2 Langage avec instructions `break` et `continue` (10 points)

On considère un mini-langage impératif BC comportant des conditionnelles et des boucles. Ce langage comporte de plus deux instructions `break` et `continue` qui permettent de modifier l'exécution du corps d'une boucle. Lorsque dans le corps d'une boucle, on rencontre l'instruction `break` alors l'exécution se poursuit à la fin de la boucle, lorsque l'on rencontre l'instruction `continue` alors l'exécution reprend au test de la boucle. Lorsque plusieurs boucles sont imbriquées, les instructions `break` et `continue` modifient l'exécution de la première boucle englobante.

On cherche à compiler le langage BC en faisant tout d'abord une phase d'analyse statique pour vérifier la bonne formation des expressions puis la génération de code mips.

Les expressions du langage BC sont soit des constantes entières, soit des variables (globales ou paramètres de fonction) soit une opération binaire (arithmétique ou test d'égalité) appliquée à deux expressions, soit l'appel d'une fonction. On convient que les booléens `true` et `false` sont représentés respectivement par les entiers 1 et 0.

Une instruction de BC peut être soit une affectation $x = e$, soit une conditionnelle **`if`** (e) s_1 **`else`** s_2 ou bien **`if`** (e) s, soit un bloc formé d'une liste d'instructions $\{s_1; \ldots; s_n\}$ soit une boucle **`while`** (e) s, soit une instruction d'échappement **break** ou **continue**, soit une instruction de retour de fonction **`return`** e.

Les types CAML suivants permettent de représenter les arbres de syntaxe abstraite de ce langage (les variables globales sont représentées par leur étiquette permettant d'accéder à la zone de données, les variables locales sont identifiées par un entier représentant le décalage par rapport au registre $fp dans le tableau d'activation):

```
type op = Plus | Mult | Sub | Div | Mod | Eq
type expr = Const of int | Glob of string | Loc of int
          | Op of op * expr * expr | Call of string * expr list
type stat =   Aff of string * expr
            | If of expr * stat * stat
            | Block of stat list
            | While of expr * stat
            | Break | Continue
            | Return of expr
```

1. Un sous-ensemble de la grammaire du langage contient les règles suivantes:

```
expr:
| CST                                { }
| IDENT                              { }

instr:
| IF LP expr RP instr                { }
| IF LP expr RP instr ELSE instr     { }
| WHILE LP expr RP instr             { }
| RETURN expr                        { }
| BREAK                              { }
| CONTINUE                           { }
;
```

Ocamlyacc indique un conflit shift/reduce dans l'état suivant

```
20: shift/reduce conflict (shift 22, reduce 3) on ELSE
state 20
instr : IF LP expr RP instr .  (3)
```

```
instr : IF LP expr RP instr . ELSE instr  (4)

ELSE  shift 22
$end  reduce 3
```

(a) Donner un exemple d'entrée sur laquelle le conflit se produit.

(b) Indiquer comment le conflit est résolu (on s'appuiera sur les conventions des langages comme C ou Java) et à quelle stratégie (lecture ou réduction) cela correspond.

(c) Donner le choix de précédence pour supprimer le conflit dans la grammaire.

Correction : *Un exemple est*

```
if (0) if (1) return 0; else return 1;
```

Le `else` *peut être associé dans la grammaire au prmier ou au second* `if`*. L'usage est d'associer au* `if` *le plus proche. Ce qui correspond dans la grammaire à une lecture du terminal* `ELSE` *plutôt que la réduction de la règle* `instr : IF LP expr RP instr`*. Il faut donc que la précédence du terminal* `else` *soit plus grande que celle de la règle qui par défaut est celle du terminal* `RP`*. On ajoutera à la grammaire les précédences:*

```
%nonassoc RP
%nonassoc ELSE
```

2. Écrire une fonction `check_loop` qui vérifie que dans chaque instruction, les commandes **break** et **continue** n'apparaissent que dans le corps d'une boucle.
   Correction :

```
let rec check_loop = function
    Break | Continue -> false
  | While(_,_) -> true
  | Block l -> List.for_all check_loop l
  | If(_,s1,s2) -> check_loop s1 && check_loop s2
  | _ -> true
```

3. Une instruction **return** correspond à la fin d'un flot de contrôle du programme. Écrire une fonction `check_return` qui vérifie que dans une instruction du langage BC, tout chemin d'exécution se termine bien par une instruction **return**. On vérifiera de plus qu'il n'y a pas d'instruction qui suit immédiatement une instruction dont le flôt de controle se termine par des **return** et qui donc n'est jamais atteinte par le flot de contrôle.
   Exemples.

```
{ if (x == 0) return 0;  x=x+1;  return x; }
```

Le programme précédent est correct: le flôt après le **if** va soit dans la branche **return** 0 qui se termine par un return, soit dans la branche `x = x+1; return x;` qui se termine bien aussi par un **return**.

```
{ while (x <= 0) { x=x+1; return 0; } }
```

Le programme précédent est incorrect: le flot de contrôle de l'instruction **while** peut ne pas entrer dans la boucle auquel cas il n'y aura pas d'instruction **return**.

```
{ return 0; x=x+1 }
```

Le programme précédent est incorrect: le flot de contrôle n'atteint jamais l'instruction `x=x+1`.

On pourra découper en deux fonctions, la première qui vérifie que chaque chemin d'exécution se termine par un **return** et la seconde qui vérifie qu'il n'y a pas d'instruction non-atteignable à la suite d'une instruction dont l'exécution se termine toujours par un **return**.

Correction :

```
(* takes as argument a statement s and returns a boolean
   the boolean is true if s has a possible execuuttion path
   not ending with return *)
let rec no_return = function
  | Return _ -> false
  | Block l -> List.for_all no_return l
  | If(_,s1,s2) -> no_return s1 || no_return s2
  | _ -> true
(* takes as argument a statement s and returns a boolean
   the boolean is true if all possible execution paths of s ends
   with return and no code appears after a return *)
let rec check_return = function
  | Return _ -> true
  | Block l -> let lr = List.rev l in
      (match lr with [] -> false
        | a::m -> List.for_all no_return m && check_return a)
  | If(_,s1,s2) -> check_return s1 && check_return s2
  | While(_,s) -> false
  | _ -> false
```

4. On suppose donnée une fonction `compile_expr` qui prend en argument un registre r et une expression e et produit le code mips permettant de calculer la valeur de e dans le registre r.
Écrire une fonction de compilation des instructions de BC vers du code mips dans le cas où il n'y a pas d'instruction **break** ou **continue** dans les boucles. La valeur de l'expression renvoyée par l'instruction **return** sera stockée dans le registre $\$v0$. On pourra utiliser l'interface Ocaml pour manipuler des instructions mips donnée en fin de document ou bien utiliser de manière informelle la syntaxe des instructions mips dans le code Ocaml.

Correction :

```
let rec compile_stat = function
    Aff(s,e) -> compile_expr A0 e ++ mips [Sw(A0,Alab s)]
  | If(e,s1,s2) ->
     let lelse = new_label "else" and eif = new_label "eif" in
     compile_expr A0 e ++ mips [Beqz(A0,lelse)] ++
     compile_stat s1 ++ mips [B eif; Label lelse] ++
     compile_stat s2 ++ mips [Label eif]
  | Block l -> List.fold_left (fun c i -> c ++ compile_stat i) nop l
  | While(e,s) ->
      let bloop = new_label "bloop" and eloop = new_label "eloop" in
      mips [Label bloop] ++ compile_expr A0 e ++
      mips [Beqz(A0,eloop)] ++ compile_stat s ++
      mips [B bloop;Label eloop]
  | Return e -> compile_expr V0 e
  | Break | Continue -> assert false
```

5. Soit le programme de BC :

```
sum = 0;
x = 0;
while (true) {
    x = x+1;
    if (x % 2 == 1) continue;
    if (x == 10) break;
    sum = sum + x;
    }
return(sum);
```

L'opération $n\%p$ calcule le reste de la division entière de n par p (opération modulo) et s'implante par l'opération mips `rem`.

(a) Quel est le résultat de ce programme ?

(b) Dessiner le graphe de flot de contrôle de ce programme.

(c) Donner des instructions mips correspondant au code compilé de cette fonction en supposant que sum, et x sont des variables globales.

Correction :

(a) résultat 20

6. Écrire une fonction de compilation des instructions dans le cas général où les corps de boucle peuvent mentionner les instructions **break** et **continue**.

Correction :

```
(* ltest is the label corresponding to the test of the current loop
   lend is the label corresponding to the end of the current loop
   they are initialized by dummy labels *)
let rec compile_break ltest lend = function
    Aff(s,e) -> compile_expr A0 e ++ mips [Sw(A0,Alab s)]
  | If(e,s1,s2) ->
     let lelse = new_label "else" and eif = new_label "eif" in
     compile_expr A0 e ++ mips [Beqz(A0,lelse)] ++
     compile_break ltest lend s1 ++ mips [B eif; Label lelse] ++
     compile_break ltest lend s2 ++ mips [Label eif]
  | Block l -> List.fold_left (fun c i -> c ++ compile_break ltest lend i)
  | While(e,s) ->
      let bloop = new_label "bloop" and eloop = new_label "eloop" in
      mips [Label bloop] ++ compile_expr A0 e ++
      mips [Beqz(A0,eloop)] ++ compile_break bloop eloop s ++
      mips [B bloop;Label eloop]
  | Return e -> compile_expr V0 e
  | Break -> mips [B lend]
  | Continue -> mips [B ltest]
```