Compilation
Master Informatique M1 2011-2012

Examen Partiel
27 octobre 2011

Les questions sont largement indépendantes. L'énoncé est composé de 9 pages. L'examen dure 3 heures.

Le seul document autorisé est une feuille A4 recto-verso manuscripte

Consigne: Lorsqu'il est demandé de décrire une méthode, il n'est pas nécessaire de donner du code dans tous ses détails (vous pouvez utiliser des fonctions auxiliaires dont l'implantation est directe en vous contentant de spécifier leur comportement). Vous devez par contre décrire les données manipulées et les algorithmes utilisés de manière précise, en particulier on indiquera pour chaque fonction introduite *ses arguments, le résultat et le comportement attendus*.

1 Analyse syntaxique (5 points)

On se propose d'écrire un outil qui analyse des expressions régulières pour construire un automate reconnaissant le langage associé. Les symboles terminaux de la grammaire contiennent les parenthèses: `(` et `)`, les caractères ASCII (`'a'`, `'b'`, ...) représentés par le terminal `char`, l'étoile `*`, la somme `+`, le choix `|`, et l'option `?`. La grammaire G des expressions régulières est donnée par les règles suivantes :

$re ::= \texttt{char}$	$re ::= (re)$	$re ::= re*$	$re ::= re+$
$re ::= re?$	$re ::= re \mid re$	$re ::= re\ re$	

1. Cette grammaire est-elle ambigüe ? justifier votre réponse.
   Correction : *Oui, il y a deux arbres de syntaxe abstraite pour l'expression* `charchar char` *suivant si on parenthèse à droite ou à gauche.*

2. Donner les étapes de l'analyse ascendante du mot: `char | ( char *)`
   (indiquer les états successifs de la pile, du mot d'entrée et les actions associées)
   Correction :

	`char|(char*)`	*lecture*
`char`	`|(char*)`	*reduction* $re ::= \texttt{char}$
re	`|(char*)`	*lecture 3 fois*
re `|(char`	`*)`	*reduction* $re ::= \texttt{char}$
re `|(` re	`*)`	*lecture*
re `|(` re `*`	`)`	*reduction* $re ::= re*$
re `|(` re	`)`	*lecture*
re `|(` re `)`		*reduction* $re ::= (re)$
re `|` re		*reduction* $re ::= re \mid re$
re		*succes*

3. Un sous-ensemble de cette grammaire a été formalisé en Ocamlyacc. Les tokens `LP`, `RP`, `CHAR`, `BAR` et `STAR` correspondent respectivement aux symboles terminaux `(`, `)`, `char`, `|` et `*` des questions précédentes.

```
1  re : CHAR
2     | LP re RP
3     | re BAR re
4     | re re
5     | re STAR
```

   Le compilateur signale 8 conflits dont 4 dans l'état suivant:

```
11: shift/reduce conflict (shift 3, reduce 3) on CHAR
11: shift/reduce conflict (shift 7, reduce 3) on STAR
11: shift/reduce conflict (shift 8, reduce 3) on BAR
11: shift/reduce conflict (shift 4, reduce 3) on LP
state 11
        re : re . BAR re  (3)
        re : re BAR re .  (3)
        re : re . re  (4)
        re : re . STAR  (5)
```

Donner pour chacun des conflits de cet état, un exemple d'entrée pour laquelle le conflit se présentera.

Correction :

(a) `char | char char` *peut s'interpréter comme* `(char | char) char` *ou* `char | (char char)`

(b) `char | char *` *peut s'interpréter comme* `(char | char) *` *ou* `char | (char *)`

(c) `char | char | char` *peut s'interpréter comme* `(char | char) | char` *ou* `char | (char | char)`

(d) `char | char ( char )` *peut s'interpréter comme* `(char | char) ( char )` *ou* `char | (char (char))`

4. Dans notre langage, le symbole $*$ a une précédence plus forte que le produit (juxtaposition $r_1 r_2$) qui a lui-même une précédence plus forte que le choix `BAR` ($r_1 \mid r_2$). Le produit et le choix associent à gauche.

   (a) Indiquer à l'aide de parenthèses comment doit être interprétée l'expression régulière:
   `char char char | char char * | char`.

   Correction : *(((*`char char`*)* `char`*)* `|` *(*`char` *(*`char *`*)))* `|` `char`

   (b) Indiquer comment déclarer les précédences dans la grammaire Ocamlyacc pour supprimer les conflits.

   Correction : *Il faut associer une précédence à la règle de produit. La précence la plus forte va au symbole* `STAR` *le symbole* `BAR` *est associatif gauche et doit avoir une précédence plus faible que la règle produit* `re re BAR re` *doit se lire* `(re re) BAR re`*. Concernant la règle produit:*
   - *l'expression* `re1 re2 re3` *doit se lire* `(re1 re2) re3`*, la situation se produit lorsque* `re1 re2` *est sur la pile et on s'apprête à lire un premier caractère de* `re3` *qui est* `LP` *ou* `char`*. Les terminaux* `LP` *ou* `char` *doivent donc avoir une précédence plus faible que celle de la règle produit;*
   - *l'expression* `re1 re2 | re3` *doit se lire* `(re1 re2) | re3`*, ce qui se produit si la précédence de la règle produit est plus forte que celle du terminal* `|`*;*
   - *l'expression* `re1 | re2 re3` *doit se lire* `re1 | (re2 re3)`*, la situation se produit lorsque* `re1 | re2` *est sur la pile et on s'apprête à lire un premier caractère de* `re3` *qui est* `LP` *ou* `char`*. Les terminaux* `LP` *ou* `char` *doivent donc avoir une précédence plus forte que celle de la règle union, c'est-à-dire* `BAR`*;*
   - *l'expression* `re1 re2 *` *doit se lire* `re1 (re2 *)`*, ce qui se produit si la précédence de la règle produit est plus faible que celle du terminal* `*`*.*

   On obtient ce comportement avec la déclaration suivante:

```
%left BAR
%nonassoc LP CHAR
%nonassoc produit
%nonassoc STAR
%%
re :
     CHAR            {}
|    LP re RP        {}
|    re BAR re       {}
|    re  re          %prec produit    {}
|    re STAR         {}
%%
```

2 Construction d'un automate déterministe (9 points)

On introduit le type suivant pour représenter les arbres de syntaxe abstraite des expressions régulières. Ce type est polymorphe par rapport à un type `'a` utilisé pour représenter les caractères (dans un premier temps `'a` égale `char`).

```
type 'a regexp =
  | Eps                                  (* Mot vide      *)
  | Char  of 'a                          (* Caractere c   *)
  | Union of 'a regexp * 'a regexp       (* r_1 | r_2     *)
  | Prod  of 'a regexp * 'a regexp       (* r_1 r_2       *)
  | Star  of 'a regexp                   (* r*            *)
```

1. Indiquer pour chaque règle de la grammaire G de l'exercice précédent l'arbre de syntaxe abstraite associé de type `char regexp`. On utilisera des notations à la Ocamlyacc, par exemple l'action associée à la constructions de l'union sera: $re ::= re\,|\,re$ {`Union`($1,$3)}.
   Correction : *La seule difficulté est d'interpréter les expression `e?` et `e+` à l'aide des constructions proposées. On utilise les equivalences `e?=e`$|\epsilon$ et `e+=ee`$*$.*

$re ::=$ *char*	{*Char*($1)}	$re ::=$ *(re)*	{$1}
$re ::= re*$	{*Star*($1)}	$re ::= re+$	{*Prod*($1, *Star*($1))}
$re ::= re?$	{*Union*(*Eps*, $1)}	$re ::= re\,|\,re$	{*Union*($1, $3)}
$re ::= re\ re$	{*Prod*($1, $3)}		

2. On souhaite identifier chaque caractère qui apparaît dans l'expression régulière par un indice unique. Pour cela on change la représentation des caractères dans l'arbre de syntaxe abstraite. Au lieu d'avoir `Char(a)`, on aura `Char(a,n)` ave n un entier qui n'apparaît qu'une seule fois dans l'expression.
   Ecrire une fonction `mark` qui étant donnée une expression régulière e de type `char regexp`, renvoie une expression de type `(char * int) regexp` dans laquelle on a ajouté à chaque caractère un numéro unique.
   L'expression `Union(Char('a'),Star(Prod(Char('b'),Char('a'))))` correspondant à $a|(ba)*$ est transformée en `Union(Char('a',0),Star(Prod(Char('b',1),Char('a',2))))` que l'on écrira aussi $a_0|(b_1a_2)*$.
   Correction :

```
let newi = let i = ref (-1) in fun () -> incr i; !i
let rec mark = function
    Eps -> Eps | Char a -> Char (a,newi())
  | Union(e1,e2) -> Union(mark e1,mark e2)
  | Prod(e1,e2) -> Prod(mark e1,mark e2)
  | Star(e) -> Star(mark e)
```

3. Ecrire une fonction `null` qui étant donnée une expression régulière e, renvoie un booléen qui est vrai si et seulement si le mot vide appartient au langage reconnu par e.
   Correction :

```
let rec null = function Eps -> true | Char a -> false | Star(e) -> true
  | Union(e1,e2) -> null e1 || null e2 | Prod(e1,e2) -> null e1 && null e2
```

4. On construit un module Ocaml qui permet de représenter des ensembles de caractères indexés.

```
module OT = struct type t = (char * int) let compare = compare end
module CS = Set.Make(OT)
```

   On a en particulier accès aux fonctions suivantes:

```
val empty : t
val mem : elt -> t -> bool
val singleton : elt -> t
val union : t -> t -> t
val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

   (a) Ecrire une fonction Ocaml `first` qui donne pour une expression régulière e, l'ensemble des caractères possibles comme première lettre d'un mot du langage associé à e. Le tableau suivant donne la valeur de `first` dans les différents cas:

e	`first`(e)	e	`first`(e)
`Eps`	$\emptyset$	`Char(a)`	$\{a\}$
$e_1\,|\,e_2$	`first`(e_1) $\cup$ `first`(e_2)	$e*$	`first`(e)
e_1e_2	`first`(e_1) $\cup$ `first`(e_2) si `null`(e_1)	e_1e_2	`first`(e_1) si `null`(e_1) est faux

   Correction :

```
let rec first = function
  Eps -> CS.empty
| Char (a) -> CS.singleton a
| Union(e1,e2) -> CS.union (first e1) (first e2)
| Prod(e1,e2) -> if null e1 then CS.union (first e1) (first e2) else first e1
| Star(e) -> first e
```

(b) Suivant le même modèle, écrire une fonction `last` qui donne pour une expression régulière e, l'ensemble des caractères possibles comme *dernière* lettre d'un mot du langage associé à e.

```
let rec last = function
  Eps -> CS.empty
| Char(a) -> CS.singleton a
| Union(e1,e2) -> CS.union (last e1) (last e2)
| Prod(e1,e2) -> if null e2 then CS.union (last e1) (last e2) else last e2
| Star(e) -> last e
```

(c) Si e est une expression régulière et a un caractère alors on note $\texttt{follows}(a, e)$ l'ensemble des caractères qui peuvent suivre a dans e. Cet ensemble est défini de la manière suivante : $b \in \texttt{follows}(a, e)$ s'il existe une sous-expression e_1e_2 dans e telle que $a \in \texttt{last}(e_1)$ et $b \in \texttt{first}(e_2)$ ou bien une sous-expression de la forme e_0* telle que $a \in \texttt{last}(e_0)$ et $b \in \texttt{first}(e_0)$.
Ecrire la fonction `follows`.

Correction :

```
let rec follows a = function
    Eps | Char(_) -> CS.empty
  |   Union(e1,e2) -> CS.union (follows a e1) (follows a e2)
  |   Prod(e1,e2) -> let s = CS.union (follows a e1) (follows a e2) in
                          if CS.mem a (last e1) then CS.union (first e2) s else s
  |   Star(e)  -> let s = follows a e in
                      if CS.mem a (last e) then CS.union (first e) s else s
```

(d) On introduit un caractère spécial $\sharp$ pour indiquer la fin du mot. Soit l'expression régulière, $(a_0|(b_1a_2)*)\sharp$ calculer les ensembles `first`, `last` de cette expression ainsi que les ensembles `follows` pour tous les caractères de l'expression sauf $\sharp$.

Correction :

$$\texttt{first}(e) = \{a_0, b_1, \sharp\} \quad \texttt{last}(e) = \{\sharp\}$$

$$\texttt{follows}(a_0, e) = \{\sharp\} \quad \texttt{follows}(b_1, e) = \{a_2\} \quad \texttt{follows}(a_2, e) = \{b_1, \sharp\}$$

5. Pour construire un automate déterministe reconnaissant le langage associé à une expression régulière e, on commence par ajouter à la fin de cette expression le caractère $\sharp$, puis on numérote les caractères de manière unique, on obtient une expression régulière e'. L'automate a pour état des ensembles de caractères indexés, les transitions se font sur des caractères simples. L'état initial est $\texttt{first}(e')$ et l'état suivant d'un état s par le caractère a est formé par l'union des ensembles $\texttt{follows}(a_i, e')$ pour tous les caractères indéxés $a_i \in s$. Un état est final s'il contient le caractère $\sharp$.

   (a) Construire l'automate en suivant cette méthode pour l'expression régulière $a|(ba)*$.

   Correction :

   – *L'etat initial est* $s_0 = \texttt{first}(e) = \boxed{a_0, b_1, \sharp}$.

   – *Les transitions possibles à partir de* s_0 *sont sur les caracteres* a *et* b.

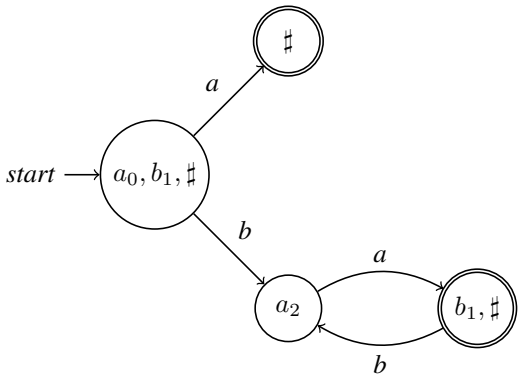
   – *Une transition de* s_0 *sur* a *amène à* $s_1 = \texttt{follows}(a_0, e) = \boxed{\sharp}$ *qui est un etat final dont on ne peut sortir.*

   – *Une transition de* s_0 *sur* b *amène à* $s_2 = \texttt{follows}(b_1, e) = \boxed{a_2}$.

   – *À partir de* s_2 *la seule transition possible est sur le caractère* a *et amène à* $s_3 = \texttt{follows}(a_2, e) = \boxed{b_1, \sharp}$.

   – *À partir de* s_3 *la seule transition possible est sur le caractère* b *et amène à* $\texttt{follows}(b_1, e) = s_2$.

   L'automate obtenu est :



(b) Ecrire une fonction Ocaml `trans` qui étant donnés une expression régulière e, un état s et un caractère a, calcule l'état suivant de l'automate (on utilisera les fonctions définies précédemment).

Correction :

```
let trans e s a =
  CS.fold (fun c ns -> if fst c = a then (CS.union (follows c e) ns) else ns)
          s CS.empty
```

3 Analyse d'exceptions (6 points)

L'objectif de cette partie est de réaliser un analyseur d'exceptions à la manière de Java. L'utilisateur doit déclarer pour chaque fonction un ensemble d'exceptions qui peuvent être levées lors de l'appel de la fonction. Le compilateur doit vérifier que chaque fonction ne lèvera que des exceptions qui ont été déclarées. On considère un langage simplifié dans lequel on peut simplement déclarer des exceptions et des fonctions.

```
exception E
let f (x1,...,xn) = expr
let f (x1,...,xn) raises E1,..,Ek = expr
```

Les valeurs manipulées sont uniquement des entiers. Les exceptions ont une valeur entière. Les fonctions ont un nombre quelconque d'arguments, elle peuvent être définies de manière mutuellement récursives. Les expressions peuvent être de l'une des formes suivantes:

- des constantes entières;
- des variables;
- une opération arithmétique binaire $e_1 \,\mathtt{op}\, e_2$;
- l'appel de fonction $f(e_1, \ldots, e_n)$;
- une conditionnelle **if** e **then** e_1 **else** e_2 (l'entier 0 est interprété comme faux);
- la levée d'une exception **raise** $(\mathtt{E}\, e)$ avec $\mathtt{E}$ un nom d'exception et e la valeur associée;
- des exceptions rattrapées **try** e **with** $\mathtt{E}_1\, x_1 \to e_1 | \ldots | \mathtt{E}_k\, x_k \to e_k$.

Un exemple de programme dans ce langage est:

```
exception Break
exception Continue
let next (i,s) raises Break,Continue =
  if i % 2 = 1 then raise (Continue s)
  else if i = 10 then raise (Break s)
  else s + i
let loop (i,s) =
    try loop(i+1,next(i,s))
    with Continue s' -> loop (i+1,s') | Break s' -> s'
let main () = loop(0,0)
```

Règles de portées Les règles de portées pour les variables sont les suivantes:

- Dans une déclaration de fonction **let** $f(x_1, \ldots, x_n) = e$, les variables $x_1, \ldots, x_n$ ne sont visibles que dans l'expression e;
- dans une condition exceptionnelle **try** e **with** $\mathtt{E}_1\, x_1 \to e_1 | \ldots | \mathtt{E}_k\, x_k \to e_k$. la variable x_i qui représente la valeur associée à l'expression $\mathtt{E}_i$ n'est visible que dans l'expression e_i.

Les exceptions et les fonctions déclarées sont visibles dans l'ensemble du programme.

1. Dans le programme donné en exemple, numéroter de manière unique chaque déclaration de variable (en paramètre ou condition exceptionnelle) et indiquer pour chaque utilisation de variable à quelle déclaration elle correspond.

   Correction :

```
let next (i1,s1) raises Break,Continue =
  if i1 % 2 = 1 then raise (Continue s)
  else if i1 = 10 then raise (Break s1)
       else s1 + i1
let loop (i2,s2) =
  try loop(i2+1,next(i2,s2))
  with Continue s'1 -> loop (i2+1,s'1)
     | Break s'2 -> s'2
let main () = loop(0,0)
```

2. Compléter les types suivants pour représenter les arbres de syntaxe abstraite des programmes de ce langage.

```
type op = Plus | Minus | Mult | Div | Mod | Eq | Le | Lt
type expr = Cte of int | Var of string | Binop of expr * op * expr
          | If of ... | Raise of ... | Try of ... | Call of ...
type decl = Exc of ... | Fun of ...
type prog = decl list
```

   Correction :

```
type expr = Cte of int | Var of string | Binop of expr * op * expr
       | If of expr * expr * expr | Raise of string * expr
       | Try of expr * (string * string * expr) list
       | Call of string * expr list
type decl = Exc of string | Fun of string * string list * string list * expr
```

3. On construit un module Ocaml pour représenter les environnements :

```
module OT = struct type t = string let compare = compare end
module Env = Map.Make(OT)
```

   Le module `Env` comporte en particulier les fonctions suivantes:

```
val empty : 'a t
val add : key -> 'a -> 'a t -> 'a t
val find : key -> 'a t -> 'a
val mem : key -> 'a t -> bool
val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

   Ecrire une fonction `check_prog` qui vérifie qu'un programme respecte les règles de portée du langage et que chaque fonction est appelée avec un nombre d'arguments correspondant à sa déclaration.

   Correction : *L'environnement contindra pour chaque identificateur (variable, fonction et exception), sa nature et dans le cas des fonctions, le nombre d'arguments attendus. Les fonctions de vérification de portee leveront l'exception* `ScopeError` *en cas d'erreur de portée.*

```
type info = IsVar | IsExn| IsFun of int
exception ScopeError
```

   On introduit d'abord une fonction pour vérifier la bonne formation d'une expression.

```
(*  check_expr : info Env.t -> expr -> unit *)
let rec check_expr env = function
    Cte (_) -> ()
  | Var (x) -> if not (Env.mem x env) || Env.find x env <> IsVar then
                 raise ScopeError
  | Binop(e1,_,e2) -> check_expr env e1; check_expr env e2
```

```
| If(e1,e2,e3) -> check_expr env e1; check_expr env e2; check_expr env e3
| Raise(s,e) -> if not (Env.mem s env) || Env.find s env <> IsExn
                then raise ScopeError
                else check_expr env e
| Try(e,le) -> check_expr env e;
             List.iter
               (fun (si,xi,ei) ->
                if not (Env.mem si env) || Env.find si env <> IsExn
                then raise ScopeError
                else check_expr (Env.add xi IsVar env) ei) le
| Call (f,le) -> if not (Env.mem f env) then
                 match Env.find f env with
                    IsFun(n) -> if List.length le = n then List.iter (check_expr env) le
                      else raise ScopeError
                  | _ -> raise ScopeError
```

Les exceptions et fonctions étant visibles dans l'ensemble du programme, on construit pour chaque programme, un environnement initial qui contient toutes les exceptions et les fonctions.

```
let env_init p
    = List.fold_left (fun d env ->
                        match d with
                           Fun(f,lvars,_,_) -> Env.add f (IsFun(List.length lvars)) env
                         | Exc (s) -> Env.add s IsExn env)
     p Env.empty
```

Pour vérifier un programme il suffit de vérifier le corps de chaque déclaration de fonction.

```
let check_prog p =
  let env = env_init p in
  let rec check_decls  = function
    [] -> ()
  | Exc (_)::d -> check_decls d
  | Fun(_,lvars,_,e)::d ->
      let env' = List.fold_right (fun x env -> Env.add x IsVar env) lvars env in
      check_expr env' e; check_decls d
  in check_decls p
```

4. Chaque expression peut en s'exécutant lever certaines exceptions. le compilateur calcule un sur-ensemble des exceptions possiblement levées de la manière suivante:
   - Les variables et constantes ne lèvent pas d'exceptions;
   - L'expression **raise** $(\mathtt{E}\ e)$ lève l'exception `E` plus toutes les exceptions possiblement levées par e;
   - Les expressions **if** e_1 **then** e_2 **else** e_3 et e_1 `op` e_2 lèvent les exceptions possiblement levées par e_1, e_2 et e_3;
   - L'expression $f(e_1, \ldots, e_n)$ lève les exceptions possiblement levées par les arguments e_i plus les exceptions levées par l'appel de fonction f, telles qu'indiquées dans la déclaration de f (après le mot clé **raises**). Les fonctions déclarées sans le mot-clé **raises** sont supposées ne pas lever d'exception.
   - L'expression **try** e **with** $\mathtt{E}_1\, x_1 \to e_1 | \ldots | \mathtt{E}_k\, x_k \to e_k$ lève les mêmes exceptions que e moins les exceptions $\mathtt{E}_i$ plus les exceptions levées par les expressions e_i.

   (a) Soit le programme:

   ```
   exception Div0
   let div0 (n,m) raises Div0 = if m=0 then raise Div0(0) else n/m
   let main(a,b,c) raises ... = try div0(a,b) with Div0 x -> div0(a,c)
   ```

   Compléter les exceptions possiblement levées par l'appel de la fonction `main`.

   Correction :

   ```
   let main(a,b,c) raises Div0 = try div0(a,b) with Div0 x -> div0(a,c)
   ```

(b) Ecrire une fonction `raises_expr` qui étant donné un environnement (dans lequel les fonctions ont été déclarées avec les exceptions possiblement levées par leur appel) et une expression e, calcule l'ensemble des exceptions possiblement levées par l'exécution de e.
On pourra utiliser le module suivant pour manipuler des ensembles d'exceptions

```
module SS = Set.Make(OT)
```

qui définit les fonctions suivantes:

```
val empty : t                    val mem : elt -> t -> bool
val add : elt -> t -> t          val remove : elt -> t -> t
val union : t -> t -> t          val diff : t -> t -> t
val inter : t -> t -> t          val subset : t -> t -> bool
val singleton : elt -> t         val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

Correction :

```
(* raises_expr : SS.t Env.t -> expr -> SS.t *)
let rec raises_expr env = function
    Cte (_) |  Var (_) -> SS.empty
  | Binop(e1,_,e2) -> SS.union (raises_expr env e1) (raises_expr env e2)
  | If(e1,e2,e3) -> SS.union (raises_expr env e1)
                      (SS.union (raises_expr env e2)(raises_expr env e3))
  | Raise(s,e) -> SS.add s (raises_expr env e)
  | Call (f,le) -> let se = Env.find f env in
                   List.fold_right (fun e se -> SS.union  (raises_expr env e) se)
                     le se
  | Try(e,le) -> let se = raises_expr env e in
                 let (se,ne) =
                   List.fold_right
                     (fun (si,_,ei) (se,ne) ->
                       (SS.remove si se,SS.union (raises_expr env ei) ne))
                     le (se,SS.empty)
                 in SS.union se ne
```

(c) Ecrire une fonction `check_exn` qui étant donné un programme vérifie que les déclarations des exceptions levées par les fonctions contiennent bien les exceptions qui peuvent être levées lors de l'exécution du corps de la fonction.
Correction : *On utilise une fonction pour transformer une liste d'exceptions en ensemble d'exceptions. Le programme lèvera l'exception* `ExnError` *lorsque les conditions de levée d'exception ne seront pas satisfaites.*

```
let list_to_set l = List.fold_right SS.add l SS.empty
exception ExnError
```

On collecte initialement l'environnement avec pour chaque fonction, l'ensemble des exceptions qu'elle peut lever.

```
let exn_init p
    = List.fold_right (fun d env ->
                         match d with
                             Fun(f,_,lex,_) -> Env.add f (list_to_set lex) env
                           | Exc (s) -> env)
      p Env.empty
```

La fonction principale, vérifie chaque corps de fonction et lève l'exception `ExnError` *en cas d'erreur.*

```
(* check_exn : prog -> unit *)
let check_exn p =
  let env = exn_init p in
  let rec check_decls  = function
    [] -> ()
  | Exc (_)::d -> check_decls d
  | Fun(_,_,lexn,e)::d ->
```

```
      let fedecl = list_to_set lexn and fe = raises_expr env e in
      if SS.subset fe fedecl then check_decls d else raise ExnError
  in check_decls p
```

On rappelle quelques fonctions utiles dans le module `List`

```
val length : 'a list -> int
val [] :  'a list
val (@) : 'a list -> 'a list -> 'a list
val mem : 'a -> 'a list -> bool
val map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```