. ];;
- : expr * 'a list = (Leaf 1.,
                      [Op Add; Val 2.; Op Mul; Val 3.; Op Sub; Val 4.])

# read_term [ Val 1.; Op Mul; Val 2.; Op Add; Val 3.; Op Sub; Val 4. ];;
- : expr * 'a list = (Node (Leaf 1., Mul, Leaf 2.),
                      [Op Add; Val 3.; Op Sub; Val 4.])

# read_term [ Val 1.; Op Div; Val 2.; Op Div; Val 3.; Op Sub; Val 4. ];;
- : expr * 'a list = (Node (Node (Leaf 1., Div, Leaf 2.), Div, Leaf 3.),
                      [Op Sub; Val 4.])
```

On prendra garde à ce dernier exemple : il est important que l'arbre corresponde bien à $(1/2)/3$ et non à $1/(2/3)$ pour que cela corresponde bien aux lois mathématiques usuelles !

13. Proposer une fonction `read_expr` de signature `elem list -> expr * elem_list` lisant une expression au début de la liste.

14. En déduire une fonction `compile` de signature `string -> expr` qui transforme une chaîne de caractères en l'arbre représentant l'expression mathématique correspondante. Une erreur doit être levée si elle n'est pas correcte (en particulier, la liste doit être vide à l'issue de la seconde étape !).

On se propose à présent d'améliorer un peu notre fonction, en ajoutant successivement les flottants négatifs (via un `-` unaire), les parenthèses, et les noms de variables.

Pour le `-` unaire, le changement n'est pas bien compliqué, il suffit de modifier légèrement facteur :

```
# facteur    :: flottant | `-` facteur
```

Le `|` dans la définition précédente indique que pour lire un facteur, on peut soit extraire de la tête de la liste un `Val ...`, soit un `Op Sub` suivi d'un facteur (qui peut également, à nouveau, commencer par un `Op Sub`). Pour éviter d'alourdir la définition de l'arbre, on pourra simplement construire le noeud avec un `0.0` dans le fils gauche.

Pour les parenthèses, c'est un peu plus délicat. L'idée de base est de redéfinir le facteur de la façon suivante :

```
# facteur    :: flottant | `-` facteur | `(` expression `)`
```

C'est le *seul* changement qu'il faut effectuer, mais il est important.

- Il faut ajouter[3] deux cas `Open` et `Close` parmi les cas du type `elem`.
- Il faut ensuite modifier `lexer` pour qu'il identifie ces jetons supplémentaires.
- Il n'est plus possible de définir successivement les fonctions `read_...`, car elles sont dorénavant mutuellement récursives ! On devra donc rassembler *toutes* ces fonctions dans un seul et unique **`let rec`**.
- On a un cas d'erreur de syntaxe supplémentaire à gérer via `read_fact` : si après avoir extrait l'expression on ne trouve pas un `Close`, l'expression est incorrecte, et sinon, il faut retirer ce `Close` de la liste retournée.

15. Implémenter ces modifications pour pouvoir tenir compte des parenthèses.

Et enfin, on peut vouloir gérer des *noms* dans les expressions, par exemple pour une chaîne `"2.5*x+mango/3.7"`.

16. Réfléchir aux modifications à effectuer dans le type `expr`, le type `elem`, la fonction `lexer` et la fonction `read_fact` pour obtenir le comportement souhaité.

Les plus véloces et acharnés d'entre vous pourront réfléchir à la manière d'inclure également des appels de fonctions. On supposera que tous les noms de fonctions sont regroupés dans une **`string list`** (toute séquence de caractères qui n'y figure pas est un nom ne désignant pas une fonction). Vous êtes libre de choisir si vous préférez la syntaxe OCaml (sans parenthèses) ou C/Python (avec des parenthèses) pour les appels de fonctions dans la chaîne.

3. On revalidera l'ensemble des fonctions après la modification du type pour éviter les confusions de type.