cours3

November 3, 2020

1 Programmation fonctionnelle

1.1 Semaine 3 : Programmer avec des arbres

Etienne Lozes - Université Nice Sophia Antipolis - 2019

---

[1]:
```
let () = ()
```

```
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,…";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads
```

1.2 Les données structurées et la programmation fonctionnelle

Au second cours, nous avons étudié les listes chaînées, un premier exemple de structures de données récursives. Nous allons approfondir dans ce cours la notion de structure de données récursives et aborder les **arbres**.

Mais avant cela, nous allons voir comment structurer des données à l'aide des **enregistrements** et des **énumérations**.

1.3 Les enregistrements

Reprennons l'exemple des points colorés de l'algorithme de recommendation du cours précédent. Nous avions représenté chaque point coloré par un triple `(x,y,c)`. Si `p` est un point coloré, on peut copier sa couleur dans une variable `c` en écrivant `let (_,_,c) = p in ...`.

Ce n'est pas toujours très pratique, en particulier il faut se souvenir que la couleur est à la troisième position. Quand on passe à des données structurées de plus grandes dimensions, on préfère utiliser des **enregistrements** (*record* en anglais) pour grouper les données ensemble.

[2]:
```
type point_colore = {x:int; y:int; c:string}
```

[2]:
```
type point_colore = { x : int; y : int; c : string; }
```

[3]:
```
let p1 = {c="rouge"; x=0; y=0}
```

[3]:
```
val p1 : point_colore = {x = 0; y = 0; c = "rouge"}
```

[4]:
```
let p2 = {y= 0; x=22; c="bleu"}
```

[4]:
```
val p2 : point_colore = {x = 22; y = 0; c = "bleu"}
```

L'étape de déclaration du type est essentielle: elle permet à OCaml de savoir que `x`, `y` et `c` sont des identifiants de **champs** du type `point_colore`. Elle permet ainsi à OCaml d'éviter les définitions de point colorés incomplètes ou mal typées

[5]:
```
let p3 = {x=0; y=0}
```

```
File "[5]", line 1, characters 9-19:
1 | let p3 = {x=0; y=0}
             ^^^^^^^^^^
Error: Some record fields are undefined: c
```

[6]:
```
let p4 = {x=0; y=0; c=0}
```

```
File "[6]", line 1, characters 22-23:
1 | let p4 = {x=0; y=0; c=0}
                          ^
Error: This expression has type int but an expression was expected of type
         string
```

[7]:
```
let p5 = {x=0; y=0; color="bleu"}
```

```
File "[7]", line 1, characters 20-25:
1 | let p5 = {x=0; y=0; color="bleu"}
                        ^^^^^
Error: Unbound record field color
```

On accède à un champs d'un enregistrement avec la notation `.`

[8]:
```
p1.c
```

[8]:
```
- : string = "rouge"
```

Prenons un autre exemple: créons un enregistrement pour représenter un nombre complexe

[9]:
```
type complexe = {re:float; im:float}
```

[9]:
```
type complexe = { re : float; im : float; }
```

[10]:
```
let plus_complexe z1 z2 = {
  re = z1.re +. z2.re;
  im = z1.im +. z2.im
}
```

[10]:
```
val plus_complexe : complexe -> complexe -> complexe = <fun>
```

On peut créer une copie d'un enregistrement `r` en modifiant certains champs seulement, et en laissant les autres inchangés. On écrit `{r with ...}` en précisant les valeurs des champs qui changent.

[11]:
```
let conjugue z = {z with im = -. z.im }
```

[11]:
```
val conjugue : complexe -> complexe = <fun>
```

[12]:
```
conjugue {re=13.; im=5.}
```

[12]:
```
- : complexe = {re = 13.; im = -5.}
```

On peut déconstruire un tuple dans une définition ou un motif et récupérer ainsi certains champs.

[13]:
```
let norme {re=r; im=i} = r ** 2. +. i ** 2.
```

[13]:
```
val norme : complexe -> float = <fun>
```

[14]:
```
(* on peut utiliser les champs comme des noms de variables *)
let norme {re; im} = re ** 2. +. im ** 2.
```

[14]:
```
val norme : complexe -> float = <fun>
```

[15]:
```
let est_rouge_ou_bleu p = match p with
| {c = "rouge"; x = _ ; y = _ } -> true
| {c = "bleu"; x = _ ;  y = _ } -> true
| _ -> false
```

[15]:
```
val est_rouge_ou_bleu : point_colore -> bool = <fun>
```

On n'est pas obligé de lister tous les champs dans un motif.

De plus, dans une définition par cas, on peut regrouper plusieurs motifs et leur associer un seul et même traitement.

La définition précédente peut donc se simplifier.

[16]:
```
let est_rouge_ou_bleu p = match p with
| {c = "rouge"} | {c = "bleu"}  -> true
| _ -> false
```

[16]:
```
val est_rouge_ou_bleu : point_colore -> bool = <fun>
```

1.4 Bien choisir les noms des champs

Souvent vous aurez plusieurs types d'enregistrements qui coexisteront dans votre programme.

Pour qu'ils vivent en bonne entente, il vaut mieux que leurs noms de champs soient bien distincts (au moins en attendant de voir les modules pour mettre un peu d'ordre dans tout ça).

Par exemple, si on avait réutilisé les noms `x` et `y` dans la définition du type `complexe`, on aurait perturbé l'usage des points colorés, de type `{x:int; y:int; c:string}`.

[17]:
```
type complexe2 = {x:float; y:float}
```

[17]:
```
type complexe2 = { x : float; y : float; }
```

[18]:
```
let p6 = {x = 1; y = 1; c = "jaune"}   (* le champs c sauve la mise ici *)
```

[18]:
```
val p6 : point_colore = {x = 1; y = 1; c = "jaune"}
```

[19]:
```
let ramene_a_0 p = {p with x=0; y=0} (* sans le champs c, OCaml infère p:
 ↪complexe *)
```

```
File "[19]", line 1, characters 29-30:
1 | let ramene_a_0 p = {p with x=0; y=0} (* sans le champs c, OCaml infère p:
 ↪complexe *)
```

```
                                        ^
Error: This expression has type int but an expression was expected of type
         float
```

Lorsque l'inférence de type d'OCaml n'est pas satisfaisante, on peut déclarer le type que l'on attend pour une variable ou une expression. OCaml ne l'acceptera pas les yeux fermés, mais il en tiendra compte.

[20]:
```
let app g x : int = g x (* je demande à ce que app g x soit de type int *)
```

[20]:
```
val app : ('a -> int) -> 'a -> int = <fun>
```

[21]:
```
let app g (x:int) = g x  (* je demande à ce que x soit de type int *)
```

[21]:
```
val app : (int -> 'a) -> int -> 'a = <fun>
```

[22]:
```
(* je demande à OCaml d'inférer un type à ramene_a_0 tel que le type de p soit␣
 ↪point_colore *)
let ramene_a_0 (p:point_colore) = {p with x=0;y=0}
```

[22]:
```
val ramene_a_0 : point_colore -> point_colore = <fun>
```

[23]:
```
(* je demande à OCaml d'inférer un type à ramene_a_0 tel que le type de la␣
 ↪valeur retournée soit point_colore *)
let ramene_a_0 p = ({p with x=0;y=0} : point_colore)
```

[23]:
```
val ramene_a_0 : point_colore -> point_colore = <fun>
```

1.5 Rajouter des opérateurs au langage

Supposons que l'on veuille calculer la somme de deux nombres complexes `z1` et `z2`. On peut appeler la fonction `plus_complex` décrite précédemment et écrire `plus_complexe z1 z2`. C'est tout de même un peu lourd, et OCaml propose mieux: introduire un nouvel opérateur dans le langage!

[24]:
```
let (++) z1 z2 = {re = z1.re +. z2.re; im= z1.im +. z2.im}
```

[24]:
```
val ( ++ ) : complexe -> complexe -> complexe = <fun>
```

[25]:
```
let z1 = {re=1.; im=0.}
let z2 = {re=0.; im=1.}
let z3 = z1 ++ z2
```

```
[25]: val z1 : complexe = {re = 1.; im = 0.}

[25]: val z2 : complexe = {re = 0.; im = 1.}

[25]: val z3 : complexe = {re = 1.; im = 1.}
```

En interne, `++` n'est rien d'autre qu'une fonction qui prend deux paramètres (c'est la fonction `plus_complexe` de tout à l'heure). Lorsqu'on parle de la fonction toute seule, on écrit `(++)`, et lorsqu'on l'applique à ses deux arguments, on l'écrit en **notation infixe**, entre les deux arguments.

Autre exemple: l'opérateur `^` ne permet pas de concaténer des caractères et des chaînes de caractères

[26]:
```
'a' ^ "bcd"
```

```
File "[26]", line 1, characters 0-3:
1 | 'a' ^ "bcd"
    ^^^
Error: This expression has type char but an expression was expected of type
         string
```

Qu'à cela ne tienne, définissons de nouveaux opérateurs pour faire des concaténation "mixtes" qui convertissent le caractère en chaîne de caractères (fonction `Char.escaped`)

[27]:
```
let (^::) c s = (Char.escaped c) ^ s
let (^@) s c = s ^ (Char.escaped c)
let chaine1 = 'a' ^:: "bcd"
let chaine2 = "abc" ^@ 'd'
```

```
[27]: val ( ^:: ) : char -> string -> string = <fun>

[27]: val ( ^@ ) : string -> char -> string = <fun>

[27]: val chaine1 : string = "abcd"

[27]: val chaine2 : string = "abcd"
```

Les possibilités sont infinies… vous pourrez vous amuser à étendre le langage comme bon vous semble, il faudra seulement prendre garde à utiliser des caractères spéciaux pour définir vos opérateurs infixes, et à respecter quelques règles.

On peut aussi définir des opérateurs unaires **préfixe**, comme `-` ou `-.`.

Leur nom devra commencer par `!`, `?`, ou `~` et ne comporter que des caractères spéciaux.

[28]:
```
let (~~) z = { z with im = -. z.im } (* un opérateur unaire pour le conjugé *)
```

[28]:
```
val ( ~~ ) : complexe -> complexe = <fun>
```

[29]:
```
let i =  {re=0.; im=1.} in ~~ (i ++ i)
```

[29]:
```
- : complexe = {re = 0.; im = -2.}
```

OCaml applique des règles d'associativité sur les opérateurs définis par l'utilisateur qui se basent sur le premier caractère spécial utilisé. Par exemple, `***` est prioritaire sur `++`, qui est prioritaire sur `====`, etc.

Enfin, on a le droit de redéfinir des opérateurs "prédéfinis". Pour ne pas les perdre définitivement, mieux vaut tout de même redéfinir localement et non globalement (en pratique, on fait ça plutôt avec des modules).

[30]:
```
let (+) {re = x1; im = y1} {re = x2; im = y2} =  {
  re = x1 +. x2;
  im = y1 +. y2;
}
in let ( * ) {re=x1; im=y1} {re=x2; im=y2} =  {
  re = x1 *. x2 -. y1 *. y2;
  im = x1 *. y2 +. x2 *. y1
}
in let i = {re = 0.; im = 1.}
in  i + i * i + i   (* égal à  i + (i*i) + i *)
```

[30]:
```
- : complexe = {re = -1.; im = 2.}
```

1.6 Les types énumérés

Certaines valeurs peuvent parfois se présenter sous plusieurs formes. Par exemple, une couleur peut être définie par

- un nom de couleur (ex: `"red"`, `"green"`, `"blue"`, …)
- trois entiers 8 bits en mode RGB (ex: `(255, 255, 0)` pour jaune=rouge+vert)
- un entier 32 bits en mode CYMK (cyan, magenta, yellow, black)
- trois entiers 8 bits en mode HSL (hue, saturation, lightness)

- …

Si l'on veut écrire une fonction qui prend en argument une couleur, il peut être intéressant de prévoir tous ces cas. C'est ce que permet de faire un **type énuméré**.

[31]:
```
type couleur =
| ParNom of string
| RGB of int * int * int
| CYMK of int
| HSL of int * int * int
```

[31]:
```
type couleur =
    ParNom of string
  | RGB of int * int * int
  | CYMK of int
  | HSL of int * int * int
```

Cette déclaration de type introduit des **constructeurs** pour le type `couleur`. Chaque valeur du type `couleur` est "étiquetée" par un constructeur.

Règle d'or de la syntaxe des constructeurs

> Un contructeur de type commence nécessairement par une majuscule

Comme leur nom l'indique, les constructeurs permettent de construire des valeurs du type `couleur`.

[32]:
```
let noir1 = ParNom "black"
```

[32]:
```
val noir1 : couleur = ParNom "black"
```

[33]:
```
let noir2 = RGB(0,0,0)  (* aucune couleur mélangée *)
```

[33]:
```
val noir2 : couleur = RGB (0, 0, 0)
```

Attention, `noir1` est de type `couleur`, mais pas du type `string`.

[34]:
```
String.length noir1
```

```
File "[34]", line 1, characters 14-19:
1 | String.length noir1
                  ^^^^^
Error: This expression has type couleur
       but an expression was expected of type string
```

C'est comme si chaque constructeur était une fonction "injective" vers le type `couleur`. En math, on noterait

$$ParNom : string \hookrightarrow couleur$$

Voyons d'autres exemples de définition de couleur.

[35]:
```
let noir3 = CYMK 0x000000ff  (* 8 bits de poids faible = 255 *)
```

[35]:
```
val noir3 : couleur = CYMK 255
```

[36]:
```
let noir4 = HSL(42,111,0)  (* luminosité = 0 *)
```

[36]:
```
val noir4 : couleur = HSL (42, 111, 0)
```

On peut maintenant écrire une fonction qui prend un argument de type `couleur` et qui renvoie un résultat qui dépend du contructeur de type qui a été utilisé.

[103]:
```
let est_noir c = match c with
| ParNom "black" | RGB(0, 0, 0) | HSL(_,_,0) -> true
| CYMK x when x mod 256 = 255 -> true
| _ -> false
```

[103]:
```
val est_noir : couleur -> bool = <fun>
```

On pourrait redéfinir les booléens avec des constructeurs "sans arguments", ou plutôt de type `unit`.

[38]:
```
type booleens =
| Vrai of unit
| Faux of unit
```

[38]:
```
type booleens = Vrai of unit | Faux of unit
```

[39]:
```
Vrai () = Faux ()  (* faux, car même si () = (), l'étiquette est différente *)
```

[39]:
```
- : bool = false
```

OCaml prévoit cet usage particulier des constructeurs "sans argument", bien utile pour décrire un ensemble fini de valeurs. On les écrit tout simplement "sans argument".

[40]:
```
type jour =  Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
```

[40]:
```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
```

1.7 Le type `'a option`

Le type `'a option` est un **type prédéfini**. Cependant, pour expliquer comment il fonctionne, rien de tel que le redéfinir.

[41]:
```
type 'a option_ =
| None
| Some of 'a
```

[41]:
```
type 'a option_ = None | Some of 'a
```

Ce type énuméré est couramment utilisé pour la valeur renvoyée par une fonction qui n'est pas définie pour tous les paramètres.

[42]:
```
let racine_carree x = if x < 0. then None else Some (sqrt x)
```

[42]:
```
val racine_carree : float -> float option_ = <fun>
```

C'est aussi un type qui permet d'avoir la valeur `None` comme "bouche-trou" dans une structure de données

[43]:
```
type etat_civil = {
  annee_naissance : int;
  annee_deces : int option
}
```

[43]:
```
type etat_civil = { annee_naissance : int; annee_deces : int option; }
```

[44]:
```
let jacques = {annee_naissance = 1932 ; annee_deces = Some 2019}
```

[44]:
```
val jacques : etat_civil = {annee_naissance = 1932; annee_deces = Some 2019}
```

[45]:
```
let valery = {annee_naissance = 1926; annee_deces = None}
```

[45]:
```
val valery : etat_civil = {annee_naissance = 1926; annee_deces = None}
```

1.8 Les types polymorphes

Le type `'a option` est paramétrique en la variable de type `'a`. C'est un type **polymorphe**, de même que le type `'a list`.

Comme on vient de le voir, il est possible de définir un type polymorphe, il suffit d'introduire la variable de type juste après le mot-clé `type`. On peut par exemple définir une "matrice" de valeurs de type `'a` comme une liste de listes de valeurs de type `'a`.

[46]:
```
type 'a matrice = 'a list list  (* pour l'exemple, en pratique à éviter *)
```

[46]:
```
type 'a matrice = 'a list list
```

Si on veut une matrice "infinie dénombrable", on pourra la représenter par une fonction

[47]:
```
type 'a matrice = int * int -> 'a  (* m (i,j) renvoie la valeur ligne i et␣
↪colonne j *)
```

[47]:
```
type 'a matrice = int * int -> 'a
```

Si on veut une définition générale de matrice, finie ou infinie, on peut considérer que les indices de lignes et de colonnes sont pris dans des "ensembles" qui sont eux-mêmes des paramètres.

[48]:
```
type ('a, 'ligne, 'colonne) matrice =  'ligne * 'colonne  ->  'a
```

[48]:
```
type ('a, 'ligne, 'colonne) matrice = 'ligne * 'colonne -> 'a
```

Notez que si le type paramétré que l'on définit prend plusieurs paramètres, on les met entre parenthèses.

1.9 Les types récursifs

Un type récursif est un type qui est défini à partir de lui-même. Le type `'a list` est un type récursif: pour s'en convaincre, essayons de le redéfinir.

[49]:
```
type 'a liste =
| Empty                        (* Empty       <=> []     *)
| Cons of 'a * 'a liste         (* Cons(hd,tl) <=> hd::tl *)
```

[49]:
```
type 'a liste = Empty | Cons of 'a * 'a liste
```

On a bien besoin du type `'a liste` pour décrire les arguments de `Cons`!

Par contre, contrairement aux définitions de fonctions récursives, inutile de rajouter le mot-clé `rec` pour une définition de type récursive.

[50]:
```
let hd l = match l with
| Empty -> invalid_arg "liste vide!"
| Cons(x,tl) -> x
```

[50]:
```
val hd : 'a liste -> 'a = <fun>
```

On peut aussi définir simultanément plusieurs types mutuellement récursifs. Le mot-clé essentiel pour des définitions mutuellement récursives est `and`.

L'exemple ci-dessous est une autre définition des listes chaînées proche de celle qu'on ferait en C.

[51]:
```
type 'a cell = {hd:'a; tl:'a liste}
and  'a liste = 'a cell option    (* <- il faut lire ('a cell) option *)
```

[51]:
```
type 'a cell = { hd : 'a; tl : 'a liste; }
and 'a liste = 'a cell option
```

[52]:
```
let empty = None
```

[52]:
```
val empty : 'a option_ = None
```

[53]:
```
let cons hd tl = Some {hd=hd; tl=tl}
```

[53]:
```
val cons : 'a -> 'a liste -> 'a cell option_ = <fun>
```

Notez dans le type inferré de `cons` qu'OCaml écrit indifféremment `'a liste` et `'a cell option`. Ces deux expressions de type s'évaluent en un seul et même type.

`'a liste` $\Leftrightarrow$ `'a cell option`

Notez aussi le type de `empty`: `'a option` $\Rightarrow$ `'a liste`.

Il n'est pas très commode du point de vue du programmeur d'avoir plusieurs expressions de type qui parlent du même type. Ceci ne se produit pas avec les types récursifs basés sur des constructeurs et une énumération, aussi on privilégie en général les définitions de type récursives basées sur des types énumérés.

D'ailleurs, pour des raisons d'efficacité de l'inférence de type, OCaml ne permet pas par défaut d'écrire tous les types récursifs que l'on voudrait.

[54]:
```
type 'a liste = ('a * 'a liste) option
```

```
File "[54]", line 1, characters 0-38:
1 | type 'a liste = ('a * 'a liste) option
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Error: The type abbreviation liste is cyclic
```

Il est possible de demander à OCaml d'utiliser l'algorithme d'inférence de type qui sait traiter les types récursifs comme celui-ci. Pour cela, on doit lancer `ocaml` ou `ocamlc` avec l'option `-rectypes`. Cette option est d'un usage assez rare, car avoir des expressions de type différentes qui décrivent le même type introduit beaucoup de confusion. C'est cependant un aspect du langage qui fait l'objet de discussions au sein de l'équipe de développeurs d'OCaml, rien ne dit que les choses n'évolueront pas dans le futur.

1.10 Les arbres d'expression arithmétique

Définissons maintenant un type pour représenter les arbres d'expressions arithmétiques que l'on rencontre par exemple en **calcul formel** (voir cours de Python de L1!).

[55]:
```
type arbre =
  | Feuille of feuille
  | Noeud of operateur * arbre * arbre

and feuille =
  | Const of int
  | Var of string

and operateur = PLUS | MULT | SUB | DIV
```

[55]:
```
type arbre = Feuille of feuille | Noeud of operateur * arbre * arbre
and feuille = Const of int | Var of string
and operateur = PLUS | MULT | SUB | DIV
```

Note : pour des questions de présentation, on a utilisé le `and` des définitions mutuellement récursives, mais on aurait pu s'en passer en définissant les types dans l'ordre inverse, "bottom-up"…

1.11 Construire un arbre d'expression arithmétique

Définissons les arbres des expressions "$3 \times x$" et "$(3 \times x) + y$"

[56]:
```
let arbre1 =
  Noeud (MULT,
         Feuille(Const 3),
         Feuille(Var "x"))
```

[56]:
```
val arbre1 : arbre = Noeud (MULT, Feuille (Const 3), Feuille (Var "x"))
```

[57]:
```
let arbre2 = Noeud(PLUS,arbre1, Feuille(Var "y"))
```

[57]:
```
val arbre2 : arbre =
  Noeud (PLUS, Noeud (MULT, Feuille (Const 3), Feuille (Var "x")),
   Feuille (Var "y"))
```

Certes, c'est un peu lourd de devoir écrire tous ces constructeurs… on pourrait améliorer cela en introduisant de nouveaux opérateurs dans le langage (exercice!). Malgré tout, le résultat affiché n'est pas très lisible.

1.12 Afficher un arbre d'expression arithmétique

On va définir une fonction `string_of_arbre : arbre -> string` qui renvoie une chaîne de caractères qui permette d'afficher "joliment" l'arbre qu'elle représente.

En anglais, on parle souvent de **pretty-printing**.

Pour définir `string_of_arbre`, on va procéder par récurrence et par études de cas, comme pour les listes de la semaine dernière, sauf que les motifs et le raisonnement se compliquent un peu…

[58]:
```
let rec string_of_arbre = function
| Feuille(Const n) -> string_of_int n
| Feuille(Var x) -> x
| Noeud(op, ag, ad) ->
    let strg = string_of_arbre ag
    and strd = string_of_arbre ad
    in let strop = match op with
    | PLUS -> "+" | SUB -> "-" | MULT -> "*" | DIV -> "/"
    in "(" ^ strg ^ strop ^ strd ^ ")"
```

[58]:
```
val string_of_arbre : arbre -> string = <fun>
```

[59]:
```
string_of_arbre arbre2
```

[59]:
```
- : string = "((3*x)+y)"
```

On pourrait économiser quelques parenthèses, à condition de se creuser un peu la tête… c'est pourquoi je vous le laisse en exercice!

Au lieu d'économiser les parenthèses, on va surtout chercher à rendre le code de notre fonction `string_of_arbre` plus lisible. C'est un exercice toujours utile à effectuer, un code lisible est un code sur lequel on peut plus facilement découvrir des erreurs.

Dans le cas présent, on a trois types, à savoir `arbre`, `feuille` et `operateur`; on a donc intérêt à avoir trois fonctions de *pretty-printing*.

[60]:
```
let rec string_of_arbre = function
| Feuille f -> string_of_feuille f
| Noeud(op, ag, ad) ->
    let strg = string_of_arbre ag
    and strd = string_of_arbre ad
    and strop = string_of_operateur op
    in "(" ^ strg ^ strop ^ strd ^ ")"

and string_of_feuille = function
| Const n -> string_of_int n
| Var x -> x

and string_of_operateur = function
| PLUS -> "+" | SUB -> "-" | MULT -> "*" | DIV -> "/"
```

[60]:
```
val string_of_arbre : arbre -> string = <fun>
val string_of_feuille : feuille -> string = <fun>
val string_of_operateur : operateur -> string = <fun>
```

[61]:
```
string_of_arbre arbre2
```

[61]:
```
- : string = "((3*x)+y)"
```

Petite astuce pour le travail dans un toplevel : on peut “installer” un *pretty-printer* pour un type `t`, de sorte que le toplevel adopte notre afficheur maison pour toutes les valeurs de type `t`.

[62]:
```
arbre2  (* l'arbre avec tous les constructeurs... MIAM! *)
```

[62]:
```
- : arbre =
Noeud (PLUS, Noeud (MULT, Feuille (Const 3), Feuille (Var "x")),
 Feuille (Var "y"))
```

Pour installer un pretty printer, on commence par définir une fonction à deux arguments: - un **formateur** qui est une sorte d’écrivain à qui on peut envoyer des ordres (on en reparlera) - un deuxième argument qui est la valeur du type à afficher.

On demande ensuite au formateur d’afficher une chaîne de caractères qui correspond à celle calculée par notre pretty-printer.

[63]:
```
let pp_arbre fmt arbre = Format.fprintf fmt "%s" (string_of_arbre arbre)
```

[63]:
```
val pp_arbre : Format.formatter -> arbre -> unit = <fun>
```

Une fois la fonction `pp_arbre` définie, on procède à son installation en appelant la macro `#install_printer`

[64]:
```
#install_printer pp_arbre
```

[65]:
```
arbre2
```

[65]:
```
- : arbre = ((3*x)+y)
```

1.13 Evaluer un arbre d'expression sans variables

On voudrait maintenant programmer une petite calculette qui prend en entrée une expression arithmétique sans variable et qui calcule l'entier correspondant. Par exemple, si `arbre` est l'arbre de l'expression $1 + (2 \times 3)$, `eval arbre` renverra 7.

[66]:
```
let rec eval = function
| Feuille(Const n) -> n
| Noeud(binop, ag, ad) -> begin
   let n1 = eval ag and n2 = eval ad in
   match binop with
   | PLUS -> n1 + n2
   | SUB -> n1 - n2
   | MULT -> n1 * n2
   | DIV -> n1 / n2
  end
| _ -> invalid_arg "contient des variables!"
```

[66]:
```
val eval : arbre -> int = <fun>
```

[67]:
```
let arbre = Noeud(PLUS,Feuille(Const 1),Noeud(MULT,Feuille(Const␣
↪2),Feuille(Const 3)))
in eval arbre
```

[67]:
```
- : int = 7
```

À nouveau, essayons d'améliorer la lisibilité de notre fonction `eval` en introduisant une fonction d'évaluation `eval_op` pour les opérateurs. Mais que doit renvoyer `eval_op PLUS`?

Tout simplement, `eval_op PLUS` renvoie la fonction d'addition `(+) : int -> int -> int`!

[102]:
```
let eval_op = function
| PLUS -> (+) | SUB -> (-) | MULT -> ( * )  | DIV -> (/)

let rec eval = function
| Feuille(Const n) -> n
| Noeud(op, ag, ad) -> eval_op op (eval ag) (eval ad)
```

```
| _ -> invalid_arg "contient des variables!"
```

[102]:
```
val eval_op : operateur -> int -> int -> int = <fun>
```

[102]:
```
val eval : arbre/2 -> int = <fun>
```

[69]:
```
eval (Noeud(PLUS,Feuille(Const 1),Feuille(Const 2)))  (*1+2*)
```

[69]:
```
- : int = 3
```

1.14 Evaluer une expression avec des variables (1)

On voudrait maintenant évaluer une expression en remplaçant les variables qui y apparaissent par leur valeur. Par exemple, si l'on remplace x par 1 et y par 2, l'expression $(3 \times x) + y$ s'évalue en l'entier 5.

Il y a deux données à prendre en compte

- l'arbre d'expression proprement dit, et
- l'environnement d'interprétation, autrement dit les valeurs que l'on souhaite donner aux variables.

Pour représenter l'environnement, on va utiliser une **liste associative** formée de couple "clé-valeur".

[70]:
```
let env = [ ("x", 1); ("y", 2)]
```

[70]:
```
val env : (string * int) list = [("x", 1); ("y", 2)]
```

Cette représentation n'est pas la seule qu'on aurait pu choisir. En Python on aurait utilisé sans doute un dictionnaire, autrement dit une **table de hachage**. Les tables de hachage sont plus efficaces que les listes associatives quand il commence à y avoir beaucoup de clés. Cependant, pour utiliser une table de hachage en OCaml il faut faire des mutations, on en reparlera la semaine prochaine…

1.15 Quelques mots sur les listes associatives

Les listes associatives sont d'usage très courant en OCaml, à tel point qu'une partie du module `List` leur est consacrée. Regardons maintenant le fonctionnement de quelques une des fonctions sur les listes associatives.

[71]:
```
List.mem_assoc "x" [ ("x", 1); ("y", 2)] (* teste si "x" est une clé de env *)
```

[71]:
```
- : bool = true
```

[72]:
```
List.assoc "x" [ ("x", 1); ("y", 2)] (* renvoie la valeur associée à la clé "x"␣
↪*)
```

[72]:
```
- : int = 1
```

[73]:
```
List.assoc "z" [ ("x", 1); ("y", 2)]  (* lève l'exception Not_found si la clé␣
↪n'est pas présente *)
```

```
Exception: Not_found.
Raised at file "list.ml", line 191, characters 16-25
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

[74]:
```
(* s'il y a plusieurs associations possibles, c'est celle en début de liste qui␣
↪est retenue *)
List.assoc "x" [ ("x", 3); ("x", 1); ("y", 2)]
```

[74]:
```
- : int = 3
```

[75]:
```
(* pour créer une nouvelle liste associative en retirant une association *)
List.remove_assoc "y" [ ("x", 1); ("y", 2); ("z", 3)]
```

[75]:
```
- : (string * int) list = [("x", 1); ("z", 3)]
```

[76]:
```
List.assoc_opt "x" env, (* comme List.assoc, mais renvoie une option *)
List.assoc_opt "z" env
```

[76]:
```
- : int option * int option = (Some 1, None)
```

1.16 Evaluer une expression avec des variables (2)

Nous sommes maintenant prêts à rajouter le cas manquant dans notre fonction `eval`

[77]:
```
let rec eval env = function   (* <- notez le paramètre env *)
| Feuille(Const n) -> n
| Feuille(Var x) -> List.assoc x env
| Noeud(binop, ag, ad) -> eval_op binop (eval env ag) (eval env ad)
```

[77]:
```
val eval : (string * int) list -> arbre -> int = <fun>
```

[78]:
```
arbre1, env, eval env arbre1
```

[78]:
```
- : arbre * (string * int) list * int = ((3*x), [("x", 1); ("y", 2)], 3)
```

[79]:
```
eval [] arbre1  (* si une variable n'est pas trouvée, l'exception Not_found␣
↪stoppe l'évaluation *)
```

```
Exception: Not_found.
Raised at file "list.ml", line 191, characters 16-25
Called from file "[77]", line 4, characters 54-67
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

On pourrait faire plus propre en utilisant `List.mem_assoc` ou `List.assoc_opt`… ou en rattrapant l'exception, on en reparlera.

1.17 Des arbres d'arité arbitraire et polymorphes

On va maintenant s'intéresser aux arbres d'arité arbitraire, dans lesquels le nombre de fils de chaque noeud n'est pas toujours le même. On va ajouter des "étiquettes" sur ces arbres à la fois sur les noeuds et sur les arètes. Nos arbres seront de plus polymorphes: les deux types d'étiquettes seront des paramètres.

[80]:
```
type ('info_noeud, 'info_arete) arbre =
| Noeud of 'info_noeud *  (('info_arete * ('info_noeud, 'info_arete) arbre)␣
↪list)
```

[80]:
```
type ('info_noeud, 'info_arete) arbre =
    Noeud of 'info_noeud *
      ('info_arete * ('info_noeud, 'info_arete) arbre) list
```

[81]:
```
let feuille x = Noeud(x, [])  (* une feuille est un noeud sans fils *)
```

[81]:
```
val feuille : 'a -> ('a, 'b) arbre = <fun>
```

[82]:
```
let fils x (Noeud(_ ,l)) = List.assoc x l (* renvoie le premier fils␣
↪d'étiquette d'arête x *)
```

```
[82]: val fils : 'a -> ('b, 'a) arbre -> ('b, 'a) arbre = <fun>
```

```
[83]: let arbre = Noeud(0, [('a', feuille 1); ('b', feuille 2)])
```

```
[83]: val arbre : (int, char) arbre =
        Noeud (0, [('a', Noeud (1, [])); ('b', Noeud (2, []))])
```

```
[84]: fils 'a' arbre
```

```
[84]: - : (int, char) arbre = Noeud (1, [])
```

1.18 Application : les *tries*

Une *trie**, aussi appelé arbre préfixe, est une structure de données utilisée pour représenter une fonction de domaine fini de type `string -> 'a`.

* le genre et la prononciation de ce "two-way anglicisme" ne fait pas l'unanimité.

Ci-dessus: une trie pour l'association `"to"`$\mapsto 7$, `"tea"`$\mapsto 3$, `"i"`$\mapsto 11$, ...

Étant donnée une clé `s:string`, on peut calculer la valeur associée en suivant la branche de l'arbre étiquettée par `s`. La valeur associée à `s` se trouve sur le noeud sur lequel la branche se termine.

Certains noeuds internes contiennent des valeurs, mais pas tous.

Pour représenter une trie, nous allons simplement utiliser un arbre, en prenant les caractères pour étiquetter les arètes

```
[85]: type 'a trie = ('a option, char) arbre
```

```
[85]: type 'a trie = ('a option, char) arbre
```

```
[86]: (* exemple : un trie avec une seule branche *)
      let trie1 : int trie = Noeud(None, ['t',Noeud(None,['o',Noeud(Some 7,[])])]) (*␣
       ↪"to"|-> 7 *)
```

```
[86]: val trie1 : int trie =
        Noeud (None, [('t', Noeud (None, [('o', Noeud (Some 7, []))]))])
```

Pour calculer la valeur associée à une clé, on va parcourir récursivement la trie.

À chaque étape, on doit lire un nouveau caractère de la clé. Ce caractère permet de trouver, à l'aide de la fonction `fils`, dans quel sous-trie continuer le parcours.

[87]:
```
let rec assoc_opt cle trie =
   match (String.length cle, trie) with
   | 0, Noeud(v,_) -> v
   | n, trie -> assoc_opt (String.sub cle 1 (n-1)) (fils cle.[0] trie)
   (* String.sub : on met à jour la clé pour la suite du parcours
      en retirant de la clé le caractère déjà lu *)
```

[87]:
```
val assoc_opt : string -> ('a, char) arbre -> 'a = <fun>
```

[88]:
```
assoc_opt "to" trie1
```

[88]:
```
- : int option = Some 7
```

Ce n'est pas idéal car `String.sub` fait une recopie de toute la chaîne à chaque fois. Pour éviter cela, je peux parcourir la chaîne de caractères `cle` avec une variable `i=0,1,...`, un peu comme une boucle for. Sauf que pour le moment je ne sais pas comment faire une boucle `for` en OCaml.

Solution: je crée une fonction auxiliaire `f` qui attend un entier `i` en paramètre; quand j'appelle `f` récursivement, il me suffit de lui passer `i+1` en paramètre. Je passe aussi à `f` le sous-trie dans lequel je suis en train de chercher, que je modifie à chaque appel récursif comme dans la première version. Une fois `f` écrite, il me suffit de l'appeler avec les bonnes valeurs initiales: `i`=0, et `strie`=`trie`, le `trie` tout entier du début.

[89]:
```
let assoc_opt cle trie =
   let n = String.length cle in
   let rec f i strie = match (i, strie) with  (* <- ma fonction auxiliaire *)
     | i, Noeud(v, _) when i = n -> v
     | _ -> f (i+1) (fils cle.[i] strie)
   in f 0 trie
```

[89]:
```
val assoc_opt : string -> ('a, char) arbre -> 'a = <fun>
```

[90]:
```
assoc_opt "to" trie1
```

[90]:
```
- : int option = Some 7
```