Why

Why est un outil développé par Jean-Christophe Filliâtre qui permet certifier des programmes impératifs à l'aide d'annotations logiques. Le langage de programmation impératif est très inspiré du «noyau impératif» de *objective caml*. Voici la version simplifiée de la grammaire de ce langage que l'on utilisera pour ce TP.

```
<const> := 0 | 1 | ... | true | false
<op>    := + | - | * | / | % | = | <> | < | <= | > | >= | || | &&
<expr>  := <ident> | !<ident> | <constant> | <expr> <op> <expr>

<prog>  := let <x> = ref <expr> in <prog>
         | <x> := <expr>
         | while <expr> do <annot><prog>
         | <prog>;<prog>
         | if <expr> then <prog> else <prog>
         | if <expr> then <prog>
         | <prog> {<form>}

<annot> :=   | {invariant <form>} | {variant <form>}
             | {invariant <form> variant <form>}
```

La grosse différence avec caml c'est la possibilité d'ajouter des annotations `{<form>}` entre les séquences d'expressions. La syntaxe des formules est donnée par la grammaire suivante.

```
     <term> := <const> | <term> <arith_op> <term> | <ident> | <ident>@
 <arith_op> := + | - | * | / | %
     <rela> := = | <> | < | <= | > | >=
     <form> := true | false | <ident>(<term>,...,<term>) | <term> <rela> <term>
             | not <form> | <form> and <form> | <form> or <form> | <form> -> <form>
             | forall <ident>:<type>.<form> | exists <ident>:<type>.<form>
     <base_type> := int | bool
```

Il faut remarquer que les `<term>` sont différents des `<expr>` essentiellement car dans le monde logique, il n'y a pas d'effet de bord.

Enfin, on déclare un nouveau programme avec la syntaxe suivante :

```
let <ident> (<ident> : <type>) ... (<ident> : <type>) = { <form> } <prog>
```

ou plus simplement (si on ne souhaite pas mettre de précondition) :

```
let <ident> (<ident> : <type>) ... (<ident> : <type>) =  <prog>
```

où `<type>` est défini par

```
<type> := <base_type> | <type> -> <type> | <type> ref
```

Voici un premier exemple :

```
let swap (a : int ref) (b : int ref) =
   let t = ref !a in
   a := !b;
   b := !t
 {a = b@ and b = a@}
```

C'est un programme qui échange le contenu de deux références. On comprend en lisant la post-condition que `a@` signifie «la valeur contenu dans `a`» avant l'éxécution.

Question 1. *Recopiez ce programme dans un fichier «*`swap.why`*» et compilez à l'aide de la commande :*

```
why --coq swap.why
```

Ça génère une fichier `swap_why.v` *contenant les obligations de preuve. Ouvrez-le avec coq-ide, essayez de les comprendre et de les prouver.*

Soit le squelette de programme ci-dessous.

```
let sort (a:int ref) (b:int ref) (c:int ref) =
  ....
  { a <= b <= c }
```

Question 2. *Complétez-le de façon non-triviale et prouvez les obligations en coq.*

Bien-sûr le programme

```
let sort (a:int ref) (b:int ref) (c:int ref) =
    b := a;
    c := b
  { a <= b <= c }
```

satisfera bien sa post-condition. Pour spécifier plus précisément sont comportement, why nous permet d'axiomatiser des prédicats logiques. Ainsi on peut définit un nouveau symbole de prédicat :

```
logic perm : int, int, int, int, int, int -> prop
```

et l'axiomatiser :

```
axiom perm_base :
   forall a:int.forall b:int.forall c:int.perm(a,b,c,a,b,c)
axiom perm_transp1 : ...
axiom perm_transp2 : ...
axiom perm_transp3 : ...
...
```

Question 3. *Recopiez et complétez l'axiomatisation pour que* `perm(a,b,c,d,e,f)` *signifie que* $[a, b, c]$ *est une permutation de* $[d, e, f]$*. Ensuite, remplacez la post-condition et prouvez les nouvelles obligations.*

Question 4. *Prouvez la correction (partielle d'abord, puis totale) des deux programmes suivants.*

```
let sum (n : int) =
{ 0 <= n}
  let r = ref 0 in
  let k = ref 0 in
  while !k <= n do
    r := !r+!k;
    k := !k+1
  done;
  !r
{ 2*result = n * (n + 1) }
```

```
let euclid (n : int) (m : int) =
{m >= 0 and n >= 0}
  let a = ref m in
  let b = ref n in
  while !a <> !b do
    if !a < !b then
      b := !b - !a
    else
      a := !a - !b
  done; !a
{ is_gcd(n,m,result) }
```

Un petit problème pour la correction totale du deuxième ?