TD1 : Remise en jambe OCaml et concepts utiles

2023-2024

OCaml est le langage que nous utiliserons durant ce cours. Le but de ce TD est de faire une remise en jambe et de servir d'aide-mémoire pratique.

Si vous n'êtes pas familiers de OCaml, vous pouvez trouver des tutoriaux ici : `http://OCaml.org/learn/`, ou, pour des interrogation précises, vous tourner vers la documentation: `http://caml.inria.fr/pub/docs/manual-OCaml/`. On va résumer ici quelques notions qui nous serons utiles avec quelques courts exercices qui abordent ces notions et syntaxes.

Télécharger l'archive du TP `TD1.tar.gz`, et décompressez-le.

Dans le cadre de ce cours, nous allons vous suggérer fortement de travailler avec `vs-code`. Nous allons avoir besoin d'une installation de `OCaml` qui n'est pas celle de base de `Debian` et le faire via `opam`. Pour mettre en place votre environnement de travail au CREMI, voici comment lancer `vs-code` avec le bon environnement : ouvrez un terminal, tapez '`export OPAMROOT=/opt/local/opam`', puis '`eval $(opam env)`, et enfin lancez `vs-code` via la commande '`code`'. Vous aurez à faire cela dès que vous voulez travailler sur ce cours. Il est évidemment possible d'ajouter les deux premières commandes à votre `.bashrc` si vous ne souhaitez pas les taper à chaque fois. Ces détails peuvent être retrouvés depuis la page du cours, ainsi que les instructions pour mettre en place l'environnement de travail sur votre machine personnelle.

Votre expérience de travail sera beaucoup plus agréable si vous utilisez l'extension `OCaml Platform` de `vs-code`, qui vous permettra notamment d'avoir vos fonctions annotées par leur type au fur et à mesure que vous codez. Si ce n'est pas déjà fait, installez cette extension. Vous pouvez éventuellement avoir besoin de faire une manipulation supplémentaire : si vous avez un message d'erreur parlant de `ocamlc` non trouvé, allez dans l'onglet de `OCaml Platform`, cliquez sur '`Select a sandbox`', puis sur '`Global`' dans le menu qui apparaîtra. Vous n'aurez à faire cela qu'une fois.

Une fois cela fait, vous devriez avoir un environnement fonctionnel.

1 Partie 1 : Tuto et référence

Les exercices de cette partie sont là pour vous réexpliquer les concepts dont nous aurons besoin pendant ce cours, vous montrer des exemples, etc. Il y a peu d'exercices construits et beaucoup à lire. Mais nous vous encourageons à tout tester.

Les exemples seront regroupés dans le dossier tuto du répertoire de ce TD, dans le fichier `tuto.ml`. Vous pouvez au choix le compiler (avec make) vers l'exécutable `tuto.out`, ou exécuter le fichier à la volée dans utop (ce fichier n'ayant besoin de rien d'autre).

Exercice 1: Utilisation et Compilation

Dans le cadre du cours, nous implémenterons un logiciel où nous avons automatisé la compilation via `dune` qui est un *builder* pour `OCaml`. Pour chaque TD, nous vous fournirons un projet `dune` déjà configuré. Pour plus de confort, un `Makefile` vous sera fourni pour lancer les commandes pertinentes. Taper '`make`' construira le projet et créera un lien vers l'exécutable (ou éventuellement plusieurs) que vous pourrez alors lancer. Un jeu de tests pour les fonctions à implémenter vous sera également fourni. Il se lancera grâce à '`make test`' (ce n'est pas le cas dans le cadre du présent tp).

La construction de la documentation se fera grâce à '`make doc`' et sera accessible via le fichier doc.html.

Pour compiler uniquement cette partie, vous pouvez taper '`make tuto.out`'.

Cependant, si vous souhaitez tester des fonctions/programmes séparément, vous pouvez lancer un *toplevel* qui vous permettra d'exécuter des fonctions séparément d'un programme complet (ce qui permettra notamment de ne pas avoir besoin de fonctions d'affichage pour tout). Au CREMI, `utop` est installé et est une version évoluée du toplevel standard de OCaml. Vous pouvez le lancer avec les modules du TD en utilisant la commande `dune utop` (évidemment, depuis la racine du projet).

Toutefois, la technique précédente ne fonctionne évidemment que si votre projet compile. Si cela n'est pas le cas et que vous voulez tester une fonction indépendamment du reste du projet, vous pouvez exécuter un bout de code en le sélectionnant, puis en tapant 'SHIFT+ENTER' (mais on déconseille cette méthode par rapport à la précédente, d'autant que si votre fonction fait appel à d'autres modules que le module courant, cela ne fonctionnera pas).

Vous aurez l'occasion de tester ces techniques dans les exercices qui suivent.

Il est évidemment possible de compiler un fichier via le compilateur ocaml (`ocamlc` pour le compilateur *bytecode* et `ocamlopt` pour le compilateur natif), mais il faut alors lier toutes les bibliothèques et fichiers manuellement et peut donc être fastidieux.

Exercice 2: Affichage

Un rapide point sur l'affichage : on le gérera avec la bibliothèque Format (`https://v2.ocaml.org/api/Format.html`) dont le `printf` fonctionne majoritairement comme vous en avez l'habitude, sauf pour les saut de lignes qui permettent des alignements en plus et ont des balises spéciales pour cela (les `@,` etc).

On ne vous demandera pas de gérer l'affichage vous-même, donc on ne s'étendra pas trop dessus. Si vous ressentez le besoin de faire de l'affichage de débogage, vous pourrez le faire comme vous en avez l'habitude, mais utilisez-bien le `printf` de Format : il est bufferisé, et si vous en utilisez un autre, l'affichage risque d'être dans le désordre par rapport au reste de l'affichage.

Exercice 3: Les bases du langage OCaml OCaml est un langage multi-paradigme, notamment fonctionnel, impératif, générique, modulaire et objet. Nous nous servirons à la fois du paradigme fonctionnel et du paradigme impératif du langage, dans le but de simplifier les codes et le type des fonctions que nous produirons (ainsi que leur efficacité), de même que nous utiliserons des modules et une partie du code qui vous sera fourni sera générique (mais vous-même n'aurez pas trop à vous préoccuper de cela).

Un programme est exécuté de haut en bas (comme en C, il peut comporter plusieurs fichiers, qui seront vus comme des modules (voir plus loin) : `dune` compilera tout le projet).

Expressions et types

Tous les éléments manipulés par le langage sont des `expressions` (on ne détaillera pas ici la notion, mais sachez que c'est la même notion que celle qu'on définira pour notre propre langage en cours.). Ces expressions sont toutes *typées*, et le compilateur refusera de compiler un code que ne serait pas correctement typé (ce qui est en réalité plus une aide qu'un inconvénient, cela empêchant des codes foireux d'être exécutés).

Par exemple, `4` ou `8-9+4/2` sont des expressions de type `int`, `4.2` ou if(x=4) then 44.9 else 55.1 sont des expressions de type `float`.

Il n'est pas nécessaire de préciser les types (le compilateur les infère et signale les inconsistances).

Requêtes

Le fichier est une suite de *requêtes* qui vont modifier la mémoire du programme. Une requête est de la forme `let name = expr`, où `name` est une chaîne de caractères, et `expr` une expression du langage.

Par exemple, `let toto = 4+2` mettra `6` à l'adresse `toto` de la mémoire (c'est assez mal dit, mais c'est l'idée).

On peut aussi évaluer une expression sans la retenir en écrivant `let _ = expr`, et préciser qu'on attend une action avec `let () = expr` (voir plus loin).

Il est possible, mais pas obligatoire de terminer une requête par `;;`. C'est nécessaire dans le toplevel, mais superflu dans un fichier compilé, à condition que la requête suivante commence par un `let`. En réalité `let` sert à la fois de début de requête et de fin de la requête précédente.

Fonctions

Le langage étant fonctionnel, les fonctions sont des expressions comme les autres (on peut également voir les expressions non fonctionnelles comme des fonctions sans argument). `fun` est le mot clé qui permet de définir des fonctions. `fun x -> x + 4` est une fonction de type `int -> int` (et aussi une expression).

La commande `let`, sans surprise, permet également de lier des fonctions à des noms : par exemple, `let f = fun x -> x + 1` définira la fonction f qui prend un entier x et renvoie $x+1$. Il existe une notation alternative pour la définition de fonction (celle principalement utilisée d'ailleurs): `let f x = x + 1`.

Pour utiliser une fonction, la syntaxe est différente de la plupart des langages impératifs que vous connaissez : la notation utilisée est la notation *polonaise* ou *préfixe*, et se fait sans parenthèse. Par exemple, `let n = (g 4 5)+ 6` signifie qu'on stocke dans n la valeur de $g(4,5)$ à laquelle on ajoute 6 (dans une notation plus proche du C). Les parenthèses ne servent que d'opérateurs de priorité. Par exemple `g (g 4 5)6` se lit comme $g(g(4,5),6)$ dans une notation plus proche du C.

L'énorme avantage de cette notation est qu'elle permet de faire de l'évaluation partielle. Par exemple, si g est une fonction de type `int -> int -> int`, alors `let h = g 4` est une fonction de type `int -> int` (et on aura `h x` qui sera la même chose que `g 4 x`).

Définition locale d'une expression

L'expression `let name = expr in` définit une expression localement. Cela est en général utile dans d'autres définitions. Attention, malgré la proximité syntaxique, un `let name = expr in` est bien une expression et non une requête (contrairement au `let name = expr`).

En particulier, les noms définis par un `let name = expr in` ne sont pas visibles en dehors de l'expression où ils sont définis.

Ainsi, avec `let f x = let succ x = x+1 in if(x < 0)then 0 else succ x`, on aura ensuite `f` d'accessible, mais pas `succ`.

Il y a également des expressions de type `unit` qui sont des actions qui sont exécutées lorsqu'elles sont définies, par exemple, l'affichage avec `Format.printf`, mais également tous les effets de bord du paradigme impératif.

Dans ce cas, il n'est pas nécessaire de leur donner un nom et on peut simplement écrire `let _ = Format.printf "Toto"`. On peut bien évidemment en utiliser dans d'autres expressions (avec `let ... in`), auquel cas l'action sera effectuée à chaque fois que l'expression est évaluée.

On peut alternativement utiliser le `;` pour séquencer deux expressions : `A ; B` sera la même chose que `let _ = A in B`.

Dans le toplevel, il est nécessaire d'utiliser `;;`, pour lancer l'évaluation d'un bloc de code. Dans les fichiers compilés, c'est inutile (mais ils n'empêchent pas la compilation).

Fonction argument d'une autre fonction

Comme dit avant, tous les objets manipulés par le programme sont des fonctions. Il est donc évidemment possible d'avoir des fonctions comme argument d'autres fonctions.

Par exemple `let g f x = (f x)+ 1` définit g qui prend en argument une fonction f de `'a -> int` et un `'a` x et renvoie le résultat de f x auquel est ajouté 1.

Ici, `'a` désigne un type quelconque : le langage supporte les fonctions polymorphes, et donc cette fonction pourra être utilisée sur n'importe qu'elle fonction dont le type de retour est un int et qui dispose d'un seul argument. Par exemple `g int_of_string "25"` est l'int 26.

Autre exemple `let disjunct f g b = if(b)then f else g` est une fonction bien définie (f et g doivent avoir le même type et peuvent accepter des arguments).

Enfin, on peut utiliser la notation `fun` pour définir des fonctions anonymes (ou autres utilisations) comme suit :

```
let res = (disjunct (fun x -> if(x < 0)then 0 else x+1)(fun x -> x-1)true)
3
```

est un int bien défini (ici, c'est 4).

On peut évidemment utiliser la syntaxe `let f = fun x -> x+1`.

Égalité

Point de syntaxe important : en OCaml, l'égalité *structurelle* se note `=` et pas `==` comme dans la plupart des langages. `==` existe, mais a un autre sens (égalité physique). De même, c'est l'inégalité structurelle est `<>` et `!=` l'inégalité physique.

En général, ce que vous voulez, c'est l'égalité structurelle. Pour vous en convaincre, vous pourrez remarquer que `let a = 1.1 in a = 1.1` est vrai alors que `let a = 1.1 in a == 1.1` est faux.

Priorité des opérateurs

Autre point : attention au parenthésage : l'associativité est à gauche, donc un terme du type `f x+1` sera interprété comme `(f x) + 1`.

Il est préférable de mettre des parenthèses pour désambiguïser les expressions. À noter cependant que le passage d'arguments à une fonction se fait sans parenthèses (ce qu'on écrirait `f(1,2)` en C s'écrit bien `f 1 2`).

Exercice

Définir une fonction `compo` qui prend en argument deux fonctions (quelconques) et renvoie leur composition (attention, ici, on ne peut pas ne pas mettre l'argument). Cet exercice peut être réalisé dans le toplevel.

Exercice 4: Récursion et fonctions mutuellement récursives

Un concept versatile et important en fonctionnel, c'est la récursion.

Quand on définit une fonction récursive, il faut cependant l'indiquer au compilateur avec le mot clé rec comme suit :

```
let rec toto x = if (x < 0)then 0 else 1 + (toto (x-1)).
```

Sans rec, le compilateur refusera de compiler (sauf si toto est déjà défini, puisque la redéfinition est autorisée, et c'est d'ailleurs pour cette raison qu'il faut indiquer rec ou non).

Dans le cadre du cours, on aura régulièrement besoin de définir des fonctions *mutuellement* récursives, c'est-à-dire qui s'appellent l'une l'autre. Pour cela, soit on définit une fonction à l'intérieur de l'autre (mais elle n'est alors pas visible à l'extérieur), par exemple:

```
let rec f x =
    let rec g y =
        if y < 0 then 0
        else f (y-1) + 2
    in
    if x < 0 then 1
    else g (x-2) * 3
```

Dans ce cas, seul `f` sera accessible à l'extérieur. Mais on peut également utiliser le mot clé `and` qui permet de définir simultanément deux noms. Avec lui, on peut définir la même chose que ci-dessus en ayant `f` et `g` visibles après leur déclaration:

```
let rec f x =
    if x < 0 then 1
    else g (x-2) * 3

and g y =
    if y < 0 then 0
    else f (y-1) + 2
```

Exercice 5: Les types et le pattern matching

En OCaml, les types de base sont les mêmes qu'en C (int, float, bool, char, string, etc). Il existe également des types complexes prédéfinis utilisables sans les définir, par exemple les tuples (`(2,'c')` est de type `int*char`), où les fonctions (`int → int*char → char` est le type des fonctions prenant un `int` et un `int*char` comme arguments et renvoyant un `char`). Il est tout à fait possible de les utiliser directement (et le compilateur y arrivera très bien).

On peut nommer des types avec la requête `type`. Par exemple `type toto = int*char`.

Mais la plus grande utilité de cette requête va être de pouvoir avoir des *types collection*, qui seront définis par des constructeurs.

On peut les définir comme suit:

```
type toto =
None
| One of int
| Two of int*char
```

qui définit un type qui peut contenir soit rien, soit un entier, soit une paire `int*char`.

On peut également définir des types récursifs (exemple `type tree = Leaf of int | Node of tree*tree`).

Note, les constructeurs doivent commencer par une majuscule et les types par une minuscule.

Ces types nous serons particulièrement utiles pour définir les programmes que nous manipulerons (l'arbre de syntaxe abstraite d'un programme sera un tel type).

Cela correspond au concept de types structurés dans d'autres langages et permet d'exprimer des ensembles de données arbitraires (avec une structure), et éventuellement hétérogènes. La plupart des types que nous manipulerons seront de ce genre, bien qu'il en existe d'autres.

Mais ces types seraient peu utilisables sans l'une des fonctionnalités les plus puissantes d'OCaml, le pattern matching: il permet de décapsuler le contenu d'un type et de renvoyer des valeurs différentes en fonction du cas dans lequel on est (tant que toutes les valeurs ont le même type, bien sûr).

Par exemple, la somme des éléments d'un arbre défini plus haut peut être définie comme suit:

```
let rec sum_of_tree t =
    match t with
    | Leaf a -> a
    | Node (t1,t2) -> (sum_of_tree t1) + (sum_of_tree t2)
```

On peut mettre des motifs plus complexes dans le pattern matching, ne pas nommer certains éléments si on n'en a pas besoin (avec _) et faire un cas «pour tout le reste» (avec _ également).

Par exemple:

```
let rec silly t =
    match t with
    | Node(Leaf _,_) -> 1
    | Node(_,t1) -> 1 + silly t1
    | _ -> failwith "arg"
```

On notera qu'ici mon cas par défaut est différent : il renvoie une exception qui terminera le programme si elle n'est pas rattrapée (cela peut être utile dans des cas qui ne sont pas censés arriver et où au lieu de mettre une valeur, on souhaite simplement que le programme plante). L'inférence de type n'aura aucun problème avec ça.

Il est toujours utile de faire cela car les pattern-matching sont censé être exhaustif (couvrir tous les cas).

Enfin, on peut mentionner le mot-clé when qui permet de restreindre le pattern à certaines valeurs seulement :

```
let toto a =
    match a with
    | Some a when a < 0 -> 0
    | Some a -> a
    | None -> 0
```

À noter que dans les deux cas précédents, il n'y a pas d'ambiguïté : le pattern matching est parcouru de haut en bas et s'arrête dès le premier match. L'ordre est donc important. Par ailleurs, les constructeurs None et Some sont les constructeurs du type option qui est un type générique permettant à une fonction de ne pas renvoyer de valeurs dans certains cas (None). Cela permet d'éviter d'utiliser des exceptions dans le flux normal du programme (et c'est très utile).

Exercice : Définissez un type représentant une direction de déplacement (selon les 4 points cardinaux) et une vitesse (ainsi qu'une valeur disant de rester sur place). Définissez aussi une fonction appliquant un objet de type précédent à une position (Autrement dit un tuple de type : `int * int`), mais qui ne fait rien si la vitesse est négative.

Exercice 6: Les listes: On utilisera beaucoup les listes dans notre projet. Vous pouvez regarder la page du manuel `https://ocaml.org/api/List.html` pour plus d'informations. On va simplement résumer les bases ici.

Une liste est une collection d'éléments linéairement ordonnée. On a accès à la tête de liste immédiatement. On peut facilement ajouter un élément en début de liste (temps constant), moins en queue (temps linéaire). La liste vide est nommée `[]`. `a::l` représente la liste `l` précédée de `a`. Par exemple `1::2::[4;5] = [1;2;4;5]`. On a les fonction `hd` et `tl` définie ainsi: `List.hd (a::l)= a` et `List.tl (a::l)` = l. Il est évidemment possible de faire du pattern matching sur les listes, par exemple :

```
    match l with
    | [] -> 0
    | a::t -> 1
```

qui renvoie 0 si la liste et vide et 1 sinon.

Cependant, la grande utilité du module `List` repose dans ses fonctions pour des opérations sur des listes. On va présenter les trois qui sont les plus versatiles et nous serons les plus utiles :

- `map` est une fonction qui va appliquer une fonction à tous les éléments d'une liste. Son type est `('a -> 'b)-> 'a list -> 'b list`. Par exemple

  ```
  List.map (fun x -> (int_of_string x) + 1) ["12";"0"
      ;"-2"]
  ```

  est la liste `[13;1;-1]`.

- `iter` est une fonction qui va appliquer récursivement à tous les éléments d'une liste (de gauche à droite) une fonction de type unit. Elle est de type `('a -> unit) -> 'a list -> unit`. Par exemple,

  ```
  List.iter (Printf.printf "\%d ++ ") [1;2;3]
  ```

  affichera "1 ++ 2 ++ 3 ++ ".

- `fold_left` est très similaire à la précédente, sauf qu'elle s'occupe de fonctions qui ont une valeur de retour qu'elle peuvent reprendre en argument. Son type est `('a -> 'b -> 'a)-> 'a -> 'b list -> 'a list`.

```
List.fold_left f a [b1;b2;b3]
```

  est la même chose que `f (f (f a b1)b2)b3`. a est la valeur d'initialisation, et la valeur de retour peut être vu comme un accumulateur. C'est une manière pratique d'itérer sur une liste sans utiliser de boucle. Par exemple

```
List.fold_left (fun x y -> x + y) 1 [2;3;4]
```

  renverra 10.

À noter que les trois fonctions précédentes sont récursives terminales (et donc ne peuvent générer de stack overflow). Il existe également `List.fold_right` qui applique la fonction dans l'ordre inverse à fold_left, mais elle n'est pas récursive terminale (donc à éviter si possible).

Exercice : Dans l'exercice précédent, appliquez la fonction de déplacement selon une direction et une vitesse à tous les points d'une liste de coordonnées (grâce à map). Déterminez ensuite le point le plus loin de (0,0) (grâce à fold_left).

Exercice 7: Les modules Vous aurez remarqué que dans le paragraphe précédent, j'ai préfixé map, iter, etc, par List. C'est parce que ces fonctions sont déclarées dans le module List. Les fichiers sont compilés vers des modules. Par exemples, après la compilation, les fonctions définies dans le fichier `toto.ml` seront dans le module `Toto`. À noter que le nom d'un module commence nécessairement par une majuscule. Pour appeler une fonction f du module `Toto`, il y a deux options : soit on fait un `open Toto` plus haut dans le fichier et on pourra ensuite utiliser les noms de `Toto` comme s'ils étaient définis dans le fichier courant ; soit on appelle simplement `Toto.f`. La deuxième méthode est en général préférée, sauf si un module est très utilisé (et encore) car elle permet d'être certain de ne pas avoir de conflit dans les noms, ou d'avoir des problème de redéfinition. Il est en effet possible de donner le même nom à des fonctions dans des modules différents. Si vous faites `open List`, vous aurez par exemple accès au nom `map`, mais si vous faites plus loin `let map = 3`, alors map désignera bien 3 et plus la fonction de List (et vice-versa si vous le faites dans l'autre sens).

À votre niveau, vous pouvez simplement considérer les modules comme des espaces de noms séparés les uns des autres.

Les modules sont en réalité bien plus riches que cela : rien n'empêche de déclarer des modules à l'intérieur d'autres modules (c'est même courant, et nous le feront). Pour cela, dans un fichier, on créée un module avec la ligne `module Titi = struct`, et on termine la déclaration avec `end`. On passera sur les détails (notamment que les modules sont «typés» via une signature). Dans le reste du fichier, le module est alors nommé `Titi`, et s'il se trouve dans le fichier `toto.ml`, depuis un autre fichier, il sera connu sous le nom Toto.Titi. Avec dune, il y aura un autre niveau de module, à savoir les bibliothèques (qui correspondront aux dossiers). Ainsi, si le fichier précédent se trouve dans le dossier `lib`, alors le nom du module précédent sera `Lib.Toto.Titi`

Exercice 8: Exceptions Un autre concept qu'on utilisera, ce sont les exceptions, qui permettront de terminer des calculs avec des types non-prévus dans le prototype de la fonction et de changer le flot du programme pour remonter plusieurs appels de fonction

d'un coup jusqu'au rattrapage de l'exception. Dans notre cas, on s'en servira principalement pour gérer les erreurs (comme la plupart des cas d'utilisation), mais également pour les retours de fonctions dans l'interpréteur.

Elles fonctionnent de manière très similaire à d'autres langages que vous avez vus notamment le Java.

En l'occurrence, en OCaml, il y a un type particulier `exn` qui représente les exceptions. La requête `exception` permet d'en déclarer de nouvelles à la volée dans n'importe quel module : cette déclaration prend la forme de l'ajout d'un constructeur (comme dans les types collections, et à ce titre, il peut contenir des éléments arbitraire).

La lancée d'une exception se fait avec l'expression `raise`. Son rattrapage se fait grâce à un bloc `try expression with` qui rattrapera les exceptions lancées par `expression`. Le `with` est suivi d'un pattern matching sur le type exception qui peut ne pas être exhaustif, et donc chaque branche doit être du même type que `expression`.

Vous pouvez regarder et jouer avec la fin du fichier tuto qui fourni deux exemples d'utilisation d'exception. Bien utilisées, elles peuvent simplifier un code qui sinon serait plus complexe à concevoir, même si on préfère usuellement préférer leur usage à la gestion d'erreur. Plus de détail peut être trouvé ici : `https://ocaml.org/docs/error-handling`.

2 Partie 2 : Une première mise en pratique

Dans cette partie, on va travailler sur un type simple représentant des *termes arithmétiques entiers* qui ressemblent à ce qu'on manipulera quand on s'attaquera aux programmes le TD prochain.

Il vous faut compléter le fichier `term/term.ml`

Le type Term vous est donné, ainsi que deux fonctions de lecture de chaînes de caractères représentant un tel terme. Attention, la lecture des termes en représentation infixe (habituelle) ne fonctionne que sur des termes entièrement parenthésés. Par exemple, "1 + 2 * 5" sera reconnu comme "1 + 2", pour avoir le terme complet au sens habituel, il faudra écrire "1 + (2 * 5)". En effet parser de telles expressions nécessite un peu plus que ce que j'ai fait là (et on verra comment faire cela plus tard).

La documentation de ce que vous avez à implémenter est fournie dans `term/term.mli` et est générée dans un fichier html via '`make doc`'. Vous pouvez compilez uniquement la partie présente via '`make term.out`'.

Exercice 9: Manipulation d'un terme Le but de cet exercice est de fournir des fonctions simples d'affichage et de calcul d'un terme.

Vous devez fournir les fonctions suivantes :

- `infix_string_of_term` qui renvoie une chaîne de caractère représentant le terme en notation infixe (comme vous en avez l'habitude). On ne vous demande pas de tenir compte des priorité, aussi vous placerez des parenthèses autour de chaque sous-terme n'étant pas un entier. Cependant, les opérateur min et max seront notées comme des fonctions (par exemple min(2,3), ce qui correspondra à l'écriture 2 v 3 en entrée du programme (ou v 2 3)).

- `polish_string_of_term` qui fait le même travail, mais en notation polonaise (ou préfixe). Cette notation a l'avantage de ne pas nécessiter de parenthèses. Par exemple, le terme 1 + (2 * 3) s'écrira + 1 * 2 3, et le terme (1 + 2) * 3 s'écrira * + 1

2 3. Les opérateurs min et max suivront les mêmes conventions : max 2 3 sera la notation qui correspond à max(2,3) en notation infixe.

- `eval_term` qui calcule la valeur d'un terme.

Ces trois fonctions seront évidemment récursives. Comme elles sont sans effet de bord, l'ordre d'évaluation est peu important, mais la construction de la valeur de retour a une importance.

Une remarque importante tant qu'à faire : remarquez que la manière d'écrire des appels de fonctions en OCaml n'est rien d'autre de de la notation polonaise (à ceci près qu'en OCaml, du fait du système de typage, il est nécessaire de mettre des parenthèses pour désambiguïser le nombre d'arguments).

Exercice 10: Tables de hachage et améliorations Dans le cadre du compilateurs, on aura régulièrement à utiliser des tables de hachages (ou d'autres structures impératives) pour simplifier l'implémentation et retenir des informations sur tout le programme analysé. Étant donné que ce sont des structures se modifiant par effet de bord, l'ordre d'évaluation prendra évidemment de l'importance.

Cet exercice peut être vu comme optionnel (en gros, finissez le reste avant de vous y attaquer) : dans à peu près tous les cas où nous utiliserons des tables de hachages, nous vous fournirons un module les utilisant à votre place et vous fournissant uniquement les fonctions utiles.

Dans cet exercice, on va utiliser deux tables de hachage pour améliorer la fonction `eval_term` pour ne pas calculer plusieurs fois des termes identiques au cours d'un même calcul. On va coder une fonction nommée `eval_term_optim` qui au lieu de renvoyer la valeur du terme renverra deux tables de hachages indexées par des termes :

- La première, `value_table` qui contiendra pour chaque sous-terme rencontré la valeur de ce sous-terme.
- La seconde, `occurence_table` qui contiendra pour chaque sous-terme le nombre de fois où il a été rencontré dans l'exploration.

Par exemple, pour le terme $(1 + (2 * 3))/((2 * 3) - (1 + (2 * 3)))$, `occurence_table` contiendra les valeurs suivantes pour chaque sous-terme :

- $1 : 1$
- $2 : 1$
- $3 : 1$
- $2 * 3 : 2$
- $1 + (2 * 3) : 2$
- $(2 * 3) - (1 + (2 * 3)) : 1$
- $(1 + (2 * 3))/((2 * 3) - (1 + (2 * 3))) : 1$

Pour implémenter cette fonction, vous aurez besoin d'une fonction auxiliaire récursive à `eval_term_opt`, cette dernière se bornant à initialiser des tables de hachage vide et à appeler sa fonction auxiliaire sur ces tables et le terme. Une fois le retour de la fonction auxiliaire effectué, elle renverra ces deux tables de hachage.

La fonction auxiliaire commencera par déterminer si le terme qu'elle a reçu est déjà présent dans `value_table`. Si oui, elle incrémente la valeur associée au terme dans `occurence_table` et renvoie la valeur associée dans `value_table`. Si non, elle calcule cette valeur (comme dans l'exercice précédent), puis, avant de la renvoyée, la place dans `value_table` et place 1 pour ce terme dans `occurence_table` (puisque c'est la première fois qu'on le vois).

La bibliothèque Hashtbl a une documentation qui peut être trouvée ici : `https://v2.ocaml.org/api/Hashtbl.html`.

Vous aurez uniquement besoin des fonctions suivantes :

- `Hashtbl.create` qui crée une table de hachage vierge. Exemple:

```
let table = Hashtbl.create 10
```

- `Hashtbl.replace` qui remplace une valeur dans une table de hachage (et la crée si elle en est absente). Exemple:

```
Hashtbl.replace table (Int(18))18
```

  remplace la valeur associée à `Int(18)` par l'entier 18. À ne pas confondre avec la fonction add, qui fait autre chose et dont nous n'aurons a priori pas besoin.

- `Hashtbl.find` qui renvoie la valeur associée à une clé dans une table de hachage, et une exception sinon. Exemple:

```
Hashtbl.find table (Int(18))
```

  renverra 18 si elle est exécutée après l'instruction `replace` ci-dessus. Il existe également la variante `Hashtbl.find_opt` qui renverrait `Some 18` dans le cas précédent, mais `None` au lieu d'une exception si la clé est absente.

- `Hashtbl.mem` qui renvoie vrai si la clé donnée est associée à une valeur dans la table. Par exemple

```
Hashtbl.mem table (Int(18))
```

  renverrait vrai après l'instruction replace précédente, mais faux avant.