



AA 2015-2016

# 25. Implementazione di un linguaggio imperativo



# 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
  - dichiarazioni di funzioni e procedure (usando `rec` per la ricorsione)

# Espressioni



```
type ide = string
```

```
type 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
```

# 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`



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

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) e i valori ottenuti sono legati nell'ambiente al corrispondente parametro formale

Un linguaggio imperativo reale ha in più i tipi, le eccezioni ed eventuali meccanismi come i puntatori

# Procedure



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

- ✎ A differenza delle funzioni, i valori proc con i quali interpretiamo le procedure sono 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



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



# 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
```



```
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")
```



## 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)))`

- 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::is1, e::es1) -> bindlist(bind(r,i,e),is1,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
```

# 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) = senden(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) = senden(e, r, s) in (match v with
| Dloc n -> mvaltoeval(applystore(s1, n))
| _ -> failwith("not a variable"))
| _ -> failwith("nonlegal expression for sem")
```

# 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::el1 -> let (v1, s1) = semden(e, r, s) in
    let (v2, s2) = semlist(el1, 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>
```

# 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, cl1, cl2) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semcl(cl1, r, s) else semcl (cl2, r, s))
  else failwith ("nonboolean guard")
| While(e, cl) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semcl((cl @ [While(e, cl)]), 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 semcl(cl, r, s) = match cl with
| [] -> s
| c::cl1 -> semcl(cl1, r, semc(c, r, s))

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

# 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, rl), r, s) = let (r1, s1) = semdv(dl, r, s) in
    semdr(rl, r1, s1)
and semdr(rl, r, s) =
    let functional ((r1: dval env)) = (match rl with
    | [] -> r
    | (i,e) :: rl1 -> let (v, s2) = semden(e, r1, s) in
        let (r2, s3) = semdr(rl1. 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>
```



# 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
```