Algorithmique des structures de données arborescentes
Feuille d'exercices - Premiers Pas en OCaml

1 Introduction à OCaml

Nous nous baserons sur le polycopié de Marc Zeitoun :
https ://moodle.u-bordeaux.fr/pluginfile.php/14500/mod_resource/content/1/main.pdf

Vous pouvez faire les premiers exercices en mode interactif, soit en tapant `ocaml` ou `utop` dans un terminal, soit en utilisant `Try OCaml` sur internet.

Vous écrirez les premières fonctions en ouvrant un fichier `td1.ml` soit avec `Emacs`, soit avec `VSCode` (cf Chap2 "Environnement de développement" du polycopié).

1.1 Les expressions

Une **expression**[1] est un morceau de code que l'ordinateur peut interpréter, calculer, et dont il peut trouver le type et la valeur. En OCaml, tout ou presque est une expression (ex : un calcul arithmétique, une chaîne de caractère, un appel de fonction, un "if ... then ... else ...") et nous programmerons uniquement en construisant des expressions.

Pour toute expression écrite en OCaml, le compilateur va vérifier la cohérence des types de données apparaissant dans l'expression. En particulier, vous ne pouvez pas additionner (sans utiliser une fonction de conversion de type) des entiers et des réels. D'ailleurs il existe deux opérateurs distincts pour l'addition des entiers (`+`) et pour l'addition des réels (`+.`). Il en va de même pour les autres opérateurs arithmétiques. Il existe des fonctions de conversion de type prédéfinies en OCaml : `float_of_int`, `int_of_float`, `int_of_char`, `char_of_int`, ...

Exercice 1.1 Déterminer sur feuille le type et la valeur des expressions suivantes, puis tester vos réponses :

- `45 + 2*(27-11)`
- `25/4 + 2*9 + (int_of_char 'A')`.
- `2e3 +. 2.5 *. (float_of_int 4)`.
- `6 / 3*2`.

Le type booléen est noté `bool`. Les constantes de ce type sont `true` et `false`, les opérateurs logiques sont `&&` pour le "et", `||` pour le "ou", `not` pour le "non". Par exemple `x>y && not (x+y > z)` est une expression de type `bool`.

Exercice 1.2 Déterminer sur feuille le type et la valeur des expressions suivantes, puis tester vos réponses, sachant que `mod` désigne l'opérateur "modulo".

- `min 22 56 > 2*7`
- `not (6 / 3*2 = 1 || 4 * 3/2 = 5)`.
- `int_of_float(3.14 *. 2.) = 6`.
- `(11+13) mod 2 = 0 && 2*7 + 1 mod 2 = 1`.

1.2 Les liaisons

En OCaml on peut attribuer un nom à une expression, avec la syntaxe `let` nom = `expression`. Ce mécanisme s'appelle une `liaison`[2].

Ainsi si l'on écrit `let x= 2*3`, la valeur de l'expression à droite du signe '=', c'est-à-dire 6, est `liée` à l'identificateur x. Dans la suite du programme, x restera lié à la valeur 6 jusqu'à ce qu'une nouvelle liaison : `let x= ...` se substitue à la précédente.

[1] polycopié page 3
[2] polycopié page 5

On peut également définir des liaisons temporaires, dont la portée sera limitée à une expression : après la déclaration de la liaison, on ajoute alors le mot-clé `in` suivi de l'expression dans laquelle la liaison sera utilisée. De même que l'on peut définir plusieurs variables locales au début d'un bloc en langage C, on peut imbriquer plusieurs liaisons locales pour calculer une expression en OCaml. Par exemple `let pi = 3.1415 in let rayon = 5.0 in 2. *. pi *. rayon` .

ATTENTION : il y une différence importante entre les liaisons globlales et les liaisons locales.
Une liaison locale est une expression qui renvoie une valeur. Elle pourra être utilisée dans une autre expression (une expression conditionnelle par exemple), une fonction ou une autre liaison.
Une liaison globale **n'est pas une expression** et **ne renvoie pas de valeur**.
Ainsi on pourra écrire `let y = (let x=2 in 2*x)` car la partie entre parenthèses est une expression de type `int` et de valeur 4. Mais on ne pourra pas écrire `let x=2 in (let y=2*x)` car la partie entre parenthèses n'est pas une expression et OCaml indique une erreur de syntaxe.

Exercice 1.3

- Déterminer la valeur de l'expression `let x = 5.0 in x *. x +. 30.0` .
- Déterminer la valeur de l'expression `let x = 5 in let y = x+2 in x+3*y` .
- Déterminer la valeur de l'expression `let x = 14 in let y = x > 0 in y && x < 30`

1.3 Les fonctions.

De même on définit une **fonction**[3] en déclarant une liaison entre un nom de fonction et le code de la fonction. Plusieurs syntaxes sont possibles. Ainsi par exemple, la fonction qui prend en paramètre un entier `x` et un réel `y` et calcule leur somme (de type réel) peut s'écrire :

```
let f = fun x y -> (float_of_int x) +. y
```

ou de manière équivalente :

```
let f x y = (float_of_int x) +. y
```

Un appel de la fonction précédente s'écrira :

```
let z = (f 3 7.5)
```

ATTENTION : en OCaml il n'y a **pas de parenthèse ouvrante** entre le nom de la fonction et ses paramètres et il n'y a **pas de virgule** (juste un espace) entre les différents paramètres (voir polycopié, page 6 et 7).

Exercice 1.4

- Déterminer la valeur de l'expression `let f = fun x y -> x - y in f (f 1 2) 3` .
- Déterminer la valeur de l'expression `let cube x = x * x * x in (cube 2) + (cube 3)` .

Exercice 1.5 Écrire une fonction `pythagore a b c` qui prend en paramètre trois entiers a, b et c qui teste si $a^2 + b^2 = c^2$. On écrira d'abord une fonction `carre x` qui prend en paramètre un entier x et renvoie x^2.

Exercice 1.6 Ecrire une fonction `vector_norm x y` qui prend en paramètre deux nombres flottants (les coordonnées x et y d'un vecteur) et qui calcule la norme du vecteur, c'est-à-dire la racine carré de $x^2 + y^2$. La racine carré est une fonction prédéfinie en OCaml de type `sqrt: float -> float` .

Le type d'une fonction s'écrit en OCaml avec une flèche entre les types des différents arguments, s'il y en a plusieurs, et une flèche avant le type du résultat. OCaml détecte pour nous le type des expressions et en particulier le type des fonctions. Par exemple OCaml détecte que la fonction `let f x y = (float_of_int x) +. y` est de type `int -> float -> float` . Lorsque les types de certains paramètres ou du résultat de la fonction peuvent être quelconques, OCaml leur attribue des noms : `'a` , `'b` ,... Ainsi la fonction identité `let id = fun x -> x` est de type `'a -> 'a` .

Exercice 1.7 Déterminer le type des expressions suivantes

- fun x y -> x

[3] polycopié page 9

- fun x y -> y
- fun f -> fun x -> f (f x)
- fun f g -> fun x -> f (g x)

En OCaml, les fonctions sont des expressions comme les autres. En particulier, une fonction peut être argument d'une autre fonction, et une fonction peut retourner une autre fonction.

Exercice 1.8 Indiquez ce qui est calculé par l'expression suivante.

```
let square = fun x -> x * x in
    let compose = fun f g -> fun x -> f (g x) in
        compose square square
```

Exercice 1.9 Écrire une fonction `diff_between a b f` qui prend en paramètre deux entiers a et b et une fonctions f de type `int -> int` et qui retourne la différence entre la valeur de `f` en b et la valeur de f en a.

1.4 Conditions et filtrage

La syntaxe d'une expression conditionnelle est la syntaxe habituelle. Elle utilise les mot-clés `if`, `then` et `else`.

ATTENTION : en OCaml toute expression doit avoir un type bien défini. Une expression conditionnelle ne peut donc avoir qu'un seul type, **le même** pour la partie `then expression1` et pour la partie `else expression2`, sans quoi l'expression conditionnelle produira une **erreur de type**.

Exercice 1.10 Déterminer le type des expressions suivantes

- fun x y -> if x = y then x else y
- fun x y z -> if x = y then x else z
- fun x y z -> if x = y then 0 else z

Exercice 1.11 Ecrire une fonction `museum_price: int -> float` qui prend en paramètre un age (de type entier) et renvoie le prix d'entrer au musée en fonction de l'age. Ce prix est 6 euros si l'age est inférieur ou égale à 11 ans, 9 euros si l'age est compris entre 12 ans et 18 ans (compris), et 15 euros si l'age est strictement supérieur à 18 ans.

En plus des expressions conditionnelles, OCaml propose un mécanisme plus puissant pour tester une donnée : le **filtrage par motif**[4] (ou analyse par cas ou pattern-matching en anglais) qui permet d'effectuer une disjonction de cas. L'idée générale est d'énumérer tous les cas possibles pour la donnée filtrée et d'associer à chaque cas une expression. La syntaxe du pattern-matching en OCaml utilise le mots-clé `match` suivi de la donnée à filtrer suivie du mot-clé `with` et des différents motifs possibles séparés par des `|`.

Remarque : si vous n'avez pas énuméré tous les cas possibles, OCaml affichera le message "pattern matching non exhaustif ". On peut terminer par un dernier cas noté `_` qui signifie tous les autres cas.

Le filtrage permet de comparer une donnée à des "motifs". Le filtrage réussit s'il y a correspondance entre la donnée et un motif (au moins). S'il y a correspondance possible avec plusieurs motifs, c'est le premier trouvé qui est sélectionné, donc l'ordre d'écriture des motifs est important. Le motif peut comporter des constantes, qui doivent alors être présentes au même endroit dans la donnée filtrée. Le motif peut comporter de nouveaux identificateurs, qui seront alors localement liées à des parties de la données filtrée.

Exemple : dans l'expression `match (0,42) with (x,0) -> 2*x | (0,y)-> y/2 | (x,y) -> x+y` le filtrage du couple d'entiers `(0,42)` avec le motif `(0,y)` réussit, une liaison locale est créée entre le nom de variable `y` et la valeur 42. Le résultat est 21.

Exercice 1.12 Ecrire une fonction `helloworld` qui prend en paramètre une chaine de charactère indiquant une langue et renvoie "Bonjour" écrit dans cette langue. Cette fonction traitera le cas de

[4] polycopié pages 20,21

plusieurs langues différentes. Si la langue passée en paramètre ne fait pas partie des cas prévus, la fonction renverra "Je ne parle pas" concaténé avec la langue demandée. ■

Exercice 1.13 Ecrire une fonction `is_vowel: char -> bool` qui prend en paramètre un caractère c et renvoie `true` si et seulement si le cactère c est une voyelle. ■

1.5 Les fonctions récursives

Dans ce cours, pour répéter un traitement, nous n'utiliserons pas de boucles, mais des fonctions récursives. Une **fonction récursive**[5] est appelée dans sa propre définition. Pour écrire une fonction récursive, il est nécessaire de nommer la fonction et d'indiquer explicitement qu'elle est récursive par le mot-clé **rec**.

Exercice 1.14 Soit la fonction `fact` définie comme suit :

```
let rec fact n =
if n = 0 then 1 else n * fact (n-1)
```

Dessiner la pile des appels pour `fact 5`. ■

Exercice 1.15 Écrire une fonction `power a n` qui prend en paramètre deux entiers a et n et qui renvoie a à la puissance n si n est positif et un message d'erreur sinon (on pourra utiliser : `failwith "message"`) ■

Exercice 1.16 Écrire une fonction `first_integers_sum` qui calcule récursivement la somme des entiers de 0 à n (n étant supposé positif). ■

Exercice 1.17 Soient n, p deux entiers. On rappelle que $\text{pgcd}(n, 0) = n$ et que $\text{pgcd}(n, p) = \text{pgcd}(p, n \bmod p)$. Écrire une fonction `pgcd` calculant le pgcd de deux entiers. ■

Exercice 1.18 La suite de Fibonacci est définie par $u_0 = u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$.

1. Écrire une fonction `fib` calculant le n-ème terme de cette suite.

■

1.6 Les types produits et les types sommes

En OCaml on peut définir un nouveau type à l'aide du mot clé `type` suivi du nom du nouveau type. On peut définir un type **produit cartésien**[6] de types existants. L'opérateur pour construire un type produit cartésien est `*`. Les éléments d'un type produit cartésien sont des n-uplets.

Exemple : pour associer un code (de type int) à un charactère de type (char), on pourra créer un nouveau type "coding" et créer un exemple de couple de ce type :

```
# type coding = char * int ;;
type coding = char * int

# let code_of_A: coding = ('A', int_of_char('A'));;
val code_of_A : coding = ('A', 65)
```

Remarque : la syntaxe `(nom_de_var:nom_de_type)` vérifie que la variable est bien du type attendu.

On peut aussi définir des types avec un nombre fini de formes possibles, ce sont les **types sommes**[7]. La déclaration d'un type somme énumère toutes les formes possibles pour les données de ce type, séparées par des barres verticales. Chaque forme possible est identifiée par un nom, appelé son **constructeur**. Les noms de constructeurs doivent commmencer par une majuscule pour les distinguer des noms de variables. Les constructeurs serviront à la fois à construire des valeurs du type somme et à distinguer les différents cas possibles dans un filtrage par motif.

[5] Polycopié page 11
[6] polycopié page 15
[7] polycopié pages 16,17

Exemple : pour définir les jours de la semaine, on écrira le type :
`type day = Monday | Tuesday | Wednesday |Thursday | Friday | Saturday | Sunday`.

Les constructeurs peuvent aussi avoir comme arguments des types existants. Le constructeur sera alors suivi du mot-clé `of` puis de son ou ses arguments.

Exemple : pour définir les lignes de tram et de bus à Bordeaux, on écrira le type
`type tbm = Tram of char | Bus of int`. Les variables `let t = Tram('B')` et `let b = Bus(23)` sont du type tbm (On remarque que le mot-clé `of` qui apparait dans le type, est remplacé par des parenthèses quand on écrit les variables de ce type).
Un filtrage par motif d'une variable de `v` type `tbm` s'écrira

```
match v with
  Bus(i) -> ...
| Tram(c) -> ...
```

Exercice 1.19 Définir des types pour représenter les valeurs suivantes et donner pour chacun des exemples de variables de ce type.

1. Un type contenant trois valeurs, nommées TTrue, TFalse et Unkown.
2. L'ensemble des entiers, auquel on a ajouté une constante symbolique Pi.
3. Les points à coordonnées entières, en dimension 2 ou 3.

■

Exercice 1.20 On souhaite représenter les activités journalières d'un étudiant. Chaque activité est caractérisée par le nom de l'activité (de type `String`), l'heure de début et l'heure de fin de l'activité, qui seront de type float (par exemple $14h30$ sera représenté par 14.5) .

1. Définir un type `activity` pour représenter les activités journalières.
2. Ecrire une fonction (appelée "constructeur") `make_activity` qui prend en paramètre une chaîne de charactère et deux réels, et qui créé la l'activité correspondante.
   Créer l'activité "judo" ayant lieu de $17h30$ à $19h$.
3. Ecrire les fonctions (appelée "accesseurs") `activity_name`, `activity_start` et `activity_end` renvoyant respectivement le nom, l'heure de début et l'heure de fin d'une activité.
4. Ecrire une fonction `activity_lt` qui prend en paramètre deux activités et renvoie `true` si la première activité à lieu strictement avant la deuxième activité, et `false` sinon.
   Ecrire une fonction `activity_gt` qui prend en paramètre deux activités et renvoie `true` si la première activité à lieu strictement après la deuxième activité, et `false` sinon.
5. Ecrire une fonction `activity_disjoint` qui prend en paramètre deux activités et renvoie `true` si leurs créneaux horaires ne se chevauchent pas et `false` sinon.

■

1.7 Les types récursifs

On peut définir des **types récursifs**[8]. Un type récursif comporte au moins deux cas, donc ce sera nécessairement un type somme. L'un au moins des constructeurs aura parmi ses arguments (après le mot-clé `of`) le type récursif lui-même.

Par exemple on peut définir les listes contenant soit des entiers, soit des caractères de la façon suivante :

```
type int_or_char_list =
Nil
| ICell of int * int_or_char_list
| CCell of char * int_or_char_list ; ;
```

Exercice 1.21
1. Définir une liste de longueur 6 contenant les entiers 1,2,3 alternés avec les caractères 'a','b','c'.

[8] polycopié page 17, 18

2. Ecrire une fonction `contains` qui teste la liste contient le caractère c.
3. Ecrire une fonction `sum_list` qui fait la somme des entiers de la liste. Si elle ne contient aucun entier, le résultat sera 0.

■

Exercice 1.22 On veut écrire un type récursif pour modéliser tous les chemins d'un labyrinthe. Il y a trois types de chemins possibles : un mur, une sortie, un embranchement avec un chemin à gauche et un à droite.

1. Ecrire la définition du type `path` correspondant.
2. Ecrire une fonction `has_exit` qui prend en paramètre un chemin et qui renvoie `true` si et seulement si ce chemin comporte au moins une sortie.

■