Consignes

- Pour avoir une correction, rendez votre copie lors de la séance de cours du 22 avril.
- Pour que la préparation joue son rôle, mettez-vous dans des conditions d'examen : aucun document à part une feuille A4 manuscrite, et arrêtez-vous au bout de 3h30.
- Ne vous lancez pas avant d'avoir révisé entièrement votre cours. Le plus important est d'avoir refait par vous-mêmes tous les exercices des transparents.
- Ce rendu n'est pas obligatoire et ne sera pas pris en compte pour la validation de l'UE. Néanmoins, il est dans votre intérêt de le faire en jouant le jeu jusqu'au bout, car cela augmentera fortement vos chances de valider.
- Sauf indication contraire, vous pouvez utiliser toute la bibliothèque standard d'OCaml et toute la bibliothèque Yojson.
- Votre code doit être clair et concis.

À titre indicatif, pour chaque question qui demande du code, il est possible de répondre à toute la question en 7 lignes ou moins, sans écrire de fonction auxiliaire.

1 Arbres binaires

1. (1 point) Définir un type `('n, 'f) tree` permettant de représenter des arbres :
   - binaires (chaque nœud a deux fils),
   - avec des étiquettes de type `'n` aux nœuds,
   - avec des étiquettes de type `'f` aux feuilles.
2. (0.5 points) Écrire deux fonctions
   - `leaf : 'f -> ('n, 'f) tree`
   - `node : 'n -> ('n, 'f) tree -> ('n, 'f) tree -> ('n, 'f) tree`

   permettant de construire respectivement des feuilles et des nœuds.
3. (1 point) Écrire une fonction

   ```
   max_leaf : ('n, int) tree -> int
   ```

   qui calcule la plus grande étiquette de *feuille* d'un arbre binaire.
4. (1.5 points) Écrire une fonction

   ```
   iter_nodes : ('n -> unit) -> ('n, 'f) tree -> unit
   ```

   telle que `iter_nodes g a` applique successivement la fonction `g` aux étiquettes de chacun des nœuds de l'arbre, dans l'ordre *infixe*, c'est-à-dire d'abord le fils gauche, puis le nœud courant, puis le fils droit. Par exemple, le code suivant :

   ```
   let l = node "Left" (leaf ()) (leaf ()) in
   let r = node "Right" (leaf ()) (leaf ()) in
   let p = node "Parent" l r in
   iter_nodes print_endline p
   ```

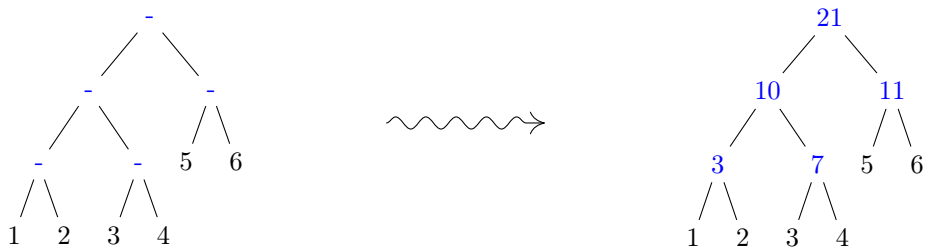
   doit afficher :

   ```
   Left
   Parent
   Right
   ```
5. (1.5 points) Qu'affiche le code suivant ?

   ```
   let a = node (ref 0) (leaf ()) (leaf ()) in
   let b = node (ref 0) a a in
   iter_nodes (fun n -> n := !n+1) b;
   iter_nodes (fun n -> print_int !n; print_newline ()) b
   ```
6. (1.5 points) Écrire une fonction

   ```
   sums : ('n, int) tree -> (int, int) tree
   ```

   qui remplace chaque étiquette de nœud par la somme des étiquettes des feuilles qui sont dessous :



7. (1 point) Écrire une fonction

```
tree_to_json : (int, unit) tree -> Yojson.Basic.t
```

qui convertit un arbre de type `(int, unit) tree` en une valeur Json avec la convention suivante :
— les feuilles sont converties en la valeur Json « `null` » ;
— les nœuds sont convertis en des valeurs Json de la forme « `{ label: <...>, left: <...>, right: <...> }` ».

2 Dictionnaires

1. (1 point) Rappeler la définition d'un *module* et la définition d'une *signature*. Chacune des deux définitions doit tenir en une une phrase de dix mots maximum.
2. (1 point) Définir une signature `DictSig` (à l'aide de la syntaxe `module type <...> = <...>`) contenant :
   — un type abstrait `'a t` ;
   — une valeur `empty` de type `'a t` ;
   — une valeur `add` de type `string -> 'a -> 'a t -> 'a t` ;
   — une valeur `find_opt` de type `string -> 'a t -> 'a option` ;
   — une valeur `find` de type `string -> 'a t -> 'a`.
3. (1 point) Attention : pour cette question (et cette question seulement), vous n'avez pas le droit d'utiliser de conteneurs standard autres que les listes. En revanche, vous pouvez utiliser toutes les fonctions du module `List`.
   Définir un module `Dict` de signature `DictSig` (à l'aide de la syntaxe `module <...> : <...> = <...>`) contenant :
   — un type `'a t` qui représente des dictionnaires associant à des clés de type `string` des valeurs de type `'a`. Les valeurs de type `'a t` doivent être *non modifiables* ;
   — une valeur `empty` de type `'a t` qui représente le dictionnaire vide ;
   — une fonction `add` de type `string -> 'a -> 'a t -> 'a t` qui ajoute une nouvelle association clé-valeur à un dictionnaire existant, et renvoie le dictionnaire ainsi obtenu. (Il n'est pas nécessaire de vérifier si la clé était déjà présente, ni de supprimer l'ancienne valeur s'il y en avait une) ;
   — une fonction `find_opt` de type `string -> 'a t -> 'a option` qui renvoie la valeur associée à une clé dans un dictionnaire (ou renvoie `None` si cette clé n'est associée à aucune valeur). Si la clé a été ajoutée plusieurs fois, il faut renvoyer la valeur *la plus récemment ajoutée*.
   — une fonction `find` de type `string -> 'a t -> 'a` qui se comporte comme `find_opt` mais lève l'exception `Not_found` si la clé n'est associée à aucune valeur.
   Par exemple, le code OCaml suivant :

```
let d = Dict.empty in
let d = Dict.add "A" 1 d in
let d = Dict.add "B" 2 d in
let d = Dict.add "A" 33 d in
(Dict.find_opt "A" d, Dict.find_opt "C" d)
```

   doit s'évaluer en `(Some 33, None)`.
4. (1 point) Le code OCaml suivant est-il correct ? Sinon, dire pourquoi (en une phrase de 10 mots maximum), et en écrire une version corrigée.

```
let d = [] in
let d = Dict.add "A" 1 d in
Dict.find_opt "A" d
```

5. (1 point) Définir (toujours à l'aide de la syntaxe `module <...> : <...> = <...>`) un module `Dict` avec le même comportement que celui de la question 3, mais en utilisant cette fois-ci un conteneur standard autre que les listes. (Il est possible d'écrire le code en une seule ligne de moins de 50 caractères). Outre la concision de la définition, quel est l'avantage de cette implémentation par rapport à celle de la question 3 ?
   (Les notions nécessaires pour répondre à cette question n'ont pas envore été vues en cours.)

3 Expressions arithmétiques avec variables

On cherche à représenter un tout petit fragment du langage OCaml, qui permette juste d'écrire des expressions arithmétiques avec des variables. Pour cela, on définit le type `expr` suivant :

```
type expr =
 | Const of int
 | Add of expr * expr
 | Mul of expr * expr
 | Var of string
 | Let of string * expr * expr
```

Par exemple, l'expression arithmétique « `(1*2)+4` » est représentée par la valeur `Add (Mul (Const 1, Const 2), Const 4)` de type `expr`.

Autre exemple, l'expression arithmétique « `let x = 1 in (let y = x*2 in x+y)` » est représentée par la valeur

```
Let ("x", Const 1, Let ("y", Mul (Var "x", Const 2), Add (Var "x", Var "y")))
```

1. (0.5 points) Quelle est l'*expression arithmétique* représentée par la valeur suivante ?

```
Add (Const 1, Let ("x", Const 2, Mul (Var "x", Var "x")))
```

2. (0.5 points) Quelle est la valeur de type `expr` qui représente l'expression « `let x=1+1 in (2 * (let y = x*x in y+1))` » ?

▶ On va maintenant chercher à *évaluer* des expressions arithmétiques. Pour pouvoir évaluer une expression, il faut se donner un *contexte* qui dise à quelle valeur est associée chaque variable.

Par exemple, dans le contexte « la variable `x` est associée à la valeur `3` » (pour faire court, on écrira : dans le contexte « `x = 3` »), l'expression arithmétique « `x+1` » s'évalue en `4`.

Autre exemple, dans le contexte « `z = 1` », l'expression arithmétique « `let y = 3 in y*y` » s'évalue en `9`.

Enfin, dans le contexte « `x = 3; y = 0` », l'expression arithmétique « `z+2` » ne peut pas s'évaluer, car la variable `z` n'est pas définie.

Attention ! Dans le contexte « `x = 1; y = 2` », l'expression arithmétique « `let y = 0 in y*y` » s'évalue en `0`, et non pas en `4`. On dit que la variable `y` est *masquée* par le `let`. C'est exactement le même phénomène que si vous écriviez en OCaml : « `let x = 1 in let y = 2 in let y = 0 in y*y` ».

3. (0.5 points) Dans le contexte « `x = 3; y = 2` », en quoi s'évalue l'expression arithmétique représentée par `Add (Var "x", Let ("x", Const 2, Mul (Var "x", Var "y")))` ?
4. (0.5 points) Dans le contexte « `x = 3; y = 2` », en quoi s'évalue l'expression arithmétique représentée par `Add (Var "z", Let ("z", Const 0, Var "z"))` ?

▶ On suppose que l'on dispose d'un module `Dict` de signature `DictSig` comme celui décrit dans l'exercice 2. On va représenter les contextes par des expressions OCaml de type `int Dict.t`. Par exemple, l'expression OCaml `Dict.add "x" 1 Dict.empty` représente le contexte « `x = 1` ».

Bien entendu, vous avez le droit de répondre aux questions suivantes même si vous n'avez pas fait l'exercice 2 (vous avez seulement besoin de lire l'énoncé de la question 3 de l'exercice 2).

5. (0.5 points) Écrire une expression OCaml de type `int Dict.t` qui représente le contexte « `x = 3; y = 2` ».
6. (2 points) Écrire une fonction

```
eval : int Dict.t -> expr -> int
```

qui prenne en argument un contexte et une expression arithmétique, et renvoie la valeur de cette expression arithmétique dans ce contexte. Si l'évaluation échoue (parce qu'une variable n'est pas définie), lever l'exception `Not_found`.

Par exemple, le code OCaml suivant doit s'évaluer en `5` :

```
let e = Add (Const 1, Let ("x", Const 2, Mul (Var "x", Var "x"))) in
eval Dict.empty e
```

▶ On dit qu'une variable est *libre* dans une expression arithmétique s'il est nécessaire de donner une valeur à cette variable pour évaluer cette expression. Par exemple, la variable `x` est libre dans « `x + 1` », dans « `x + (let x = 2 in x)` » et dans « `let x = x in 2` », mais pas dans « `y + 1` » ni dans « `let x = y in x` ».

Les règles pour déterminer l'ensemble des variables libres d'une expression arithmétique sont les suivantes :

— l'expression arithmétique `Const k` n'a pas de variable libre ;

— l'unique variable libre de `Var x` est `x` ;

- les variables libres de `Add (e1, e2)` sont les variables libres de `e1` plus celles de `e2` ;
- idem pour `Mul (e1, e2)` ;
- les variables libres de `Let (x, x_def, body)` sont les variables libres de `body`, *moins* la variable `x`, plus les variables libres de `x_def`.

7. (0.5 points) Quelles sont les variables libres de l'expression arithmétique « `let x = y in x + z` », qui est représentée par `Let ("x", Var "y", Add (Var "x", Var "z"))` ?
8. (0.5 points) En utilisant le foncteur approprié de la bibliothèque standard, définir un module `StringSet` permettant de manipuler des ensembles de chaînes de caractères.
9. (1.5 points) En appliquant les règles posées ci-dessus, écrire une fonction `free_vars : expr -> StringSet.t` qui calcule l'ensemble des variables libres d'une expression.
10. (1 point) En utilisant la fonction `free_vars`, écrire une fonction `print_free_vars : expr -> unit` qui *affiche* l'ensemble des variables libres d'une expression arithmétique (afficher simplement un nom de variable par ligne, sans autre formatage).

    Par exemple, le code OCaml suivant :

    ```
    let e = Add (Var "x", Mul (Var "y", Add (Var "x", Var "z"))) in
    print_free_vars e
    ```

    doit afficher les lignes suivantes (dans un ordre quelconque) :

    ```
    x
    y
    z
    ```