```
(* Romain SADOK & Antoine MULLIER : TP67 ACF Groupe 2.1 *)
theory tp67
imports Main (* "~~/src/HOL/Library/Code_Target_Int" "~~/src/HOL/Library/Code_Char" *)
begin

(* Types des expressions, conditions et programmes (statement) *)
datatype expression= Constant int | Variable string | Sum expression expression | Sub expression expression

datatype condition= Eq expression expression

datatype statement= Seq statement statement |
                    Aff string expression |
                    Read string |
                    Print expression |
                    Exec expression |
                    If condition statement statement |
                    Skip
(* Un exemple d'expression *)

(* expr1= (x-10) *)
definition "expr1= (Sub (Variable ''x'') (Constant 10))"


(* Des exemples de programmes *)

(* p1= exec(0) *)
definition "p1= Exec (Constant 0)"

(* p2= {
        print(10)
        exec(0+0)
       }
*)

definition "p2= (Seq (Print (Constant 10)) (Exec (Sum (Constant 0) (Constant 0))))"

(* p3= {
         x:=0
         exec(x)
       }
*)

definition "p3= (Seq (Aff ''x'' (Constant 0)) (Exec (Variable ''x'')))"

(* p4= {
         read(x)
         print(x+1)
       }
*)
definition "p4= (Seq (Read ''x'') (Print (Sum (Variable ''x'') (Constant 1))))"


(* Le type des evenements soit X: execute, soit P: print *)
datatype event= X int | P int

(* les flux de sortie, d'entree et les tables de symboles *)

type_synonym outchan= "event list"
definition "el1= [X 1, P 10, X 0, P 20]"                    (* Un exemple de flux de sortie *)

type_synonym inchan= "int list"
definition "il1= [1,-2,10]"                                 (* Un exemple de flux d'entree [1,-2,10]              *)

type_synonym symTable= "(string * int) list"
definition "(st1::symTable)= [(''x'',10),(''y'',12)]"       (* Un exemple de table de symbole *)


(* La fonction (partielle) de recherche dans une liste de couple, par exemple une table de symbole *)
datatype 'a option= None | Some 'a

fun assoc:: "'a \<Rightarrow> ('a * 'b) list \<Rightarrow> 'b option"
where
"assoc _ [] = None" |
"assoc x1 ((x,y)#xs)= (if x=x1 then Some(y) else (assoc x1 xs))"

```



```
(* Exemples de recherche dans une table de symboles *)

value "assoc ''x'' st1"      (* quand la variable est dans la table st1 *)
value "assoc ''z'' st1"      (* quand la variable n'est pas dans la table st1 *)


(* Evaluation des expressions par rapport a une table de symboles *)
fun evalE:: "expression \<Rightarrow> symTable \<Rightarrow> int"
where
"evalE (Constant s) e = s" |
"evalE (Variable s) e= (case (assoc s e) of None \<Rightarrow> -1 | Some(y) \<Rightarrow> y)" |
"evalE (Sum e1 e2) e= ((evalE e1 e) + (evalE e2 e))" |
"evalE (Sub e1 e2) e= ((evalE e1 e) - (evalE e2 e))"

(* Exemple d'évaluation d'expression *)

value "evalE expr1 st1"

(* Evaluation des conditions par rapport a une table de symboles *)
fun evalC:: "condition \<Rightarrow> symTable \<Rightarrow> bool"
where
"evalC (Eq e1 e2) t= ((evalE e1 t) = (evalE e2 t))"

(* Evaluation d'un programme par rapport a une table des symboles, a un flux d'entree et un flux de sortie.
   Rend un triplet: nouvelle table des symboles, nouveaux flux d'entree et sortie *)
fun evalS:: "statement \<Rightarrow> (symTable * inchan * outchan) \<Rightarrow> (symTable * inchan * outchan)"
where
"evalS Skip x=x" |
"evalS (Aff s e)  (t,inch,outch)=  (((s,(evalE e t))#t),inch,outch)" |
"evalS (If c s1 s2)  (t,inch,outch)=  (if (evalC c t) then (evalS s1 (t,inch,outch)) else (evalS s2 (t,inch,outch)))" |
"evalS (Seq s1 s2) (t,inch,outch)=
    (let (t2,inch2,outch2)= (evalS s1 (t,inch,outch)) in
        evalS s2 (t2,inch2,outch2))" |
"evalS (Read _) (t,[],outch)= (t,[],outch)" |
"evalS (Read s) (t,(x#xs),outch)= (((s,x)#t),xs,outch)" |
"evalS (Print e) (t,inch,outch)= (t,inch,((P (evalE e t))#outch))" |
"evalS (Exec e) (t,inch,outch)=
  (let res= evalE e t in
   (t,inch,((X res)#outch)))"



(* Exemples d'évaluation de programmes *)
(* Les programmes p1, p2, p3, p4 ont été définis plus haut *)
(* p1= exec(0) *)

value "evalS p1 ([],[],[])"

(* ------------------------------------ *)
(* p2= {
        print(10)
        exec(0+0)
       }
*)

value "evalS p2 ([],[],[])"

(* ------------------------------------ *)
(* p3= {
         x:=0
         exec(x)
       }
*)

value "evalS p3 ([],[],[])"

(* ------------------------------------ *)
(* p4= {
         read(x)
         print(x+1)
       }
*)

value "evalS p4 ([],[10],[])"


```

```
definition "bad1= (Exec (Constant 0))"
definition "bad2= (Exec (Sub (Constant 2) (Constant 2)))"
definition "bad3= (Seq (Aff ''x'' (Constant 1)) (Seq (Print (Variable ''x'')) (Exec (Sub (Variable ''x'') (Constant 1)))))"
definition "bad4= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) Skip (Aff ''y'' (Constant 1))) (Exec (Sum (Variable ''y'') (Constant 1)))))"
definition "bad5= (Seq (Read ''x'') (Seq (Aff ''y'' (Sum (Variable ''x'') (Constant 2))) (Seq (If (Eq (Variable ''x'') (Sub (Constant 0) (Constant 1))) (Seq (Aff ''x'' (Sum (Variable ''x'') (Constant 2))) (Aff ''y'' (Sub (Variable ''y'') (Variable ''x'')))) (Seq (Aff ''x'' (Sub (Variable ''x'') (Constant 2))) (Aff ''y'' (Sub (Variable ''y'') (Variable ''x''))))) (Exec (Variable ''y'')))))"
definition "bad6= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''z'' (Constant 1)) (Aff ''z'' (Constant 0))) (Exec (Variable ''z'')))))"
definition "bad7= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''z'' (Constant 0)) (Aff ''z'' (Constant 1))) (Exec (Variable ''z'')))))"
definition "bad8= (Seq (Read ''x'') (Seq (Read ''y'') (If (Eq (Variable ''x'') (Variable ''y'')) (Exec (Constant 1)) (Exec (Constant 0))))))"
definition "ok0= (Seq (Aff ''x'' (Constant 1)) (Seq (Read ''y'') (Seq (If (Eq (Variable ''y'') (Constant 0)) (Seq (Print (Sum (Variable ''y'') (Variable ''x'')))
(Print (Variable ''x''))
) (Print (Variable ''y''))
) (Seq (Aff ''x'' (Constant 1)) (Seq (Print (Variable ''x''))
 (Seq (Aff ''x'' (Constant 2)) (Seq (Print (Variable ''x'')))
 (Seq (Aff ''x'' (Constant 3)) (Seq (Print (Variable ''x'')))
 (Seq (Read ''y'') (Seq (If (Eq (Variable ''y'') (Constant 0)) (Aff ''z'' (Sum (Variable ''x'') (Variable ''x'')))) (Aff ''z'' (Sub (Variable ''x'') (Variable ''y'')))) (Print (Variable ''z''))
))))))))))"
definition "ok1= (Seq (Aff ''x'' (Constant 1)) (Seq (Print (Sum (Variable ''x'') (Variable ''x'')))
 (Seq (Exec (Constant 10)) (Seq (Read ''y'') (If (Eq (Variable ''y'') (Constant 0)) (Exec (Constant 1)) (Exec (Constant 2)))))))"
definition "ok2= (Exec (Variable ''y''))"
definition "ok3= (Seq (Read ''x'') (Exec (Sum (Variable ''y'') (Constant 2))))"
definition "ok4= (Seq (Aff ''x'' (Constant 0)) (Seq (Aff ''x'' (Sum (Variable ''x'') (Constant 20))) (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''z'' (Constant 0)) (Aff ''z'' (Constant 4))) (Seq (Exec (Variable ''z'')) (Exec (Variable ''x'')))))))"
definition "ok5= (Seq (Read ''x'') (Seq (Aff ''x'' (Constant 4)) (Exec (Variable ''x'')))))"
definition "ok6= (Seq (If (Eq (Constant 1) (Constant 2)) (Aff ''x'' (Constant 0)) (Aff ''x'' (Constant 1))) (Exec (Variable ''x''))))"
definition "ok7= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''x'' (Constant 1)) (If (Eq (Variable ''x'') (Constant 4)) (Aff ''x'' (Constant 1)) (Aff ''x'' (Constant 1)))) (Exec (Variable ''x'')))))"
definition "ok8= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''x'' (Constant 1)) (Aff ''x'' (Constant 2))) (Exec (Sub (Variable ''x'') (Constant 3)))))"
definition "ok9= (Seq (Read ''x'') (Seq (Read ''y'') (If (Eq (Sum (Variable ''x'') (Variable ''y'')) (Constant 0)) (Exec (Constant 1)) (Exec (Sum (Variable ''x'') (Sum (Variable ''y'') (Sum (Variable ''y'') (Variable ''x'')))))))))"
definition "ok10= (Seq (Read ''x'') (If (Eq (Variable ''x'') (Constant 0)) (Exec (Constant 1)) (Exec (Variable ''x'')))))"
definition "ok11= (Seq (Read ''x'') (Seq (If (Eq (Variable ''x'') (Constant 0)) (Aff ''x'' (Sum (Variable ''x'') (Constant 1))) Skip) (Exec (Variable ''x'')))))"
definition "ok12= (Seq (Aff ''x'' (Constant 1)) (Seq (Read ''z'') (If (Eq (Variable ''z'') (Constant 0)) (Exec (Variable ''y'')) (Exec (Variable ''z'')))))"
definition "ok13= (Seq (Aff ''z'' (Constant 4)) (Seq (Aff ''x'' (Constant 1)) (Seq (Read ''y'') (Seq (Aff ''x'' (Sum (Variable ''x'') (Sum (Variable ''z'') (Variable ''x'')))) (Seq (Aff ''z'' (Sum (Variable ''z'') (Variable ''x'')))) (Seq (If (Eq (Variable ''y'') (Constant 1)) (Aff ''x'' (Sub (Variable ''x'') (Variable ''y'')))) Skip) (Seq (If (Eq (Variable ''y'') (Constant 0)) (Seq (Aff ''y'' (Sum (Variable ''y'') (Constant 1))) (Exec (Variable ''x'')))) Skip) (Exec (Variable ''y'')))))))))"
definition "ok14= (Seq (Read ''x'') (Seq (Read ''y'') (If (Eq (Sum (Variable ''x'') (Variable ''y'')) (Constant 0)) (Exec (Constant 1)) (Exec (Sum (Variable ''x'') (Variable ''y'')))))))"


(* Le TP commence ici! *)

(*** Définition du prédicat bad ***)
fun BAD:: "(symTable * inchan * outchan) \<Rightarrow> bool"
where    (* dit si le résultat de l'évaluation d'un programme a provoqué un exec(0)  *)
"BAD (_,_,el) = (List.member  el (X 0))"


(*** Définition de l'analyseur statique V1 ***)
fun san1::"statement \<Rightarrow> bool"
where    (* un analyseur qui accepte un programme s'il ne comporte pas d'instruction exec *)
"san1 (Seq s1 s2) = ((san1 s1) \<and> (san1 s2))" |
"san1 (If _ s1 s2) = ((san1 s1) \<and> (san1 s2))"|
"san1 (Exec _) = False"|
"san1 _ = True"
```



```

(*** Définition de l'analyseur statique V2 ***)
fun san2::"statement \<Rightarrow> bool"
where   (* un analyseur qui accepte un programme s'il ne comporte que des exec sur des const antes différentes de 0 *)
"san2 (Seq s1 s2) = ((san2 s1) \<and> (san2 s2))" |
"san2 (If _ s1 s2) = ((san2 s1) \<and> (san2 s2))"|
"san2 (Exec (Constant c)) = ((c > 0) \<or> (c < 0))"|
"san2 (Exec _) = False"|
"san2 _ = True"

(***_ Définition de l'analyseur final _***)

  (** Debut **)
                (* Paire i *)
fun supDoubl::"(string * int) \<Rightarrow> symTable \<Rightarrow> symTable"
where (* methode qui supprime une paire i de la table des symbole s'il y a une de trop *)
"supDoubl _ [] = []"|
"supDoubl (e,s) ((e1,s2)#ts) = (if ((e =e1)\<and>(s = s2)) then ts else ((e1,s2)#(supDoubl (e,s) ts)))"
  (** Fin **)

  (** Debut **)
datatype etat = Vrai | Faux | Dknow | Result int
(**
 * sert à définir l'état de mon analyseur : eg: 'Dknow' si on sait pas qu'elle traitement faire, dans ce cas, faut tout traiter
 * read x ; if x > 1 ; ici l'état de mon analyseur est 'Dknow' donc faut traiter en premier temps le then, et le else bien sur.
**)
  (** Fin **)

  (** Debut **)
(* Require :: etat : Result e *)
fun deserilize::"etat \<Rightarrow> int"
where (* methode qui sert juste à donner la valeur de l'état *)
"deserilize (Result e) = e"|
"deserilize _ = 1" (* on a pas à se préocuper de cette ligne, car la méthode est appeler que quand etat est un (Result e)  *)
  (** Fin **)

  (** Debut **)
fun siExpRead::"expression \<Rightarrow> symTable \<Rightarrow> bool"
where (* methode qui renvoi vrai dans le cas où l'expression dépend d'un read, faux sinon*)
"siExpRead (Constant c) _= False"|
"siExpRead (Variable x) ts = (if ((List.member (supDoubl (x,0) ts) (x,0))) then True else False)"|
"siExpRead (Sum e1 e2) ts = ((siExpRead e1 ts) \<or> (siExpRead e2 ts))"|
"siExpRead (Sub e1 e2) ts = ((siExpRead e1 ts) \<or> (siExpRead e2 ts))"
  (** Fin **)

  (** Debut **)
fun evalExpCond ::"expression \<Rightarrow> symTable \<Rightarrow> etat"
where (* methode qui evalue les expressions d'une condtion, elle est appeler par la methode evalCond *)
"evalExpCond (Constant c) ts = (Result c)"|
"evalExpCond (Variable x) ts = (if (List.member (supDoubl (x,0) ts) (x,0)) then Dknow else (Result ((evalE (Variable x) ts))))"|
"evalExpCond (Sum (Constant c1) (Constant c2)) _ = (Result (c1+c2))" |
"evalExpCond (Sub (Constant c1) (Constant c2)) _  = (Result (c1-c2))" |
"evalExpCond (Sum (Variable x) (Constant c2)) ts = (if (List.member (supDoubl (x,0) ts) (x,0)) then Dknow
                                                    else (Result ((evalE (Variable x) ts) + c2)))"|
"evalExpCond (Sum (Variable x) (Variable y)) ts= (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or>
                                                      (List.member (supDoubl (y,0) ts) (y,0))) then Dknow
                                                     else (Result ((evalE (Variable x) ts) + ((evalE (Variable y) ts)))))"|
"evalExpCond (Sub (Variable x) (Constant c2)) ts = (if (List.member (supDoubl (x,0) ts) (x,0)) then Dknow
                                                    else (Result ((evalE (Variable x) ts) - c2)))"|
"evalExpCond (Sub (Variable x) (Variable y)) ts= (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or>
                                                      (List.member (supDoubl (y,0) ts) (y,0))) then Dknow
                                                     else (Result ((evalE (Variable x) ts) - ((evalE (Variable y) ts)))))"|
```

```
"evalExpCond (Sub e1 e2) ts = (if ( ((evalExpCond e1 ts)= Dknow) \<or> ((evalExpCond e2 ts)= Dknow) ) then Dknow
                                else(let d1 = (deserilize (evalExpCond e1 ts)) in (Result (d1 - (deserilize (evalExpCond e2 ts))))))  "|
"evalExpCond (Sum e1 e2) ts = (if ( ((evalExpCond e1 ts)= Dknow) \<or> ((evalExpCond e2 ts)= Dknow) ) then Dknow
                                else(let d1 = (deserilize (evalExpCond e1 ts)) in (Result (d1 + (deserilize (evalExpCond e2 ts))))))  "
  (** Fin **)

  (** Debut **)
fun evalCondition ::"condition \<Rightarrow> symTable \<Rightarrow> etat"
where (* methode qui evalue une condition, appeler à chaque fois qu'on rencontre un if, elle nous renvoi un etat,soit vrai, faux ou on sait pas 'Dknow' *)
"evalCondition (Eq (Constant c1) (Constant c2)) _ = (if (c1 = c2) then Vrai else Faux)"|
"evalCondition  (Eq (Variable x) (Constant c)) ts = (if (List.member (supDoubl (x,0) ts) (x,0)) then Dknow
                                                    else(if (evalC  (Eq (Variable x) (Constant c)) ts)
                                                            then Vrai
                                                           else Faux))"|
"evalCondition (Eq  (Constant c) (Variable x)) ts= (if (List.member (supDoubl (x,0) ts) (x,0)) then Dknow
                                                      else (if (evalC  (Eq  (Constant c) (Variable x)) ts)
                                                              then Vrai
                                                             else Faux))"|
"evalCondition (Eq (Variable x) (Variable y)) ts= (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or> (List.member (supDoubl (y,0) ts) (y,0)))
                                                    then Dknow
                                                   else (if (evalC  (Eq  (Variable x) (Variable y)) ts)
                                                            then Vrai
                                                           else Faux))"|
"evalCondition (Eq (Variable x) e) ts = (if ( ((evalExpCond e ts)= Dknow) \<or> (List.member (supDoubl (x,0) ts) (x,0)))
                                         then Dknow
                                        else(if ((deserilize (evalExpCond e ts)) = (evalE (Variable x) ts))
                                         then Vrai
                                         else Faux))"|
"evalCondition (Eq e (Variable x)) ts = (if ( ((evalExpCond e ts)= Dknow) \<or> (List.member (supDoubl (x,0) ts) (x,0))) then Dknow
                                          else(if ((deserilize (evalExpCond e ts)) = (evalE (Variable x) ts)) then Vrai else Faux))"|
"evalCondition (Eq e (Constant c)) ts = (if ((evalExpCond e ts)= Dknow) then Dknow
                                         else(if ((deserilize (evalExpCond e ts)) = c) then Vrai else Faux))"|
"evalCondition (Eq (Constant c) e) ts = (if ((evalExpCond e ts)= Dknow) then Dknow
                                         else(if ((deserilize (evalExpCond e ts)) = c) then Vrai else Faux))"|
"evalCondition (Eq e1 e2) ts =(if ( ((evalExpCond e1 ts)= Dknow) \<or> ((evalExpCond e2 ts)= Dknow) ) then Dknow
                               else (if ((evalExpCond e1 ts) = (evalExpCond e2 ts)) then Vrai else Faux))"
  (** Fin **)

   (** Debut **)
fun evalExec::"expression  \<Rightarrow> symTable \<Rightarrow> bool"
where (* methode qui evalue l'expression d'un exec. renvoi faux si exec de cette expression est dangereuse *)
"evalExec (Sub (Variable x) e) ts = (if (((List.member (supDoubl (x,0) ts) (x,0))) \<or> (siExpRead e ts)) then False
                                    else (((evalE (Sub (Variable x) e) ts) >0) \<or> ((evalE (Sub (Variable x) e) ts) < 0) ))"|
"evalExec (Sub  e (Variable x)) ts = (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or> (siExpRead e ts)) then False
                                     else (((evalE (Sub e  (Variable x)) ts) >0) \<or> ((evalE (Sub e  (Variable x)) ts) < 0) ))"|
"evalExec (Sum (Variable x) e) ts = (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or> (siExpRead e ts)) then False
                                    else (((evalE (Sum (Variable x) e) ts) >0) \<or> ((evalE (Sum (Variable x) e) ts) < 0) ))"|
"evalExec (Sum  e (Variable x)) ts = (if ((List.member (supDoubl (x,0) ts) (x,0)) \<or> (siExpRead e ts)) then False
                                     else (((evalE (Sum e  (Variable x)) ts) >0) \<or> ((evalE (Sum e  (Variable x)) ts) < 0) ))"|
"evalExec exp ts = (if (siExpRead exp ts) then False else (((evalE exp ts) >0) \<or> ((evalE exp ts) < 0)))"
  (** Fin **)
```



```


  (** Debut **)
fun sanIIIaux::"string \<Rightarrow> expression \<Rightarrow> symTable \<Rightarrow>symTable "
where (* methode appeler si une variable est définit à partir d'une expression, et renvoi une table de symbole *)
"sanIIIaux str (Sum e1 e2) ts = ((let tsaux = (sanIIIaux str e1 ts) in (* si e1 possede une variable qui depend d'un read str(c'est à dire notre variable  sera influencer *)
                                  (if ((List.member (supDoubl (str,0) tsaux) (str,0))) then
                                    (sanIIIaux str e2 ((str,0)#tsaux))
                                  else((let tsaux2 = (sanIIIaux str e2 ts) in (* si e2 possede une variable qui depend d'un read str sera influencer *)
                                    (if((List.member (supDoubl (str,0) tsaux2) (str,0))) then
                                      (sanIIIaux str e2 ((str,0)#tsaux))
                                     else ((str,(evalE (Sum e1 e2) ts))#ts)))))))"|
"sanIIIaux str (Sub e1 e2) ts = ((let tsaux = (sanIIIaux str e1 ts) in (* si e1 possede une variable qui depend d'un read str sera influencer *)
                                  (if ((List.member (supDoubl (str,0) tsaux) (str,0))) then
                                    (sanIIIaux str e2 ((str,0)#tsaux))
                                  else((let tsaux2 = (sanIIIaux str e2 ts) in (* si e2 possede une variable qui depend d'un read str sera influencer *)
                                    (if((List.member (supDoubl (str,0) tsaux2) (str,0))) then
                                      (sanIIIaux str e2 ((str,0)#tsaux))
                                    else ((str,(evalE (Sub e1 e2) ts))#ts)))))))"|
"sanIIIaux str (Variable str2) ts = (if ((List.member (supDoubl (str2,0) ts) (str2,0)))  then (let l = (str,0)#ts in (str,0)#l)
                                     else let l =(evalE (Variable str2) ts) in if(l =0 \<and> (str = str2)) then ts else (str,l)#ts)"|
"sanIIIaux str (Constant c) ts = (if ((List.member (supDoubl (str,0) (supDoubl (str,0) ts)) (str,0)))then (supDoubl (str,0) ts)
                                   else if ((List.member (supDoubl (str,0) ts) (str,0))) then (str,c)#(supDoubl (str,0) (supDoubl (str,0) ts)) else  ((str,c)#(supDoubl (str,0) ts)))"
  (* Fin *)

  (* Debut *)
fun san3TabSymb::"statement \<Rightarrow> symTable \<Rightarrow> symTable"
where (* methode qui constuit la table des symboles d'un statement à partir d'une table de symbole *)
"san3TabSymb (Seq s1 s2) ts = (let ts2 = (san3TabSymb s1 ts) in  (san3TabSymb s2 ts2))" |
"san3TabSymb (If c s1 s2) ts = (if(evalC c ts) then (san3TabSymb s1 ts) else (san3TabSymb s2 ts))"|
"san3TabSymb (Read str) ts = (let l = (str,0)#ts in (str,0)#l )"|
"san3TabSymb (Aff str exp) ts =(sanIIIaux str exp ts)"|
"san3TabSymb _ ts = ts"
  (* Fin *)

  (* Debut *)
fun san3aux2::"statement \<Rightarrow> symTable \<Rightarrow> bool"
where (* methode qui traite un programme, appeler par la methode safe avec le programme et une table de symbole vide, qui sera construite au fure et à mesure du traitement *)

"san3aux2 (Seq (Seq s1 s2) s3) ts = (let ts2 = (san3TabSymb (Seq s1 s2 ) ts) in ((san3aux2 (Seq s1 s2 ) ts) \<and> (san3aux2 s3 ts2)))"|
                                     (* la faut evaluer la condition d'abord *)
"san3aux2 (Seq (If c s1 s2) s3) ts = (if ((evalCondition c ts) = Vrai) then
                                       (let ts0 = (san3TabSymb s1 ts) in ((san3aux2 s1 ts0) \<and> (san3aux2 s3 ts0)))
                                     else if((evalCondition c ts) = Faux) then
                                       (let ts1 = (san3TabSymb s2 ts) in ((san3aux2 s2 ts1) \<and> (san3aux2 s3 ts1)))
                                     else(
                                        (let ts2 = (san3TabSymb s3 ts)  in ( ((san3aux2 (If c s1 s2) ts) \<and>
                                         (san3aux2 s3 ((san3TabSymb s1 ts)@ts2))) \<and> (san3aux2 s3 ((san3TabSymb s2 ts)@ts2)) ))))"|
"san3aux2 (Seq  s3 (If c s1 s2)) ts =(let ts2 = (san3TabSymb s3 ts) in((san3aux2 s3 ts2) \<and> (san3aux2 (If c s1 s2) ts2)))"|
"san3aux2 (Seq s1 s2) ts = (let ts2 = (san3TabSymb s1 ts) in ((san3aux2 s1 ts2) \<and> (san3aux2 s2 ts2)) )"|
                                     (* la faut evaluer la condition *)
"san3aux2 (If c s1 s2) ts = (if ((evalCondition c ts) = Vrai) then
                                       (san3aux2 s1 ts)
                                     else if((evalCondition c ts) = Faux) then
                                       (san3aux2 s2 ts)
                                     else(
                                        ((san3aux2 s1 (san3TabSymb s1 ts)) \<and> (san3aux2 s2 ((san3TabSymb s2 ts))) ))"|
"san3aux2 (Exec  e) ts = (evalExec e ts)"|
```

```
"san3aux2 _ _ = True"
  (* Fin *)

  (* Debut *)
fun safe::"statement \<Rightarrow> bool"
where   (* un analyseur qui accepte un programme dès lors que ses exec portent sur des expressions qui ne s'évaluent jamais a 0 *)
"safe p = (san3aux2 p  [])"
  (* Fin *)

(*

(*** Completude ***)
lemma lmComp:"(\<forall> input.\<forall> prog.(\<not>(BAD (evalS prog ([],input,[])))  \<longrightarrow> san prog))"
nitpick [timeout=28800]
sorry
(*
quickcheck [tester=narrowing, timeout=2800, size=15]
sorry
*)
*)

(*

(*** Correction ***)
lemma lmCor:"(\<forall> input.\<forall> prog. (safe prog\<longrightarrow> \<not>(BAD (evalS prog ([],input,[])))  ) )"
nitpick [timeout=28800]
sorry
(*
quickcheck [tester=narrowing, timeout=2800, size=15]
sorry
*)
*)


(* ----- Restriction de l'export Scala (Isabelle 2014) -------*)
(* ! ! !  NE PAS MODIFIER ! ! ! *)
(* Suppression de l'export des abstract datatypes (Isabelle 2014) *)

code_reserved Scala
  expression condition statement
code_printing
  type_constructor expression \<rightharpoonup> (Scala) "expression"
  | constant Constant \<rightharpoonup> (Scala) "Constant"
  | constant Variable \<rightharpoonup> (Scala) "Variable"
  | constant Sum \<rightharpoonup> (Scala) "Sum"
  | constant Sub \<rightharpoonup> (Scala) "Sub"

  | type_constructor condition \<rightharpoonup> (Scala) "condition"
  | constant Eq \<rightharpoonup> (Scala) "Eq"

  | type_constructor statement \<rightharpoonup> (Scala) "statement"
  | constant Seq \<rightharpoonup> (Scala) "Seq"
  | constant Aff \<rightharpoonup> (Scala) "Aff"
  | constant Read \<rightharpoonup> (Scala) "Read"
  | constant Print \<rightharpoonup> (Scala) "Print"
  | constant Exec \<rightharpoonup> (Scala) "Exec"
  | constant If \<rightharpoonup> (Scala) "If"
  | constant Skip \<rightharpoonup> (Scala) "Skip"
  | code_module "" \<rightharpoonup> (Scala)

{*// Code generated by Isabelle
package tp67

import utilities.Datatype._
// automatic conversion of utilities.Datatype.Int.int to Int.int
object AutomaticConversion{
  implicit def int2int(i:utilities.Datatype.Int.int):Int.int =
      i match {
      case utilities.Datatype.Int.int_of_integer(i)=>Int.int_of_integer(i)
  }
}
import AutomaticConversion._
*}


(* Directive pour l'exportation de l'analyseur *)

```



```
(*
export_code safe in Scala
file "~/Bureau/safe.scala"   (* à adapter en fonction du chemin de votre projet TP67 *)
*)

end
```