Trabalho 2015/2 - Semântica Formal
Gerador de Condições de Verificaçãoo para IMP

- Trabalho a ser feito de forma individual
- Submissão pelo Moodle até as 23:59 minutos do dia **4 de dezembro**.

TRABALHO - implementar em OCaml um gerador de condições de verificaçãoo para a linguagem IMP seguindo estritamente a definição dada em aula para a funçãoo `vcgen`. A linguagem IMP, porém, pode ser aumentada com novos operadores e a linguagem de asserções pode ser aumentada com novos símbolos funcionais e predicativos. Para implementar a funçãoo `vcgen` devem ser implementas também a função `wpc`, para obtenção de pré-condiçãoo mais fraca de comandos, e `vcg`.

Abaixo segue código para definição das árvores de sintaxe abstrata da linguagem IMP

```
type status = Progr | Logical

type aexp =
       Var of string * status
     | Num of int
     | Sum of aexp * aexp
     | Mult of aexp * aexp
     | Min of aexp * aexp
     | Fat of aexp

type bexp =
      PBool of bool
    | POr of bexp * bexp
    | PAnd of bexp * bexp
    | PNot of bexp
    | PEq of aexp * aexp
    | PLeq of aexp * aexp
    | PUneq of aexp * aexp

type astn =
      ABool of bool
    | AOr of astn * astn
    | AAnd of astn * astn
    | ANot of astn
    | AImpl of astn * astn
    | AEq of aexp * aexp
    | ALeq of aexp * aexp
    | AUneq of aexp * aexp

type cmd =
      Skip
    | Asg of string * aexp
    | Seq of cmd * cmd
    | If of bexp * cmd * cmd
    | Wh of bexp * astn * cmd   (* annotated while command *)
```

As funções auxiliares abaixo convertem expressões aritméticas e asserções em strings

```
let rec aexpTostr (a:aexp) = match a with
      Var(x,_) → x
    | Num(n) → string_of_int n
    | Sum(a1, a2) → "(" ^ (aexpTostr a1) ^ " + " ^ (aexpTostr a2) ^ ")"
    | Mult(a1, a2) → "(" ^ (aexpTostr a1) ^ " x " ^ (aexpTostr a2) ^ ")"
    | Min(a1, a2) → "(" ^ (aexpTostr a1) ^ " - " ^ (aexpTostr a2) ^ ")"
    | Fat(a1) → "(" ^ (aexpTostr a1) ^ "!)"

let rec astnTostr (a:astn) = match a with
    ABool (true) → "T"
  | ABool(false) → "F"
  | AOr (f, g) → "(" ^ (astnTostr f) ^ " or " ^ (astnTostr g) ^ ")"
  | AAnd (f, g) → "(" ^ (astnTostr f) ^ " and " ^ (asntTostr g) ^ ")"
  | ANot f → "(not " ^ (astnTostr f) ^ ")"
  | AImpl (f, g) → "(" ^ (astnTostr f) ^ " ==> " ^ (astnTostr g) ^ ")"
  | AEq(a1, a2) → "(" ^ (aexpTostr a1) ^ " == " ^ (aexpTostr a2) ^ ")"
  | ALeq(a1,a2) → "(" ^ (aexpTostr a1) ^ " <= " ^ (aexpTostr a2) ^ ")"
  | AUneq(a1,a2) → "(" ^ (aexpTostr a1) ^ " <> " ^ (aexpTostr a2) ^ ")"
```

A função auxiliar abaixo converte árvores de sintaxe abstrata de expressões boleanas de IMP em fórmulas da linguagem lógica das asserções:

```
let rec bexpToastn = function
    | PBool(t) → ABool(t)
    | POr(t1,t2) → (AOr(bexpToastn(t1), bexpToastn(t2)))
    | PAnd(t1,t2) →(AAnd(bexpToastn(t1), bexpToastn(t2)))
    | PNot(t) → ANot(bexpToastn(t))
    | PEq(t1,t2) → AEq(t1,t2)
    | PLeq(t1,t2) → ALeq(t1,t2)
    | PUneq(t1,t2) → AUneq(t1,t2)
```

As funções abaixo implementam substituição em expressões aritméticas e em asserções respectivamente.

```
let rec asubst x (a:aexp) = function
      | Var (y,s) → if ((x = y) && (s = Progr)) then a else Var (y,s)
      | Num(n) → Num n
      | Sum(a1, a2) → Sum (asubst x a a1, asubst x a a2)
      | Mult(a1, a2) → Mult (asubst x a a1, asubst x a a2)
      | Min(a1, a2) → Min (asubst x a a1, asubst x a a2)
      | Fat(a1) → Fat(asubst x a a1)

let rec subst x (a:exp) = function
    | ABool(b) → ABool(b)
    | AOr(t1,t2) → AOr(subst x a t1, subst x a t2)
    | AAnd(t1,t2) → AAnd (subst x a t1, subst x a t2)
    | ANot(t) → ANot(subst x a t)
    | AImpl(t1,t2) → AImpl(subst x a t1, subst x a t2)
    | AEq(t1,t2) → AEq(asubst x a t1, asubst x a t2)
    | ALeq(t1, t2) → ALeq(asubst x a t1, asubst x a t2)
    | AUneq(t1, t2) → AUneq(asubst x a t1, asubst x a t2)
```

Substituição em asserções ocorrem no cálculo de pré-condição mais fraca de comandos de atribuição.

Para testar o programa devem ser coletadas as condições de verificação do programas da lista de exercícios. A árvore de sintaxe abstrata para o programa `fat` abaixo

```
y := 1;
z := 0;
while (z <> x) (
  invariante(y = z!)
  z:= z +1;
  y:= y * z
)
```

é representada da seguinte forma em OCaml:

```
let fat =
Seq(Seq(Asg("y", Num 1), Asg("z", Num 0)),
        Wh(PUneq(Var("z", Progr), Var("x", Progr)),
                AEq(Var("y", Progr), Fat(Var("z", Progr))),
                Seq(Asg("z", Sum(Var("z", Progr), Num 1)),
                        Asg("y", Mult(Var("y", Progr), Var("z", Progr))))))
```

A seguinte chamada de `vcgen` deve retornar a fórmula com as condições de verificação dadas $0 \leq x$ e $x =!y$ como pré e pós-condições, respectivamente:

```
vcgen (ALeq(Num 0, Var("x", Progr)),
              fat,
        AEq(Var("y", Progr), Fat(Var("x",Progr))))
```