

PROGRAMMATION FONCTIONNELLE

RÉÉCRITURE ET INTERPRÉTATION

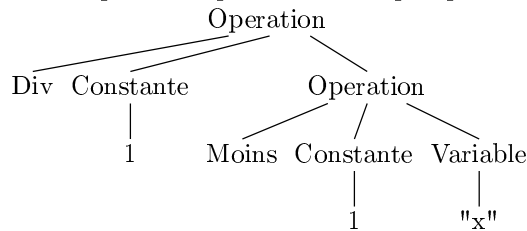
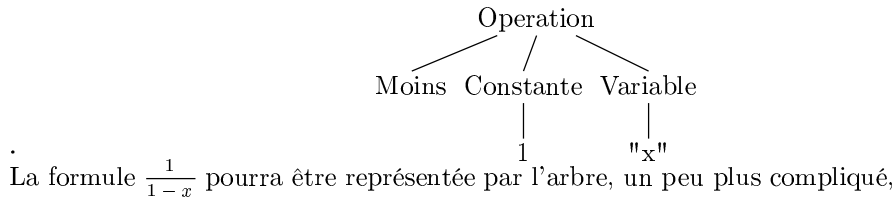
Il existe de nombreuses manières de représenter les expressions arithmétiques. Ce que nous écrivions spontanément $1 - x$ pourrait aussi bien être noté “soustraction à 1 de x ” ou encore “je pose 1, je pose x et j’applique la soustraction des deux derniers éléments”. En fait, ce n’est que depuis environ un siècle que nous écrivions quelque chose comme $1 - x$ et non pas “la valeur de la variable soustraite de 1”.

Si la notation $1 - x$ est simple et paraît de nos jours la plus évidente, votre ordinateur ou votre calculatrice ont été fabriqués pour comprendre plutôt d’autres formulations. Du coup, lorsque vous écrivez une formule arithmétique ou logique dans un programme ou dans votre calculatrice, une phase de compilation ou d’interprétation va être nécessaire pour traduire tout cela en quelque chose de manipulable directement par l’ordinateur.

L’objectif de ce TD est de regarder quelques-unes des techniques impliquées dans ces transformations. Pour ce faire, une fois de plus, nous allons parler d’arbres et de listes.

Arbre de syntaxe abstraite

Lors de manipulations sur un langage, on emploie systématiquement ce qu’on appelle des *arbres de syntaxe abstraite*. Vous avez déjà vu des arbres de syntaxe abstraite pour les formules de calculs ensemblistes, il en existe aussi pour les expressions arithmétiques et, en fait, il est possible d’en inventer pour presque tous les langages, OCaml, Java et le français y compris. Pour le moment, contentons-nous de travailler sur les formules arithmétiques. Ainsi, la formule simple $1 - x$ pourra être représentée par l’arbre



Afin de manipuler ces arbres en OCaml, pour ce TD, nous allons employer le type `expression`, défini par :

```
type operateur_binaire = Plus
                        | Fois
                        | Moins
                        | Div

type expression = Constante of int
                | Variable of string
                | Operation of operateur_binaire * expression * expression
```

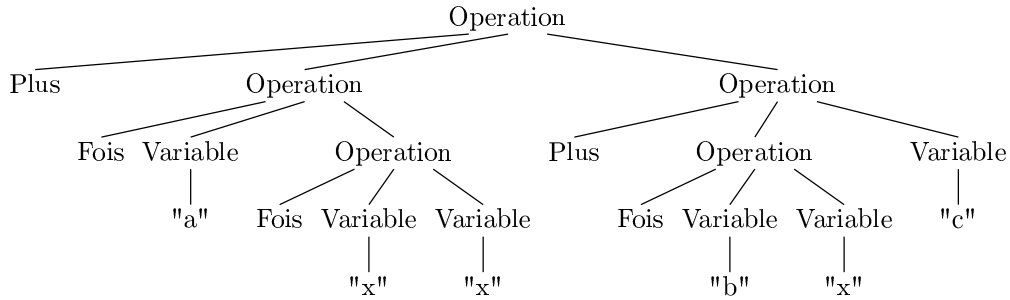
Ainsi, à l’aide de ce type, l’arbre de $1 - x$ précédent sera noté

```
Operation (Moins (Constante 1), (Variable "x"))
```

et l’arbre de $\frac{1}{1-x}$ sera noté

Operation (Div, (Constante 1), (Operation (Moins, (Constante 1), (Variable "x")))))

De même, la formule $a \cdot x^2 + b \cdot x + c$ sera représentée par l'arbre



ou encore par la valeur OCaml

```
Operation (Plus, Operation (Fois, Variable "a", Operation (Fois, Variable "x",
Variable "x") ), Operation (Plus, Operation (Fois, Variable "b", Variable "x"),
Variable "c") )
```

Donnons tout de suite un nom à ces exemples

```
# let test_1 = Operation (Div, (Constante 1), (Operation (Moins, (Constante 1),
(Variable "x"))));;
val test_1 : expression = Operation (Div, Constante 1, Operation (Moins,
Constante 1, Variable "x"))

# let test_2 = Operation (Plus, Operation (Fois, Variable "a", Operation (Fois,
Variable "x", Variable "x") ), Operation (Plus, Operation (Fois, Variable "b",
Variable "x"), Variable "c") );;
val test_2 : expression = Operation (Plus, Operation (Fois, Variable "a",
Operation (Fois, Variable "x", Variable "x") ), Operation (Plus, Operation (Fois,
Variable "b", Variable "x"), Variable "c") )
```

Exercice 1. Écrire une fonction qui, à partir d'une *expression*, retrouve la formule de base sous forme d'une chaîne de caractères lisible par un être humain.

Exemple :

```
# string_of_expression test_1;;
- : string = "1 / ( 1 - x )"

# string_of_expression test_2;;
- : string = "a * ( x * x ) + ( ( b * x ) + c)"
```

Ne vous inquiétez pas si votre chaîne de caractères contient trop de parenthèses. C'est plus gênant si elle n'en contient pas assez.

Rappelons que la concaténation de chaînes de caractères se fait avec l'opérateur `^`.

Écriture préfixe

Nous allons maintenant nous intéresser à l'écriture *préfixée* des expressions arithmétiques, c'est-à-dire une notation dans laquelle les opérateurs apparaissent avant les opérandes et non pas entre les opérandes, comme on en a l'habitude. Cette convention est employée sur un certain nombre de calculatrices et dans le langage de programmation Lisp. Ainsi, sous forme préfixée,

- la formule $1 - x$ s'écrit `- 1 x`

- la formule $\frac{1}{1-x}$ s'écrit comme la suite d'instructions `/ 1 - 1 x`
- la formule $a \cdot x^2 + b \cdot x + c$ peut s'écrire, par exemple, comme la suite d'instructions `+ × a × x x + × b x c`.

Pour représenter une expression de manière préfixée, nous allons nous servir du type suivant :

```
type instruction = Const of int
                | Var   of string
                | Op    of operateur_binaire
```

Exercice 2. Écrire une fonction qui transforme une valeur de type `expression` en une liste d'instructions sous forme préfixe.

Exemple :

```
# prefixe_of_expression (Div (Constante 1, Variable "x"));
- : instruction list = [Op Div; Const 1; Var "x"]

# prefixe_of_expression test_1;;
- : instruction list = [Op Div; Const 1; Op Moins; Const 1; Var "x"]

# prefixe_of_expression test_2;;
- : instruction list = [Op Plus; Op Fois; Var "a"; Op Fois; Var "x"; Var "x"; Op
Plus; Op Fois; Var "b"; Var "x"; Var "c"]
```

Exercice 3. Que fait l'extrait suivant ? Dans quels cas est lancée l'exception `Expression_incorrecte` ?

```
type raison_de_l_erreur = Liste_trop_longue | Liste_trop_courte

exception Expression_incorrecte of raison_de_l_erreur

let expression_of_prefixe expression =
  let rec aux = function
    | (Const c)::t -> (Constante c, t)
    | (Var v)::t   -> (Variable v, t)
    | (Op o)::t   ->
      let (e_gauche, t_gauche) = aux t in
      let (e_droite, t_droite) = aux t_gauche in
      (Operation (o, e_gauche, e_droite), t_droite)
    | [] -> raise (Expression_incorrecte Liste_trop_courte)
  in
  match aux expression with
  | (e, []) -> e
  | (_, _::_) -> raise (Expression_incorrecte Liste_trop_longue);;
```

Écriture postfixe

De la même manière qu'on peut noter une expression arithmétique en écriture préfixée (cf. juste au-dessus) ou infixée (l'écriture habituelle), on peut employer une notation postfixée, dans laquelle les opérateurs apparaissent après les opérandes. C'est la convention notamment sur les calculatrices HP ou dans le langage de programmation Forth.

Ainsi, sous forme postfixée,

- la formule $1 - x$ s'écrit $1 x -$
- la formule $\frac{1}{1-x}$ s'écrit comme la suite d'instructions $1 1 x - /$
- la formule $a \cdot x^2 + b \cdot x + c$ peut s'écrire, par exemple, comme la suite d'instructions $a x x \times \times b x \times c + +$.

Exercice 4. * Écrire une fonction qui transforme une valeur de type `expression` en la liste d'instructions postfixée correspondante.

Exemple :

```
# postfixe_of_expression (Div (Constante 1, Variable "x"));;
- : instruction list = [Const 1; Var "x"; Op Div]

# postfixe_of_expression test_1;;
- : instruction list = [Const 1; Const 1; Var "x"; Op Moins; Op Div]

# postfixe_of_expression test_2;;
- : instruction list = [Var "a"; Var "x"; Var "x"; Op Fois; Op Fois; Var "b"; Var
"x"; Op Fois; Var "c"; Op Plus; Op Plus]
```

Exercice 5. * Écrire une fonction qui transforme une liste d'instructions sous forme postfixe en l'expression correspondante.

En cas de problème avec la liste d'instructions, soulevez une exception `Expression_incorrecte`.

Exemple :

```
# expression_of_postfixe [Const 1; Var "x"; Op Div];;
- : expression = Operation (Div, Constante 1, Variable "x")

# expression_of_postfixe [Const 1; Const 1; Var "x"; Op Moins; Op Div];;
- : expression = (*... un gros truc égal à test_1...*)

# expression_of_postfixe [Var "a"; Var "x"; Var "x"; Op Fois; Op Fois; Var "b";
Var "x"; Op Fois; Var "c"; Op Plus; Op Plus];;
- : expression = (*... un gros truc égal à test_2...*)
```

À partir de là...

Exercice 6. Quel est l'intérêt des notations préfixe ou postfixe par rapport à la notation infixe (qui est, rappelons, la notation habituelle, dans laquelle on écrit les opérateurs au milieu de l'expression) ?

Exercice 7. * Écrire une fonction qui transforme une liste d'instructions sous forme préfixe en une liste sous forme postfixe.

Exemple :

```
# postfixe_of_prefixe [Op Div; Const 1; Op Moins; Const 1; Var "x"];;
- : instruction list = [Const 1; Const 1; Var "x"; Op Moins; Op Div]
```

Exercice 8. Comment pourrait-on faire pour éviter de mélanger par accident les listes en notation préfixe et les listes en notation postfixe ?