Programming Languages

Cimbir Cristian VR386661
Colognese Marco VR386474
Rossini Mattia VR386327

Indice

Introduction

The project extends the interpreter of the imperative and functional language including blocks, procedures and functions.

It has been extended compared to the one seen in class with:

- *String* type;
- functions for *Strings* required by the specifics;
- functions used by *Reflect* command;
- more (not required) functions on *Strings*;
- *Char* type.

This project is written in ***OCaml*** using Denotational Semantic.

Considerations:
Not all the expressions are implemented into the *Reflect*;
The *String* that the *Reflect* analyzes must be without spaces.

New types

String type

```
| Estring of string                    (* Add Estring in exp *)
| Len of exp                           (* Constructor of Length *)
| Concat of exp * exp                  (* Constructor of Concatenation *)
| Substr of exp * exp * exp            (* Constructor of Substr *)
| Equals of exp * exp                  (* Constructor of Equals *)
| Trim of exp                          (* Constructor of Trim *)
| Uppercase of exp                     (* Constructor of Uppercase *)
| Lowercase of exp                     (* Constructor of Lowercase *)
```

These expressions are added to extend the *exp* type.
The *Estring* type is an expression that rappresents the *String* type.
This allows us to implement the type checking and use the other functions we have implemented in the interpreter.

Char type

```
| Echar of char                       (* Add Echar in exp *)
| Contain of exp * exp                (* Constructor of Contain *)
```

We have also implemented the *Char* type to introduce the *contain* function.

Functions

Here are all the functions requested by the specifics that we have implemented and also many other that we implemented to expand o interpreter.
Every function is paired with it's ***OCaml*** code and a short description.

String functions

Len x

```
(* INPUT: string        OUTPUT: string's length *)
let len x =
    if typecheck("string", x)
        then (match x with String(u) -> Int(String.length u))
        else failwith("type error")
```

The *len* function takes in input an expression and do a *type check* on it, if it is a *String* type, it returns its length.

Concat (x, y)

```
(* INPUT: (string, string)      OUTPUT: the concatenation of the two strings *)
let concat (x,y) =
    if typecheck("string",x) && typecheck("string",y)
        then (match (x,y) with (String(u),String(w)) -> String(String.concat "" [u;w]) )
        else failwith("type error")
```

The *concat* function takes in input two expressions and do a *type check* on them, if they are *String* types, it returns the concatenation of the two strings.

Substr (x, y, z)

```
(* INPUT: (string, start, end)      OUTPUT: the substring starting from "start" to "end" *)
let substr (x,y,z) =
    if typecheck("string",x) && typecheck("int",y) && typecheck("int",z)

        then (match (x,y,z) with (String(s),Int(i),Int(j)) -> String(
            if (j-i)>0
                then    String.sub s i (j-i+1)
                else    ""
        ))
        else failwith("type error")
```

The *substr* function takes in input three expressions and do a *type check* on them, if the first expression is a *String* type and the other two are *Integers*, it returns the substring of x from index y to z.
If $y > z$ it returns an empty string.

Other functions

The following functions are added to the ones that are requested for this project. Some of these will be used from the *Reflect* command.

Trim x

```
(* INPUT: string        OUTPUT: the string without spaces at the edges *)
let trim x =
    if typecheck("string", x)
        then (match x with String(u) -> String(String.trim u))
        else failwith("type error")
```

The *trim* function takes in input an expression and do a *type check* on it, if it is a *String* type, it returns the same string without spaces at the edges.

Uppercase x

```
(* INPUT: string        OUTPUT: the string in uppercase *)
let uppercase x =
    if typecheck("string", x)
        then (match x with String(u) -> String(String.uppercase_ascii u))
        else failwith("type error")
```

The *uppercase* function takes in input an expression and do a *type check* on it, if it is a *String* type, it returns the same string in uppercase.

Lowercase x

```
(* INPUT: string        OUTPUT: the string in lowercase *)
let lowercase x =
    if typecheck("string", x)
        then (match x with String(u) -> String(String.lowercase_ascii u))
        else failwith("type error")
```

The *lowercase* function takes in input an expression and do a *type check* on it, if it is a *String* type, it returns the same string in lowercase.

Equals (x, y)

```
(* INPUT: (string, string)      OUTPUT: "true" if strings are equals, "false" otherwise *)
let equals (x, y) =
    if typecheck("string",x) && typecheck("string",y)
        then (match (x,y) with (String(u),String(w)) -> Bool(String.equal u w) )
    else failwith("type error")
```

The *equals* function takes in input two expressions and do a *type check* on them, if they are *String* types, it returns *true* if the strings are equals, *false* otherwise.

Contain (x, y)

```
(* INPUT: (string, char)       OUTPUT: "true" if the string contains the character, "false" otherwise *)
let contain (x, y) =
    if typecheck("string",x) && typecheck("char",y)
        then (match (x,y) with (String(u),Char(w)) -> Bool(String.contains u w) )
    else failwith("type error")
```

The *contain* function takes in input two expressions and do a *type check* on them, if the first is a *String* type and the other one is a *Char* type, it returns *true* if the string contains the character, *false* otherwise.

Explode s

```
(* INPUT: string        OUTPUT: converts the string into a list of char *)
let explode s =
    let rec expl i l =
        if i < 0 then l else
            expl (i - 1) (s.[i] :: l) in
    expl (String.length s - 1) [];;
```

The *explode* function takes in input a *String* and converts it into a list of *Char*.

Implode l

```
(* INPUT: list of char      OUTPUT: converts the list of char into a string *)
let implode l =
    let result = String.create (List.length l) in
    let rec imp i = function
    | [] -> result
    | c :: l -> result.[i] <- c; imp (i + 1) l in
    imp 0 l;;
```

The *implode* function takes in input a list of *Char* and converts it into a *String*.

ReplaceChar (s, c1, c2)

```
(* INPUT: string, charToReplace, newChar       OUTPUT: string without charToReplace, replaced by newChar *)
let replaceChar s c1 c2 =
    let sl=String.length s in
    let rec loop i =
        if i<0 then s
        else if s.[i]==c1 then let a = s.[i]<-c2 in
            loop (i-1)
        else loop (i-1)
    in
    loop (sl - 1)
```

The *replaceChar* function takes in input a *String* and two *Char* and returns the string without *Char* $c1$, replaced by *Char* $c2$.

Reflect

```
(* INPUT: string        OUTPUT: commands list represented by the string *)
and reflect (s:string) = List.map stringToCom (stringSplitToList ';' s)
```

The *reflect* function takes in input a *String* and returns the commands list represented by the *String* and splitted by semicolons.

StringSplitToList (c, s)

```
(* INPUT: char string    OUTPUT: divides the string using the given character, ignoring characters inside brackets *)
let stringSplitToList c s =
    let sl = String.length s in
    let rec loop fine parentesi parentesiQ t i=
        if i<0 then (String.sub s (i+1) (fine-(i)) :: t)
        else if s.[i]==c && parentesi==0 && parentesiQ==0 then
            loop (i-1) (parentesi) (parentesiQ) (String.sub s (i+1) (fine-(i)) :: t) (i-1)
        else if s.[i]=='(' then loop (fine) (parentesi-1) (parentesiQ) t (i-1)
        else if s.[i]==')' then loop (fine) (parentesi+1) (parentesiQ) t (i-1)
        else if s.[i]=='[' then loop (fine) (parentesi) (parentesiQ-1) t (i-1)
        else if s.[i]==']' then loop (fine) (parentesi) (parentesiQ+1) t (i-1)
        else loop (fine) (parentesi) (parentesiQ) t (i-1)
    in
    loop (String.length s - 1) 0 0 [] (String.length s - 1)
```

The *stringSplitToList* function takes in input a *Char* and a *String*. It returns the *string list* represented by the *String s* divided using the character *c* and ignoring the characters inside brackets.

StringToExp x

```
(* INPUT: string        OUTPUT: expression represented by the string *)
let rec stringToExp x =
    let s = explode x in
    match s with
    |   'E'::'i'::'n'::'t'::'('::r ->   Eint(int_of_string (String.sub (implode r) 0 (String.length (implode r) - 1)) )
    |   'E'::'s'::'t'::'r'::'i'::'n'::'g'::'('::r ->  Estring(String.sub (implode r) 1 ((String.length (implode r)) - 3))
    |   'E'::'b'::'o'::'o'::'l'::'('::r ->  Ebool(bool_of_string (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'D'::'e'::'n'::'('::r ->    Den(String.sub (implode r) 1 ((String.length (implode r)) - 3))
    |   'V'::'a'::'l'::'('::r ->    Val(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'N'::'o'::'t'::'('::r ->    Not(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'E'::'q'::'('::r ->
                let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Eq(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    |   'I'::'s'::'z'::'e'::'r'::'o'::'('::r -> Iszero(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'P'::'r'::'o'::'d'::'('::r ->
                let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Prod(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    |   'D'::'i'::'f'::'f'::'('::r ->
                let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Diff(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    | _ -> failwith ("Errore sintassi, stringToExp, " ^ (implode s))
```

The *StringToExp* function takes in input a *String* and trasforms it into a *char list* to perform a match operation to return the right expression represented by the *String*.

StringToCom x

```
(* INPUT: string        OUTPUT: command represented by the string *)
let rec stringToCom x =
    let s = explode x in
    match s with
    |   'A'::'s'::'s'::'i'::'g'::'n'::'('::r ->
            let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                else Assign(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))

    |   'C'::'i'::'f'::'t'::'h'::'e'::'n'::'e'::'l'::'s'::'e'::'('::r ->
            let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 3  then failwith ("Errore sintassi, " ^ (implode s))
                else Cifthenelse(stringToExp(List.hd l) ,
                    reflect(String.sub (List.hd (List.tl l)) 1 (String.length (List.hd (List.tl l)) - 2)),
                    reflect(String.sub (List.hd (List.tl (List.tl l))) 1 (String.length (List.hd (List.tl (List.tl l))) - 2))
                    )

    |   'W'::'h'::'i'::'l'::'e'::'('::r ->
            let l = stringSplitToList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s))
                else While(stringToExp(List.hd l) ,
                                reflect(String.sub (List.hd (List.tl l)) 1 (String.length (List.hd (List.tl l)) - 2))
                                )

    | _ -> failwith ("Errore sintassi, stringToCom, " ^ (implode s))
```

The *stringToCom* function takes in input a *String* and trasforms it into a *char list* to perform a match operation to return the right command represented by the *String*.

Examples

The following examples are made to test the functions we have implemented. Every test is paired with it's ***OCaml*** code and the shell's output.

Tests on *String* functions

Test *Len*

```
(* 1 - TEST len *)
let s1 = Estring("Linguaggi");;
let ex1 = Len s1;;
let rho1 = Funenv.emptyenv(Unbound);;
let sigma1 = Funstore.emptystore(Undefined);;
let result1 = sem ex1 rho1 sigma1;;
```

```
val s1 : exp = Estring "Linguaggi"
val ex1 : exp = Len (Estring "Linguaggi")
val rho1 : dval Funenv.env = <abstr>
val sigma1 : mval Funstore.store = <abstr>
val result1 : eval = Int 9
```

Test *Concat*

```
(* 2 - TEST concat *)
let s2 = Estring(" di Programmazione");;
let ex2 = Concat(s1, s2);;
let rho2 = Funenv.emptyenv(Unbound);;
let sigma2 = Funstore.emptystore(Undefined);;
let result2 = sem ex2 rho2 sigma2;;
```

```
val s2 : exp = Estring " di Programmazione"
val ex2 : exp = Concat (Estring "Linguaggi", Estring " di Programmazione")
val rho2 : dval Funenv.env = <abstr>
val sigma2 : mval Funstore.store = <abstr>
val result2 : eval = String "Linguaggi di Programmazione"
```

Test *Substr*

```
(* 3 - TEST substr *)
let ex3 = Substr(ex2, Eint(1), Eint(5));;
let rho3 = Funenv.emptyenv(Unbound);;
let sigma3 = Funstore.emptystore(Undefined);;
let result3 = sem ex3 rho3 sigma3;;
```

```
val ex3 : exp =
  Substr (Concat (Estring "Linguaggi", Estring " di Programmazione"),
   Eint 1, Eint 5)
val rho3 : dval Funenv.env = <abstr>
val sigma3 : mval Funstore.store = <abstr>
val result3 : eval = String "ingua"
```

Test *Trim*

```
(* 4 - TEST trim *)
let s4 = Estring("   Linguaggi di Programmazione  ");;
let ex4 = Trim(s4);;
let rho4 = Funenv.emptyenv(Unbound);;
let sigma4 = Funstore.emptystore(Undefined);;
let result4 = sem ex4 rho4 sigma4;;
```

```
val s4 : exp = Estring "   Linguaggi di Programmazione  "
val ex4 : exp =
      Trim (Estring "   Linguaggi di Programmazione  ")
val rho4 : dval Funenv.env = <abstr>
val sigma4 : mval Funstore.store = <abstr>
val result4 : eval = String "Linguaggi di Programmazione"
```

Test *Uppercase*

```
(* 5 - TEST uppercase *)
let ex5 = Uppercase(s1);;
let rho5 = Funenv.emptyenv(Unbound);;
let sigma5 = Funstore.emptystore(Undefined);;
let result5 = sem ex5 rho5 sigma5;;
```

```
val ex5 : exp = Uppercase (Estring "Linguaggi")
val rho5 : dval Funenv.env = <abstr>
val sigma5 : mval Funstore.store = <abstr>
val result5 : eval = String "LINGUAGGI"
```

Test *Lowercase*

```
(* 6 - TEST lowercase *)
let ex6 = Lowercase(s1);;
let rho6 = Funenv.emptyenv(Unbound);;
let sigma6 = Funstore.emptystore(Undefined);;
let result6 = sem ex6 rho6 sigma6;;
```

```
val ex6 : exp = Lowercase (Estring "Linguaggi")
val rho6 : dval Funenv.env = <abstr>
val sigma6 : mval Funstore.store = <abstr>
val result6 : eval = String "linguaggi"
```

Test *Equals*

```
(* 7 - TEST equals *)
let s7 = Estring("Linguaggi");;
let st7 = Estring("Linguaggi");;
let ex7 = Equals(s7,st7)
let rho7 = Funenv.emptyenv(Unbound);;
let sigma7 = Funstore.emptystore(Undefined);;
let result7 = sem ex7 rho7 sigma7;;
```

```
val s7 : exp = Estring "Linguaggi"
val st7 : exp = Estring "Linguaggi"
val ex7 : exp = Equals (Estring "Linguaggi", Estring "Linguaggi")
val rho7 : dval Funenv.env = <abstr>
val sigma7 : mval Funstore.store = <abstr>
val result7 : eval = Bool true
```

Test *Contain*

```
(* 8 - TEST contain *)
let c8 = Echar('g');;
let ex8 = Contain(s7,c8);;
let rho8 = Funenv.emptyenv(Unbound);;
let sigma8 = Funstore.emptystore(Undefined);;
let result8 = sem ex8 rho8 sigma8;;
```

```
val c8 : exp = Echar 'g'
val ex8 : exp = Contain (Estring "Linguaggi", Echar 'g')
val rho8 : dval Funenv.env = <abstr>
val sigma8 : mval Funstore.store = <abstr>
val result8 : eval = Bool true
```

Test *Explode*

```
(* 9 - TEST explode *)
let s9 = "Linguaggi";;
let ex9 = explode s9;;
```

```
val s9 : string = "Linguaggi"
val ex9 : char list = ['L'; 'i'; 'n'; 'g'; 'u'; 'a'; 'g'; 'g'; 'i']
```

Test *Implode*

```
(* 10 - TEST implode *)
let ex10 = implode ex9;;
```

```
val ex10 : bytes = "Linguaggi"
```

Test *StringSplitToList*

```
(* 11 - TEST stringSplitToList *)
let s11 = "Linguaggi";;
let c11 = 'g';;
let result11 = stringSplitToList c11 s11;;
```

```
val s11 : string = "Linguaggi"
val c11 : char = 'g'
val result11 : string list = ["Lin"; "ua"; ""; "i"]
```

Test *ReplaceChar*

```
(* 12 - TEST replaceChar *)
let c12 = 'a';;
let result12 = replaceChar s11 c11 c12;;
```

```
val c12 : char = 'a'
val result12 : bytes = "Linauaaai"
```

Test *StringToExp*

```
(* 13 - TEST stringToExp *)
let s13 = "Estring(\"Linguaggi\")";;
let st13 = stringToExp s13;;
let rho13 = Funenv.emptyenv(Unbound);;
let sigma13 = Funstore.emptystore(Undefined);;
let result13 = sem st13 rho13 sigma13;;
```

```
val s13 : string = "Estring(\"Linguaggi\")"
val st13 : exp = Estring "Linguaggi"
val rho13 : dval Funenv.env = <abstr>
val sigma13 : mval Funstore.store = <abstr>
val result13 : eval = String "Linguaggi"
```

Test *StringToCom*

```
(* 14 - TEST stringToCom *)
let d14 = [("x",Newloc(Eint 4));("y",Newloc(Eint 1))];;
let (rho14, sigma14) = semdv d14 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
let s14 = "While(Not(Eq(Val(Den(\"x\")),Eint(0))),[Assign(Den(\"y\"),Prod(Val(Den(\"y\")),
          Val(Den(\"x\"))));Assign(Den(\"x\"),Diff(Val(Den(\"x\")),Eint(1)))])";;
let es14 = stringToCom s14;;
let result14 = semc es14 rho14 sigma14;;
let result14Y = sem (Val(Den "y")) rho14 result14;;
```

```
val d14 : (string * exp) list =
  [("x", Newloc (Eint 4)); ("y", Newloc (Eint 1))]
val rho14 : dval Funenv.env = <abstr>
val sigma14 : mval Funstore.store = <abstr>
val s14 : string =
  "While(Not(Eq(Val(Den(\"x\")),Eint(0))),
    [Assign(Den(\"y\"),Prod(Val(Den(\"y\")),Val(Den(\"x\"))));
    Assign(Den(\"x\"),Diff(Val(Den(\"x\")),Eint(1)))])"
val es14 : com =
  While (Not (Eq (Val (Den "x"), Eint 0)),
    [Assign (Den "y", Prod (Val (Den "y"), Val (Den "x")));
     Assign (Den "x", Diff (Val (Den "x"), Eint 1))])
val result14 : mval Funstore.store = <abstr>
val result14Y : eval = Int 24
```

Test on *Block*, *Functions* and *Procedures*

```
(* 17 - TEST Block, procedure, function *)
(*
    int r=0
    mul2(int x)
        return x*2
    testproc(int n){
        w=1
        w=n+1
        r=mul(w)
    }
    testproc(4)
*)
let(es17: block) =
 (  ([],
    [
        ("mult2", Fun(["x"],
                Prod(Eint 2, Den "x"))
        );
        ("procedure", Proc( ["n"],
                ((([("z", Newloc(Den "n"));("w", Newloc(Eint 1))],
                []),
                [
                    Assign(Den "w",Sum(Val(Den "z"),Val(Den "w")));
                    Assign (Den "r", Appl (Den "mult2", [Val(Den "w")]))
                ])
                )
        )
    ]),
    [ Call(Den "procedure", [Eint 4])]
);;

let dr17 = [("r",Newloc(Eint 0))];;
let (rho17, sigma17) = semdv dr17 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
let result17 = semb es17 rho17 sigma17;;
let result17R = sem (Val(Den "r")) rho17 result17;;
```

```
val es17 : block =
  (([],
    [("mult2", Fun (["x"], Prod (Eint 2, Den "x")));
     ("procedure",
      Proc (["n"],
       ((([("z", Newloc (Den "n")); ("w", Newloc (Eint 1))], []),
         [Assign (Den "w", Sum (Val (Den "z"), Val (Den "w")));
          Assign (Den "r", Appl (Den "mult2", [Val (Den "w")]))])))]),
   [Call (Den "procedure", [Eint 4])])
val dr17 : (string * exp) list = [("r", Newloc (Eint 0))]
val rho17 : dval Funenv.env = <abstr>
val sigma17 : mval Funstore.store = <abstr>
val result17 : mval Funstore.store = <abstr>
val result17R : eval = Int 10
```

Test on *Block*

```
(* 16 - TEST Block *)
(*
    z=4
    w=1
    while(z!=0){
        w=w*z
        z=z-1
    }
*)
let d16 = ([("z",Newloc(Eint 4));("w",Newloc(Eint 1))],[]);;
let es16 =  [While(Not(Eq(Val(Den "z"), Eint 0)),
            [Assign(Den "w", Prod(Val(Den "w"),Val(Den "z")));
             Assign(Den "z", Diff(Val(Den "z"), Eint 1))]);
             Assign(Den "y", Val(Den "w"))
            ];;
let (es16: block) = (d16,es16);;
let dl16 = [("y",Newloc(Eint 0))];;
let (rho16, sigma16) = semdv dl16 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
let result16 = semb es16 rho16 sigma16;;
let result16Y = sem (Val(Den "y")) rho16 result16;;
```

```
val d16 : (string * exp) list * 'a list =
  ([("z", Newloc (Eint 4)); ("w", Newloc (Eint 1))], [])
val es16 : com list =
  [While (Not (Eq (Val (Den "z"), Eint 0)),
    [Assign (Den "w", Prod (Val (Den "w"), Val (Den "z")));
     Assign (Den "z", Diff (Val (Den "z"), Eint 1))]);
   Assign (Den "y", Val (Den "w"))]
val es16 : block =
  (([("z", Newloc (Eint 4)); ("w", Newloc (Eint 1))], []),
   [While (Not (Eq (Val (Den "z"), Eint 0)),
     [Assign (Den "w", Prod (Val (Den "w"), Val (Den "z")));
      Assign (Den "z", Diff (Val (Den "z"), Eint 1))]);
    Assign (Den "y", Val (Den "w"))])
val dl16 : (string * exp) list = [("y", Newloc (Eint 0))]
val rho16 : dval Funenv.env = <abstr>
val sigma16 : mval Funstore.store = <abstr>
val result16 : mval Funstore.store = <abstr>
val result16Y : eval = Int 24
```

Test on *Recursive Function*

```
(* 15 - TEST Let, Fun, Rec *)
(*
    fact(x)
        if(x==0)
            then 1
        else
            x*fact(x-1)
    result=fact(4)
*)
let es15 = Let("fact",
            Rec("fact",
                Fun(
                    ["x"],
                    Ifthenelse(
                        Eq(Den "x", Eint 0),
                        Eint 1,
                        Prod(Den "x", Appl(Den "fact", [Diff(Den "x", Eint 1)]))
                    )
                )
            ),
            Appl(Den "fact",[Eint 5])
          )
let rho15 = Funenv.emptyenv(Unbound);;
let sigma15 = Funstore.emptystore(Undefined);;
let result15 = sem es15 rho15 sigma15;;
```

```
val es15 : exp =
  Let ("fact",
   Rec ("fact",
    Fun (["x"],
     Ifthenelse (Eq (Den "x", Eint 0), Eint 1,
      Prod (Den "x", Appl (Den "fact", [Diff (Den "x", Eint 1)])))))),
   Appl (Den "fact", [Eint 5]))
val rho15 : dval Funenv.env = <abstr>
val sigma15 : mval Funstore.store = <abstr>
val result15 : eval = Int 120
```

Tests on *Reflect*

Test on Reflect with *if*

```
(* 18 - TEST reflect with "if" *)
(*
    z=1
    z=2
    if(false)
        then z=5
        else z=10
*)
let d18 = [("z",Newloc(Eint 1))];;
let es18 = Reflect("Assign(Den(\"z\"),Eint(2));Cifthenelse(Ebool(false),
                [Assign(Den(\"z\"),Eint(5))],[Assign(Den(\"z\"),Eint(10))])");;
let cl18 = reflect("Assign(Den(\"z\"),Eint(2));Cifthenelse(Ebool(false),
                [Assign(Den(\"z\"),Eint(5))],[Assign(Den(\"z\"),Eint(10))])");;
let (rho18, sigma18) = semdv d18 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
let result18 = semc es18 rho18 sigma18
let result18Z = sem (Val(Den "z")) rho18 result18;;
```

```
val d18 : (string * exp) list = [("z", Newloc (Eint 1))]
val es18 : com =
  Reflect
   "Assign(Den(\"z\"),Eint(2));Cifthenelse(Ebool(false),
   [Assign(Den(\"z\"),Eint(5))],
   [Assign(Den(\"z\"),Eint(10))])"
val cl18 : com list =
  [Assign (Den "z", Eint 2);
   Cifthenelse (Ebool false, [Assign (Den "z", Eint 5)],
    [Assign (Den "z", Eint 10)])]
val rho18 : dval Funenv.env = <abstr>
val sigma18 : mval Funstore.store = <abstr>
val result18 : mval Funstore.store = <abstr>
val result18Z : eval = Int 10
```

Test *Reflect*, *while* and expressions

```
(* 19 - TEST reflect, while, expressions *)
(*
    z=4
    w=1
    while(!(z==0)){
        w=w*z
        z=z-1
    }
*)
let d19 = [("z",Newloc(Eint 4));("w",Newloc(Eint 1))];;
let es19 = Reflect("While(Not(Eq(Val(Den(\"z\")),Eint(0))),[Assign(Den(\"w\"),
    Prod(Val(Den(\"w\")),Val(Den(\"z\"))));Assign(Den(\"z\"),Diff(Val(Den(\"z\")),Eint(1)))])");;
(* creo com list per debug*)
let cl19 = reflect("While(Not(Eq(Val(Den(\"z\")),Eint(0))),[Assign(Den(\"w\"),
    Prod(Val(Den(\"w\")),Val(Den(\"z\"))));Assign(Den(\"z\"),Diff(Val(Den(\"z\")),Eint(1)))])")
let (rho19, sigma19) = semdv d19 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
let sigma19final = semc es19 rho19 sigma19;;
let result19Z = sem (Val(Den "z")) rho19 sigma19final;;
let result19W = sem (Val(Den "w")) rho19 sigma19final;;
```

```
val d19 : (string * exp) list =
  [("z", Newloc (Eint 4)); ("w", Newloc (Eint 1))]
val es19 : com =
  Reflect
   "While(Not(Eq(Val(Den(\"z\")),Eint(0))),
   [Assign(Den(\"w\"),Prod(Val(Den(\"w\")),Val(Den(\"z\"))));
    Assign(Den(\"z\"),Diff(Val(Den(\"z\")),Eint(1)))])"
val cl19 : com list =
  [While (Not (Eq (Val (Den "z"), Eint 0)),
    [Assign (Den "w", Prod (Val (Den "w"), Val (Den "z")));
     Assign (Den "z", Diff (Val (Den "z"), Eint 1))])]
val rho19 : dval Funenv.env = <abstr>
val sigma19 : mval Funstore.store = <abstr>
val sigma19final : mval Funstore.store = <abstr>
val result19Z : eval = Int 0
val result19W : eval = Int 24
```