

Sottoprogrammi in linguaggi imperativi

1

- introduciamo le procedure nel linguaggio imperativo
 - astrazione
 - chiamata
 - regole di scoping
- semantica delle procedure con scoping statico
 - operativa
 - dominio delle procedure, `makeproc` e `applyproc`
 - iterativa
- (digressione su meccanismi alternativi) scoping dinamico
 - dominio delle procedure in semantica operativa

2

Caratteristiche del linguaggio imperativo

- include totalmente il linguaggio funzionale
 - inclusi i costrutti **Fun**, **Apply** e **Rec**
- le espressioni includono un nuovo costrutto **PROC** da usare soltanto nelle dichiarazioni di sottoprogrammi nei blocchi
 - il costrutto permette di specificare sottoprogrammi che hanno come corpo un comando
 - la loro invocazione non provoca la restituzione di un valore ma la modifica dello store
- i comandi includono un nuovo costrutto **Call** per la chiamata di sottoprogrammi
- i blocchi (oltre alla lista di comandi) hanno
 - dichiarazioni di costanti e variabili (già viste)
 - dichiarazioni di funzioni e procedure (usare **rec** per la ricorsione)

3

Espressioni

```
type ide = string
type exp = Eint of int
         | Ebool of bool
         | Den of ide
         | Prod of exp * exp
         | Sum of exp * exp
         | Diff of exp * exp
         | Eq of exp * exp
         | Minus of exp
         | Iszero of exp
         | Or of exp * exp
         | And of exp * exp
         | Not of exp
         | Ifthenelse of exp * exp * exp
         | Let of ide * exp * exp
         | Newloc of exp
         | Fun of ide list * exp
         | Appl of exp * exp list
         | Rec of ide * exp
         | Proc of ide list * decl * com list
```

4

Dichiarazioni e comandi

```
and decl = (ide * exp) list * (ide * exp) list
```

```
and com =
```

```
  | Assign of exp * exp  
  | Cifthenelse of exp * com list * com  
                                list  
  | While of exp * com list  
  | Block of decl * com list  
  | Call of exp * exp list
```

5

Commenti

```
type exp = ...  
  | Proc of ide list * decl * com list  
and com = ...  
  | Call of exp * exp list
```

- come nel caso delle funzioni, le procedure hanno
 - una lista di parametri
 - identificatori nel costrutto di astrazione procedurale (formali)
 - espressioni nel costrutto di chiamata (attuali)
- come nel caso delle funzioni, assumiamo la modalità standard di passaggio dei parametri (ovvero il passaggio per valore)
 - le espressioni parametro attuale sono valutate (**dval**) ed i valori ottenuti sono legati nell'ambiente al corrispondente parametro formale
- con l'introduzione delle procedure, il linguaggio imperativo è completo
 - lo ritoccheremo solo per discutere alcune modalità di passaggio dei parametri
- un linguaggio imperativo reale ha in più i tipi, le eccezioni ed eventuali meccanismi come i puntatori che vedremo nella estensione orientata ad oggetti

6

Semantica delle procedure

```
type exp = ...
  | Proc of ide list * decl * com list
and com = ...
  | Call of exp * exp list
```

- decidiamo che, a differenza delle funzioni, i valori `proc` con cui interpretiamo le procedure siano soltanto denotabili
- le procedure possono
 - essere dichiarate
 - essere passate come parametri
 - essere utilizzate nel comando `Call`
- le procedure non possono essere restituite come valore di una espressione

7

```
type eval =
  | Int of int | Bool of bool | Novalue
  | Funval of efun
and dval =
  | Dint of int | Dbool of bool | Unbound
  | Dloc of loc
  | Dfunval of efun
  | Dprocval of proc
and mval =
  | Mint of int
  | Mbool of bool
  | Undefined

and efun = (dval list) * (mval store) -> eval

and proc = (dval list) * (mval store) -> mval store
```

8

Soliti meccanismi di conversione

```
exception Nonstorable
exception Nonexpressible
let evaltomval e = match e with
  | Int n -> Mint n
  | Bool n -> Mbool n
  | _ -> raise Nonstorable
let mvaltoeval m = match m with
  | Mint n -> Int n
  | Mbool n -> Bool n
  | _ -> Novalue
let evaltodval e = match e with
  | Int n -> Dint n
  | Bool n -> Dbool n
  | Novalue -> Unbound
  | Funval n -> Dfunval n
let dvaltoeval e = match e with
  | Dint n -> Int n
  | Dbool n -> Bool n
  | Dloc n -> raise Nonexpressible
  | Dfunval n -> Funval n
  | Dprocval n -> raise Nonexpressible
  | Unbound -> Novalue
```

9

```
type efun = expr * dval env
type proc = expr * dval env
let rec makefun ((a:exp),(x:dval env)) = match a with
  | Fun(ii,aa) -> Dfunval(a,x)
  | _ -> failwith ("Non-functional object")
and makefunrec (i, Fun(ii, aa), r) =
  let functional (rr: dval env) = bind(r, i, makefun(e1,rr)) in
  let rec rfix = function x -> functional rfix x in
  dvaltoeval(makefun(e1, rfix))
and makeproc((a:exp),(x:dval env)) = match a with
  | Proc(ii,b) -> Dprocval(a, x)
  | _ -> failwith ("Non-functional object")
and applyfun ((ev1:dval),(ev2:dval list), s) = match ev1 with
  | Dfunval(Fun(ii,aa), x) -> sem(aa, bindlist(x, ii, ev2), s)
  | _ -> failwith ("attempt to apply a non-functional object")
and applyproc ((ev1:dval),(ev2:dval list), s) = match ev1 with
  | Dprocval(Proc(ii,b), x) -> semb(b, bindlist(x, ii, ev2), s)
  | _ -> failwith ("attempt to apply a non-functional object")
```

10

makefunrec: Esposizione rappr. ENV

```
and makefunrec (i, e1, r) =  
  let rec rfix = bind(r, i, makefun(e1,rfix)) in  
    dvaltoeval(makefun(e1, rfix))
```

← Attenzione

Soluzione:

Introduciamo nell'ambiente un operatore ausiliario bindfun:

```
bindfun(i,e,r,mk) = let rec rfix = bind(r,i,mk(e,rfix)) in rfix
```

ridefiniamo (fuori dall'ambiente) makefunrec:

```
makefunrec(i,e,r) = dvaltoeval(makefun(e,bindfun(i,e,r,makefun)))  
  o volendo distinguere funzioni da procedure avremmo potuto  
  usare il costruttore makeproc invece di makefun
```

Per liste di procedure e funzioni mutuamente ricorsive, occorre anche:

- un analogo di bind per coppie di liste
let rec bindlist(r,is,es) = match (is,es) with
 |([],[]) -> r
 |(i::isl, e::es1) -> bindlist(bind(r,i,e),isl,es1)
 |_ -> raise
- un analogo di bindfunlist per fix su più coppie:
bindfunlist(is,es,r,mk) =
 let rec rfix = bindlist(r,is,map(function x -> mk(x,rfix))es)
 in rfix

11

Semantica operativa espressioni 1

```
let rec sem ((e:exp), (r:dval env), (s: mval store)) = match e with  
| Eint(n) -> Int(n)  
| Ebool(b) -> Bool(b)  
| Den(i) -> dvaltoeval(applyenv(r,i))  
| Iszero(a) -> iszero(sem(a, r, s))  
| Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))  
| Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))  
| Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))  
| Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))  
| Minus(a) -> minus(sem(a, r, s))  
| And(a,b) -> et (sem(a, r, s), sem(b, r, s))  
| Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))  
| Not(a) -> non(sem(a, r, s))  
| Ifthenelse(a,b,c) ->  
  let g = sem(a, r, s) in  
  if typecheck("bool",g) then  
    (if g = Bool(true)  
     then sem(b, r, s)  
     else sem(c, r, s))  
  else failwith ("nonboolean guard")  
| Let(i,e1,e2) -> let (v, s1) = semden(e1, r, s) in sem(e2, bind (r ,i, v), s1)  
| Fun(i,a) -> dvaltoeval(makefun(Fun(i,a), r))  
| Appl(a, b) -> let (v1, s1) = semlist(b,r,s) in  
  applyfun(evaltodval(sem(a,r,s)), v1, s1)  
| Rec(i, e) -> makefunrec(i, e, r)  
| Val(e) -> let (v, s1) = semden(e, r, s) in (match v with  
| Dloc n -> mvaltoeval(applystore(s1, n))  
| _ -> failwith("not a variable"))  
| _ -> failwith("nonlegal expression for sem")
```

12

Semantica operativa espressioni 2

```
and semden ((e:exp), (r:dval env), (s: mval store)) = match e with
| Den(i) -> (applyenv(r,i), s)
| Fun(i,e1) -> (makefun(e,r), s)
| Proc(il,b) -> (makeproc(e,r), s)
| Newloc(e) -> let m = evaltomval(sem(e, r, s)) in
  let (l, s1) = allocate(s, m) in (Dloc l, s1)
| _ -> (evaltodval(sem(e, r, s)), s)
and semlist(el, r, s) = match el with
| [] -> ([], s)
| e::e1 -> let (v1, s1) = semden(e, r, s) in
  let (v2, s2) = semlist(e1, r, s1) in (v1 :: v2, s2)

val sem : exp * dval env * mval store -> eval = <fun>
val semden : exp * dval env * mval store ->
  dval * mval store = <fun>
val semlist : exp list * dval env * mval store ->
  dval list * mval store = <fun>
```

13

Semantica operativa comandi

```
let rec semc((c: com), (r:dval env), (s: mval store)) = match c with

| Assign(e1, e2) -> let (v1, s1) = semden(e1, r, s) in
  (match v1 with
  | Dloc(n) -> update(s1, n, evaltomval(sem(e2, r, s1)))
  | _ -> failwith ("wrong location in assignment"))
| Cifthenelse(e, c11, c12) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semc(c11, r, s) else semc(c12, r, s))
  else failwith ("nonboolean guard")
| While(e, c1) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semc((c1 @ [While(e, c1)]), r, s)
    else s)
  else failwith ("nonboolean guard")
| Block(b) -> semb(b, r, s)
| Call(e1, e2) -> let (p, s1) = semden(e1, r, s) in
  let (v, s2) = semlist(e2, r, s1) in applyproc(p, v, s2)

and semc1(c1, r, s) = match c1 with
| [] -> s
| c::c11 -> semc1(c11, r, semc(c, r, s))

val semc : com * dval env * mval store -> mval store = <fun>
val semc1 : com list * dval env * mval store -> mval store = <fun>
```

14

Semantica operativa dichiarazioni

```
and semb ((dl, rdl, cl), r, s) =
  let (r1, s1) = semdl((dl, rdl), r, s) in semcl(cl, r1, s1)

and semdv(dl, r, s) = match dl with
| [] -> (r,s)
| (i,e)::dl1 -> let (v, s1) = semden(e, r, s) in
  semdv(dl1, bind(r, i, v), s1)
and semdl ((dl, r1), r, s) = let (r1, s1) = semdv(dl, r, s) in
  semdr(r1, r1, s1)
and semdr(r1, r, s) =
  let functional ((r1: dval env)) = (match r1 with
  | [] -> r
  | (i,e) :: r11 -> let (v, s2) = semden(e, r1, s) in
    let (r2, s3) = semdr(r11. bind(r, i, v), s) in r2) in
  let rec rfix = function x -> functional rfix x in (rfix, s)

val semb : (decl * com list) * dval env * mval store -> mval store = <fun>
val semdl : decl * dval env * mval store -> dval env * mval store = <fun>
val semdv : (ide * expr) list * dval env * mval store ->
  dval env * mval store = <fun>
val semdr : (ide * expr) list * dval env * mval store ->
  dval env * mval store = <fun>
```

15

Mutua ricorsione (implicita)

```
let(mdiccom: block) =
  [{"y", Newloc (Eint 0)},
  [{"impfact", Proc(["x",
    [{"z", Newloc(Den "x")} ; {"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))]);
    Cifthenelse
      (Eq (Val (Den "w"), Appl (Den "fact", [Den "x"])),
       [Assign (Den "y", Val (Den "w"))],
       [Assign (Den "y", Eint 0)]) ] )];
  ("fact", Fun(["x",
    Ifthenelse (Eq (Den "x", Eint 0), Eint 1,
      Prod (Den "x", Appl (Den "fact", [Diff (Den "x", Eint 1)])) ) ]),
  [ Call(Den "impfact", [Eint 4]) ] ] ) ;;

# let itestore1 = semb(mdiccom, (emptyenv Unbound), (emptystore Undefined));
# applystore(itestore1, 0);
- : mval = Mint 24
```

16

Interprete iterativo

- non servono strutture dati diverse da quelle già introdotte per gestire i blocchi
 - la chiamata di procedura crea un nuovo frame invece di fare una chiamata ricorsiva a **semb**
- pila dei records di attivazione realizzata attraverso sei pile gestite in modo “parallelo”
 - **envstack** pila di ambienti
 - **cstack** pila di pile di costrutti sintattici etichettati
 - **tempvalstack** pila di pile di eval
 - **tempdvalstack** pila di pile di dval
 - **storestack** pila di memorie
 - **labelstack** pila di costrutti sintattici etichettati
- usiamo le due operazioni introdotte nel linguaggio funzionale per
 - inserire nella pila sintattica una lista di espressioni etichettate (argomenti da valutare nell’applicazione)
 - prelevare dalla pila dei temporanei una lista di eval (argomenti valutati nell’applicazione)

17

Le strutture dell’interprete iterativo 1

```
let cframesize(e) = 20
let tframesize(e) = 20
let tdframesize(e) = 20
let stacksize = 100
type labeledconstruct =
  | Expr1 of exp
  | Expr2 of exp
  | Exprd1 of exp
  | Exprd2 of exp
  | Com1 of com
  | Com2 of com
  | Com1 of labeledconstruct list
  | Decl of ide * exp
  | Decl2 of ide * exp
  | Recl of (ide * exp) list
  | Decl of labeledconstruct list
let (cstack: labeledconstruct stack stack) = emptystack(stacksize,emptystack(1,Expr1(Eint(0))))
let (tempvalstack: eval stack stack) = emptystack(stacksize,emptystack(1,Novalue))
let (tempdvalstack: dval stack stack) = emptystack(stacksize,emptystack(1,Unbound))
let envstack = emptystack(stacksize,(emptyenv Unbound))
let storestack = emptystack(stacksize,(emptystore Undefined))
let (labelstack: labeledconstruct stack) = emptystack(stacksize,Expr1(Eint(0)))
```

18

Le strutture dell'interprete iterativo 2

```
let labelcom (dl: com list) = let dlr = ref(dl) in let ldlr = ref({}) in
  while not (ldlr = []) do
    let i = List.hd !dlr in
    ldlr := !ldlr @ [Com1(i)]; dlr := List.tl !dlr
  done;
  Com1(!ldlr)
let labeldec (dl: (ide * exp) list) = let dlr = ref(dl) in let ldlr = ref({}) in
  while not (ldlr = []) do
    let i = List.hd !dlr in
    ldlr := !ldlr @ [Decl(i)]; dlr := List.tl !dlr
  done;
  Decl(!ldlr)
let pushenv(r) = push(r,envstack)
let topenv() = top(envstack)
let popenv () = pop(envstack)
let svuotaenv() = svuota(envstack)
let pushstore(s) = push(s,storestack)
let popstore () = pop(storestack)
let svuotastore () = svuota(storestack)
let topstore() = top(storestack)
let pushargs ((b: exp list),(continuation: labeledconstruct stack)) =
  let br = ref(b) in
  while not(!br = []) do
    push(Exprpr1(List.hd !br),continuation); br := List.tl !br
  done
let getargs ((b: exp list),(tempstack: dval stack)) =
  let br = ref(b) in let er = ref({}) in
  while not(!br = []) do
    let arg=top(tempstack) in
    pop(tempstack); er := !er @ [arg]; br := List.tl !br
  done;
  !er
```

19

makefun, applyfun, makefunrec

```
let makefun ((a:exp),(x:dval env)) =
  (match a with
  | Fun(ii,aa) -> Dfunval(a,x)
  | _ -> failwith ("Non-functional object"))
let applyfun ((ev1:dval),(ev2:dval list), s) =
  ( match ev1 with
  | Dfunval(Fun(ii,aa),r) -> newframes(Expr1(aa),bindlist(r, ii, ev2), s)
  | _ -> failwith ("attempt to apply a non-functional object"))
let makefunrec (i, e1, (r:dval env)) =
  let functional (rr: dval env) =
    bind(r, i, makefun(e1,rr)) in
  let rec rfix = function x -> functional rfix x
    in dvaltoeval(makefun(e1, rfix))
let makeproc((a:exp),(x:dval env)) = match a with
| Proc(ii,b) -> Dprocval(a, x)
| _ -> failwith ("Non-functional object")
let applyproc ((ev1:dval),(ev2:dval list), s) = match ev1 with
| Dprocval(Proc(ii,(l1, l2, l3)), x) ->
  newframes(labelcom(l3), bindlist(x, ii, ev2), s);
  push(Rdecl(l2), top(cstack));
  push(labeldec(l1),top(cstack))
| _ -> failwith ("attempt to apply a non-functional object")
```

20

L'interprete iterativo 0

- la creazione di un nuovo record di attivazione (frame): invariata!

```
let newframes(ss, rho, sigma) =
  pushenv(rho);
  pushstore(sigma);
  let cframe = emptystack(cframesize(ss), Expr1(Eint 0)) in
  let tframe = emptystack(tframesize(ss), Novalue) in
  let dframe = emptystack(tdframesize(ss), Unbound) in
  push(ss, cframe);
  push(ss, labelstack);
  push(cframe, cstack);
  push(dframe, tempdvalstack);
  push(tframe, tempvalstack)
val newframes : labeledconstruct * dval env *
  mval store -> unit = <fun>
```

21

L'interprete iterativo 1

```
let itsem() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
  | Expr1(x) ->
    (pop(continuation); push(Expr2(x), continuation);
    (match x with
    | Iszero(a) -> push(Expr1(a), continuation)
    | Eq(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Prod(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Sum(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Diff(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Minus(a) -> push(Expr1(a), continuation)
    | And(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Or(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Not(a) -> push(Expr1(a), continuation)
    | Ifthenelse(a,b,c) -> push(Expr1(a), continuation)
    | Val(a) -> push(Expr1(a), continuation)
    | Newloc(e) -> failwith ("nonlegal expression for sem")
    | Let(i,e1,e2) -> push(Expr1(e1), continuation)
    | Appl(a,b) -> push(Expr1(a), continuation);
      pushargs(b, continuation)
    | Proc(i,b) -> failwith ("nonlegal expression for sem")
    | _ -> ()))
```

22

L'interprete iterativo 2

```

Expr2(x) ->
(pop(continuation); (match x with
| Eint(n) -> push(Int(n),tempstack)
| Ebool(b) -> push(Bool(b),tempstack)
| Den(i) -> push(dvaltoeval(applyenv(rho,i)),tempstack)
| Iszero(a) -> let arg=top(tempstack) in pop(tempstack); push(iszero(arg),tempstack)
| Eq(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(equ(firstarg,sndarg),tempstack)
| Prod(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(mult(firstarg,sndarg),tempstack)
| Sum(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(plus(firstarg,sndarg),tempstack)
| Diff(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(diff(firstarg,sndarg),tempstack)
| Minus(a) -> let arg=top(tempstack) in pop(tempstack); push(minus(arg),tempstack)
| And(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(et(firstarg,sndarg),tempstack)
| Or(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
let sndarg=top(tempstack) in pop(tempstack); push(vel(firstarg,sndarg),tempstack)
| Not(a) -> let arg=top(tempstack) in pop(tempstack); push(non(arg),tempstack)
| Ifthenelse(a,b,c) -> let arg=top(tempstack) in pop(tempstack); if typecheck("bool",arg) then
(if arg = Bool(true) then push(Expr1(b),continuation) else push(Expr1(c),continuation))
else failwith("type error"))
| Val(e) -> let v = top(tempdstack) in pop(tempdstack); (match v with
| Dloc n -> push(mvaltoeval(applystore(sigma, n)), tempstack)
| _ -> failwith("not a variable"))
| Fun(i,a) -> push(dvaltoeval(makefun(Fun(i,a),rho)),tempstack)
| Rec(f,e) -> push(makefunrec(f,e,rho),tempstack)
| Let(i,e1,e2) -> let arg= top(tempdstack) in
pop(tempdstack); newframes(Expr1(e2), bind(rho, i, arg), sigma)
| Appl(a,b) -> let firstarg=evaltodval(top(tempdstack)) in
pop(tempdstack); let sndarg=getargs(b,tempdstack) in applyfun(firstarg, sndarg, sigma)
| _ -> failwith("no more cases for itsem")
-> failwith("no more cases for itsem"))
val itsem : unit -> unit = <fun>

```

L'interprete iterativo 3

```

let itsemnden() =
let continuation = top(cstack) in
let tempstack = top(tempvalstack) in
let tempdstack = top(tempdvalstack) in
let rho = topenv() in
let sigma = topstore() in
(match top(continuation) with
| Exprd1(x) -> (pop(continuation); push(Exprd2(x),continuation);
match x with
| Den i -> ()
| Fun(i, e) -> ()
| Proc(i, b) -> ()
| Newloc(e) -> push(Expr1( e), continuation)
| _ -> push(Expr2(x), continuation))
| Exprd2(x) -> (pop(continuation); match x with
| Den i -> push(applyenv(rho,i), tempdstack)
| Fun(i, e) -> push(makefun(x, rho), tempdstack)
| Proc(i, b) -> push(makeproc(x, rho), tempdstack)
| Newloc(e) -> let m=evaltomval(top(tempstack)) in pop(tempstack);
let (l, s1) = allocate(sigma, m) in push(Dloc l, tempdstack);
popstore(); pushstore(s1)
| _ -> let arg = top(tempstack) in pop(tempstack);
push(evaltodval(arg), tempdstack)
| _ -> failwith("No more cases for semden") )
val itsemnden : unit -> unit = <fun>

```

L'interprete iterativo 4

```

let itsemcl () =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let cl = (match top(continuation) with
    | Coml(d11) -> d11
    | _ -> failwith("impossible in semdecl")) in
  if cl = [] then pop(continuation) else
  (let currc = List.hd cl in let newcl = List.tl cl in pop(continuation); push(Coml(newcl),continuation);
  (match currc with
    | Coml(Assign(e1, e2)) -> pop(continuation); push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
    push(Exprd1(e1), continuation); push(Expr1(e2), continuation)
    | Com2(Assign(e1, e2)) -> let arg2 = evaltomval(top(tempstack)) in pop(tempstack);
    let arg1 = top(tempdstack) in pop(tempdstack); (match arg1 with
      | Dloc(n) -> popstore(); pushstore(update(sigma, n, arg2))
      | _ -> failwith ("wrong location in assignment"))
    | Coml(While(e, cl)) -> pop(continuation); push(Coml(Com2(While(e, cl))::newcl),continuation);
    push(Expr1(e), continuation)
    | Com2(While(e, cl)) -> let g = top(tempstack) in pop(tempstack);
    if typecheck("bool",g) then (if g = Bool(true) then (let old = newcl in let newl =
      (match labelcom cl with
        | Coml newl1 -> newl1
        | _ -> failwith("impossible in while")) in
      let nuovo = Coml(newl @ [Coml(While(e, cl))] @ old) in pop(continuation); push(nuovo,continuation))
    else ()) else failwith ("nonboolean guard")
  ))
  )

```

25

L'interprete iterativo 5

```

| Coml(Cifthenelse(e, c11, c12)) -> pop(continuation);
push(Coml(Com2(Cifthenelse(e, c11, c12))::newcl),continuation);
push(Expr1(e), continuation)
| Com2(Cifthenelse(e, c11, c12)) -> let g = top(tempstack) in pop(tempstack);
if typecheck("bool",g) then (let temp = if g = Bool(true) then
labelcom (c11) else labelcom (c12) in let newl = (match temp with
| Coml newl1 -> newl1
| _ -> failwith("impossible in cifthenelse")) in
let nuovo = Coml(newl @ newcl) in pop(continuation); push(nuovo,continuation))
else failwith ("nonboolean guard")
| Coml(Call(e, el)) -> pop(continuation);
push(Coml(Com2(Call(e, el))::newcl),continuation);
push(Exprd1(e), continuation); pushargs(el, continuation)
| Com2(Call(e, el)) ->
let p = top(tempdstack) in pop(tempdstack);

let args = getargs(el,tempdstack) in applyproc(p, args, sigma)

| Coml(Block((l1, l2,l3))) -> newframes(labelcom(l3), rho, sigma);
push(Rdecl(l2),top(cstack));
push(labeldec(l1),top(cstack))
| _ -> failwith("no more sensible cases in commands" ))
)
let itsemcl : unit -> unit = <fun>

```

26

L'interprete iterativo 6 (invariata!)

```
let itsemdecl () =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let dl = (match top(continuation) with
    | Decl(d1) -> d1
    | _ -> failwith("impossible in semdecl")) in
  if dl = [] then pop(continuation) else
  (let currd = List.hd dl in
  let newdl = List.tl dl in pop(continuation); push(Decl(newdl),continuation);
  (match currd with

  | Decl( (i,e) ) ->
    pop(continuation);
    push(Decl(Decl2(i, e)::newdl,continuation);
    push(Exprd1(e), continuation)

  | Decl2(i,e) ->
    let arg = top(tempdstack) in
    pop(tempdstack);
    popenv(); pushenv(bind(rho, i, arg))

  | _ -> failwith("no more sensible cases for semdecl")))

val itsemdecl : unit -> unit = <fun>
```

27

L'interprete iterativo 7

```
let itsemrdecl() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let rl = (match top(continuation) with
    | Rdecl(r1) -> r1
    | _ -> failwith("impossible in semrdecl")) in
  pop(continuation);
  let functional (rr: dval env) =
    let pr = ref(rho) in
    let prl = ref(rl) in
    while not(!prl = []) do
  let currd = List.hd !prl in
  prl := List.tl !prl;
  let (i, den) =
    (match currd with
    |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rr))
    |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rr))
    | _ -> failwith("no more sensible cases in recursive declaration")) in
  pr := bind(!pr, i, den)
  done;
  !pr in
  let rec rfix = function x -> functional rfix x in
  popenv();
  pushenv(rfix)

val itsemrdecl : unit -> unit = <fun>
```

28

L'interprete iterativo 8

```
let initState() = svuota(continuation); svuota(tempstack); svuota(tempdvalstack);
svuotaenv(); svuotastore(); svuota(labelstack)
val initState : unit -> unit = <fun>

let loop () =
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      let currconstr = top(top(cstack)) in
      (match currconstr with
      | Expr1(e) -> itsem()
      | Expr2(e) -> itsem()
      | Exprd1(e) -> itsenden()
      | Exprd2(e) -> itsenden()
      | Coml(cl) -> itsemcl()
      | Rdecl(l) -> itsemdecl()
      | Decl(l) -> itsemdecl()
      | _ -> failwith("non legal construct in loop"))
    done;
    (match top(labelstack) with
    | Expr1(_) -> let valore = top(top(tempvalstack)) in
      pop(top(tempvalstack)); pop(tempvalstack); push(valore, top(tempvalstack));
      popenv(); popstore(); pop(tempdvalstack)
    | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
      pop(top(tempdvalstack)); pop(tempdvalstack); push(valore, top(tempdvalstack));
      popenv(); popstore(); pop(tempvalstack)
    | Decl(_) -> pop(tempvalstack); pop(tempdvalstack)
    | Rdecl(_) -> pop(tempvalstack); pop(tempdvalstack)
    | Coml(_) -> let st = topstore() in popenv(); popstore(); popstore(); pushstore(st);
      pop(tempvalstack); pop(tempdvalstack)
    | _ -> failwith("non legal label in loop"));
    pop(cstack); pop(labelstack)
  )
val loop : unit -> unit = <fun>
```

L'interprete iterativo 9 (invariato!)

```
let sem (e,(r: dval env), (s: mval store)) = initState();
push(emptystack(tframesize(e), Novalue), tempvalstack);
newframes(Expr1(e), r, s);
loop();
let valore = top(top(tempvalstack)) in
pop(tempvalstack); valore
val sem : exp * dval env * mval store -> eval = <fun>

let senden (e,(r: dval env), (s: mval store)) = initState();
push(emptystack(tframesize(e), Unbound), tempdvalstack);
newframes(Exprd1(e), r, s);
loop();
let valore = top(top(tempdvalstack)) in
pop(tempdvalstack);
valore
val senden : exp * dval env * mval store -> dval = <fun>

let semcl (cl,(r: dval env), (s: mval store)) = initState();
pushstore(emptystore(Undefined));
newframes(labelcom(cl), r, s);
loop();
let st = topstore() in popstore();
st
val semcl : com list * dval env * mval store -> mval store = <fun>
```

L'interprete iterativo 10 (invariato!)

```
let semdv(dl, r, s) = initState();
  newframes(labeldec(dl), r, s);
  loop();
  let st = topstore() in popstore();
  let rt = topenv() in popenv();
  (rt, st)
val semdv : (ide * exp) list * dval env * mval store ->
  dval env * mval store = <fun>

let semc((c: com), (r:dval env), (s: mval store)) = initState();
  pushstore(emptystore(Undefined));
  newframes(labelcom([c]), r, s);
  loop();
  let st = topstore() in popstore();
  st
val semc : com * dval env * mval store -> mval store = <fun>
```

31

L'interprete iterativo 11

```
let semdr(dl, r, s) = initState();
  newframes(Rdecl(dl), r, s);
  loop();
  let st = topstore() in popstore();
  let rt = topenv() in popenv();
  (rt, st)
val semdr : (ide * exp) list * dval env * mval store ->
  dval env * mval store = <fun>

let semdl((dl, rl), r, s) = initState();
  newframes(Rdecl(rl), r, s);
  push(labeldec(dl), top(cstack));
  loop();
  let st = topstore() in popstore();
  let rt = topenv() in popenv();
  (rt, st)
val semdl : decl * dval env * mval store ->
  dval env * mval store = <fun>

let semb((dl, rl, cl), r, s) = initState();
  pushstore(emptystore(Undefined));
  newframes(labelcom(cl), r, s);
  push(Rdecl(rl), top(cstack));
  push(labeldec(dl), top(cstack));
  loop();
  let st = topstore() in popstore();
  st
val semb : (decl * com list) * dval env * mval store ->
  mval store = <fun>
```

32

Cosa abbiamo realizzato?

- come per il linguaggio funzionale, manca l'implementazione vera del dominio ambiente e quella del dominio store!
- nella implementazione attuale abbiamo una pila di ambienti ed una pila di memorie relativi alle varie attivazioni
 - ognuno degli ambienti è l'ambiente complessivo
 - rappresentato attraverso una funzione
 - ognuna delle memorie è la memoria complessiva
 - rappresentata attraverso una funzione
- in una implementazione reale ogni attivazione dovrebbe avere
 - l'ambiente locale (ed un modo per reperire il resto dell'ambiente visibile)
 - la memoria locale
 - l'ambiente e la memoria locali dovrebbero essere "implementati" al prim'ordine (con strutture dati)
- vedremo tali implementazioni tra un po' di tempo

33

scoping dinamico

- tutto come nel linguaggio funzionale
 - in particolare per quanto riguarda ricorsione, verifiche statiche ed ottimizzazioni
- vediamo soltanto i domini di funzioni e procedure in semantica operativa
 - quelli della semantica iterativa sono ovviamente gli stessi
 - le funzioni di creazione e applicazione (per funzioni e procedure) sono lasciate come esercizio
- l'implementazione al prim'ordine dell'ambiente in presenza di scoping dinamico verrà vista quando parleremo di implementazione dello stato

34

Funzioni e procedure con scoping dinamico

scoping statico

```
type efun = exp * (dval env)
```

```
type proc = exp * (dval env)
```

scoping dinamico

```
type efun = exp
```

```
type proc = exp
```