```
//week 8 intepreter
(* Interpreter for a simple WHILE-language.           MRH 21/10 2013 *)
(* Program skeleton                                                  *)
(* Based on a natural semantics of WHILE                             *)

type AExp =
    (* Arithmetical expressions *)
    | N of int                     (* numbers                  *)
    | V of string                  (* variables                *)
    | Add of AExp * AExp           (* addition                 *)
    | Mul of AExp * AExp           (* multiplication           *)
    | Sub of AExp * AExp           (* subtraction              *)


type BExp =
    (* boolean expressions      *)
    | TT                           (* true                     *)
    | FF                           (* false                    *)
    | Eq of AExp * AExp            (* equality                 *)
    | Lt of AExp * AExp            (* less than                *)
    | Neg of BExp                  (* negation                 *)
    | Con of BExp * BExp           (* conjunction              *)

type Stm =
    (* statements             *)
    | Ass of string * AExp         (* assignment             *)
    | Skip
    | Seq of Stm * Stm             (* sequential composition *)
    | ITE of BExp * Stm * Stm      (* if-then-else           *)
    | While of BExp * Stm          (* while                  *)



type State = Map<string, int>

(* update: string -> int -> State -> State   *)
let update x v s = Map.add x v s

(* A: AExp -> State -> int                   *)
let rec A a s =
    match a with
    | N n -> n
    | V x -> Map.find x s
    | Add (a1, a2) -> A a1 s + A a2 s
    | Mul (a1, a2) -> A a1 s * A a2 s
    | Sub (a1, a2) -> A a1 s - A a2 s

(* B: BExp -> State -> bool                  *)
let rec B b s =
    match b with
    | TT -> true
    | FF -> false
    | Eq (a1, a2) -> if A a1 s = A a2 s then true else false
    | Lt (a1, a2) -> if A a1 s < A a2 s then true else false
    | Neg (bexp) -> not (B bexp s)
    | Con (b1, b2) -> B b1 s && B b2 s

(* I: Stm -> State -> State                          *)
let rec I stm s =
    match stm with
    | Ass (x, a) -> update x (A a s) s
    | Skip -> s
    | Seq (stm1, stm2) -> I stm2 (I stm1 s)
    | ITE (b, stm1, stm2) -> if B b s then I stm1 s else I stm2 s
    | While (b, stm1) -> if B b s then I stm (I stm1 s) else s

let fac =
    Seq(Ass("y", N 1), While(Neg(Eq(V "x", N 0)), Seq(Ass("y", Mul(V "x", V "y")), Ass("x", Sub(V "x", N 1)))))

(* Define an initial state                           *)
let s0 = Map.ofList [ ("x", 4) ]

(* Interpret the program                             *)
let s1 = I fac s0

(* Inspect the resulting state                       *)
Map.find "y" s1
```