



---

## SOTTOPROGRAMMI E ASTRAZIONI FUNZIONALI IN LINGUAGGI FUNZIONALI

1



---

## Di cosa parliamo oggi?

- 👉 nascono i sottoprogrammi (ma sembra di parlare di archeologica .. informatica)
  - per rispondere a quali esigenze?
  - cosa veniva offerto per la loro simulazione
- 👉 Il punto importante: astrazione funzionale
- 👉 Data abstraction (ne parleremo nella seconda parte del corso)

2



- ✎ introduciamo le funzioni nel linguaggio funzionale
  - astrazione
  - applicazione
  - regole di scoping: quando un binding e' attivo?
- ✎ semantica delle funzioni con scoping statico e con scoping dinamico)
- ✎ scoping statico vs. scoping dinamico

3

## Breve storia dei sottoprogrammi



- ✎ astrazione di una sequenza di istruzioni
- ✎ un frammento di programma (sequenza di istruzioni) risulta utile in diversi punti del programma
  - riduco il "costo della programmazione" se posso dare un nome al frammento e qualcuno per me inserisce automaticamente il codice del frammento ogni qualvolta nel "programma principale" c'è un'occorrenza del nome
    - ✓ macro e macro-espansione

4

## Breve storia dei sottoprogrammi



- riduco anche l'occupazione di memoria se esiste un meccanismo che permette al programma principale
  - ✓ di trasferire il controllo ad una unica copia del frammento memorizzata separatamente
  - ✓ di riprendere il controllo quando l'esecuzione del frammento è terminata
  - ✓ la subroutine supportata anche dall'hardware (codice rientrante)

5

## Breve storia dei sottoprogrammi (2)



- ✎ astrazione via parametrizzazione
- ✎ il frammento diventa ancora più importante se può essere realizzato in modo parametrico
  - astraendo dall'identità di alcuni dati
  - la cosa è possibile anche con le macro ed il codice rientrante
    - ✓ macroespansione con rimpiazzamento di entità diverse
    - ✓ associazione di informazioni variabili al codice rientrante

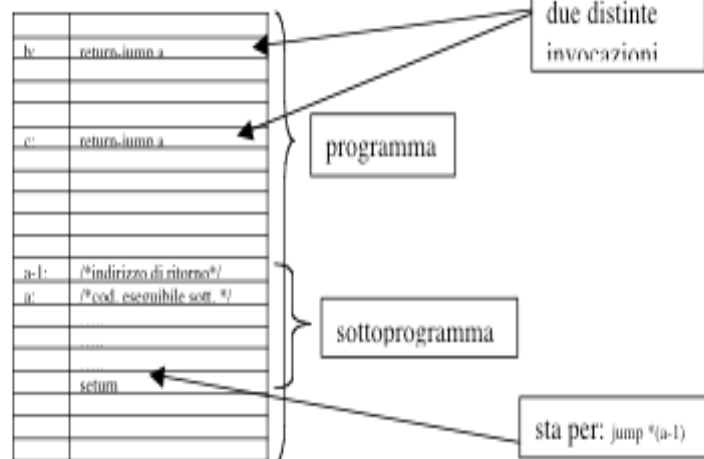
6

## Cosa fornisce l'hardware?



- una operazione primitiva di `return jump`
  - Con opportune strutture ausiliarie
- viene eseguita (nel programma chiamante) l'istruzione `return jump` memorizzata nella cella `b`
  - il controllo viene trasferito alla cella `a` (entry point della subroutine)
  - l'indirizzo dell'istruzione successiva (`b + 1`) viene memorizzato in qualche posto noto, per esempio nella cella (`a - 1`) (punto di ritorno)
- quando nella subroutine si esegue una operazione di `return`
  - il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno

7



8

## Archeologia: FORTRAN



- ✦ una subroutine è un pezzo di codice compilato, a cui sono associati
  - una cella destinata a contenere (a tempo di esecuzione) i punti di ritorno relativi alle (possibili varie) chiamate
  - alcune celle destinate a contenere i valori degli eventuali parametri
  - ambiente e memoria locali
  - ambiente locale è statico

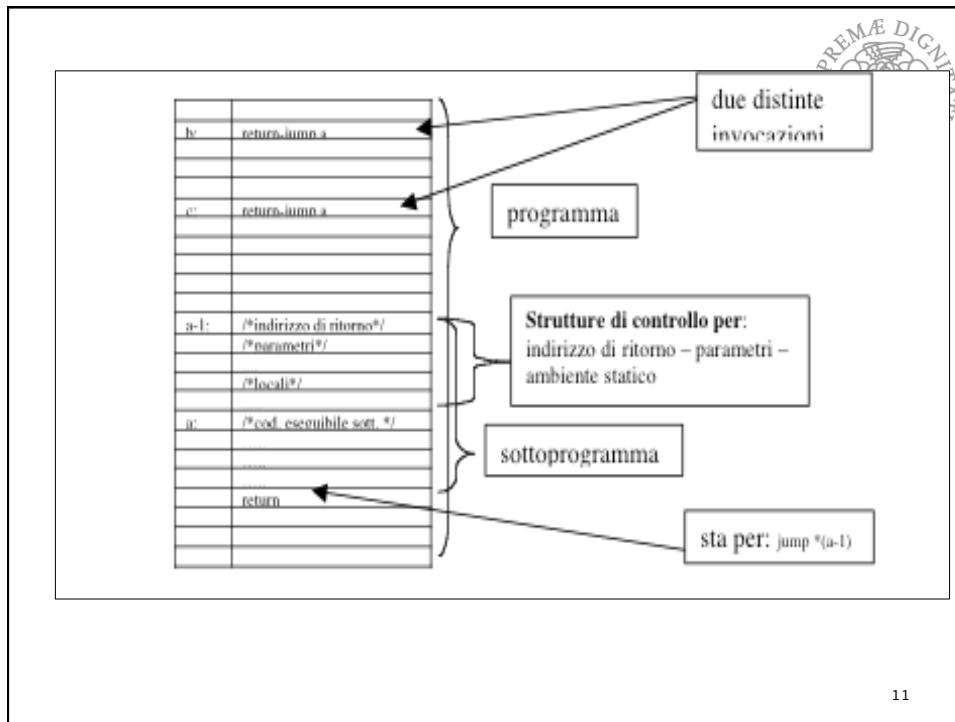
9

## Semantica della subroutine à la FORTRAN



- ✦ si può definire facilmente attraverso la *copy rule statica* (macroespansione!)
  - ogni chiamata di sottoprogramma è *testualmente rimpiazzata* da una copia del codice
    - ✓ facendo qualcosa per i parametri
    - ✓ ricordandosi che le dichiarazioni sono eseguite una sola volta
- ✦ il sottoprogramma non è semanticamente qualcosa di nuovo è solo un (importante) strumento metodologico (astrazione!)

10



## Semantica della subroutine à la FORTRAN

- ✎ Osservazione: non è compatibile con la ricorsione
  - la macroespansione darebbe origine ad un programma infinito
  - l'implementazione à la FORTRAN (con un solo punto di ritorno) non permetterebbe di gestire più attivazioni presenti allo stesso tempo
- ✎ il fatto che le subroutine FORTRAN siano concettualmente una cosa statica fa sì che
  - non esista di fatto il concetto di attivazione
  - l'ambiente locale sia necessariamente statico

## Attivazione



- ✚ se ragioniamo in termini di attivazioni come già abbiamo fatto con i blocchi la semantica può essere ancora definita da una *copy rule*, ma *dinamica*
  - ogni chiamata di sottoprogramma è *rimpiazzata a tempo di esecuzione* da una copia del codice
- ✚ il sottoprogramma è ora semanticamente qualcosa di nuovo
- ✚ ragionare in termini di attivazioni
  - rende naturale la ricorsione
  - porta ad adottare la regola dell'ambiente locale dinamico

13

## Le strutture di implementazione



- ✚ invece delle informazioni staticamente associate al codice compilato di FORTRAN
  - punto di ritorno, parametri, ambiente e memoria locale
- ✚ *record di attivazione*
  - contenente le stesse informazioniassociato dinamicamente alle varie chiamate di sottoprogrammi
- ✚ dato che i sottoprogrammi hanno un comportamento LIFO
  - l'ultima attivazione creata nel tempo è la prima che ritornaci possiamo aspettare che i record di attivazione siano organizzati in una pila

14



- ☛ abbiamo già incontrato questa struttura di implementazione nell'interprete iterativo dei frammenti con blocchi
  - i blocchi sono un caso particolare di sottoprogrammi

15

## Cosa è un sottoprogramma vero



- ☛ astrazione procedurale (operazioni)
  - astrazione di una sequenza di istruzioni
  - astrazione via parametrizzazione
- ☛ luogo di controllo per la gestione dell'ambiente e della memoria
  - estensione del blocco
  - in assoluto, l'aspetto più interessante dei linguaggi, intorno a cui ruotano tutte le decisioni semantiche importanti
  - Binding: statico o dinamico

16



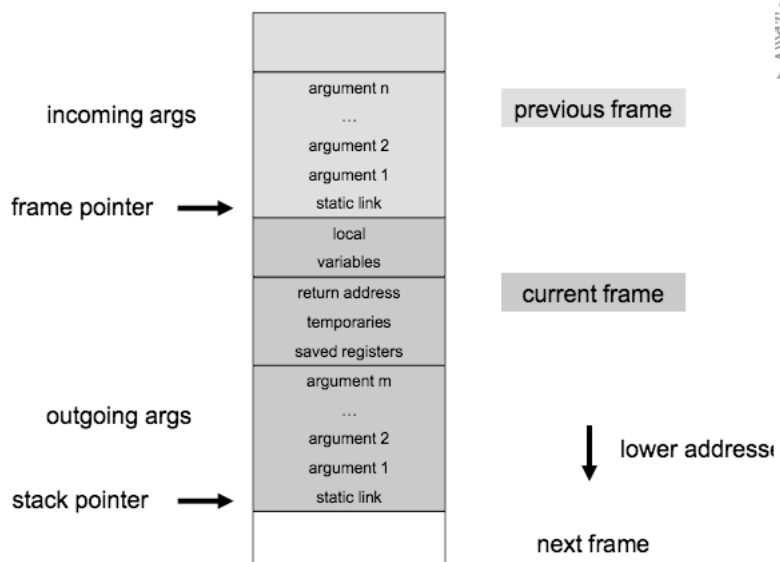
# Andiamo per la tangente



🐞 Come si progetta la struttura del record di attivazione?

- Dipende dal linguaggio di programmazione
- Dipende dalla struttura della macchina ospite

17



18

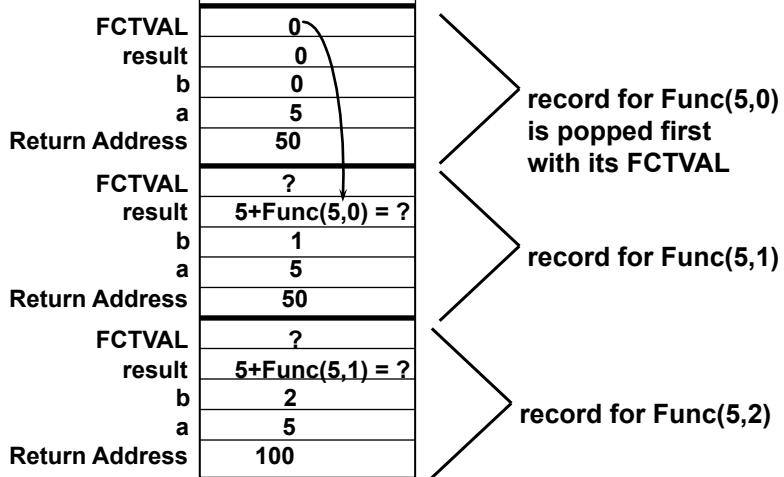






## Run-Time Stack Activation Records

`x = Func(5, 2); // original call at instruction 100`

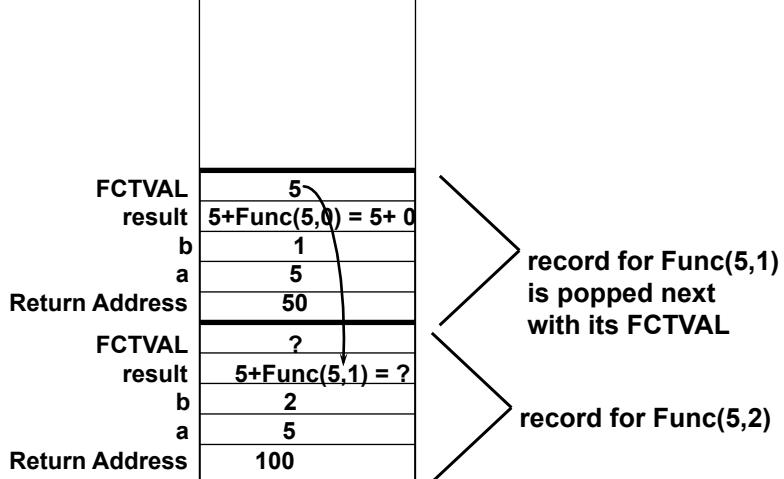


23



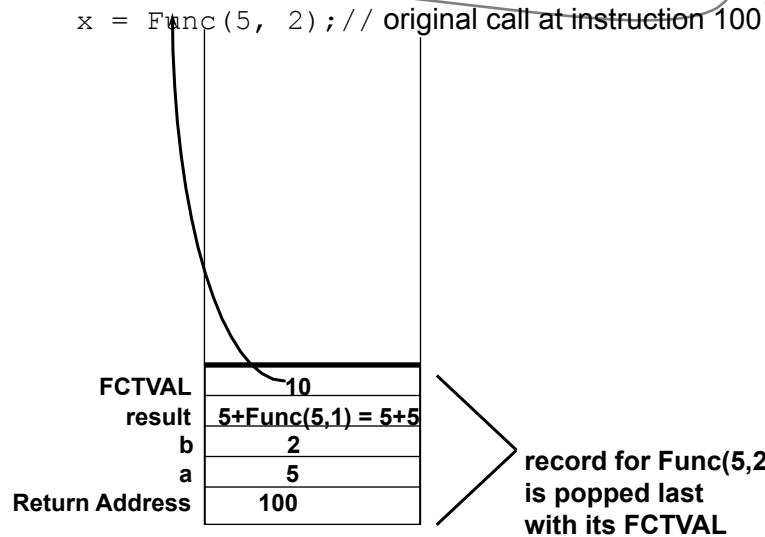
## Run-Time Stack Activation Records

`x = Func(5, 2); // original call at instruction 100`



24

## Run-Time Stack Activation Records



25

## Activation Record

- ✎ La struttura del record di attivazione puo' essere determinata staticamente (la funzione *newframe*) ma la dimensione effettiva dipende da valori a run time
- ✎ Informazioni di controllo: *dynamic link*
  - punta all'inizio del record di attivazione del chiamante (la pila dei costrutti etichettati)
- ✎ I record di attivazione vengono allocati sullo stack
- ✎ Il run time prevede un *Environment Pointer (EP)*
  - punta alla base del record di attivazione corrente.

# C : Function



```
void sub(float total, int part)
{
    int list[5];
    float sum;
    ...
}
```

Local	num
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

# Call/Return



## Chiamante

- Crea una istanza del record di attivazione
- Salva lo stato dell'unità corrente di esecuzione (il chiamante)
- Effettua il passaggio dei parametri
  - ✓ (noi lo vediamo dopo!!)
- Inserisce il punto di ritorno
  - ✓ Nel nostro caso la pila dei costrutti etichettati
- Trasferisce il controllo al chiamato

## Chiamato (prologo):

- Salva il valore corrente di EP e lo memorizza nel link dinamico.
- Definisce il nuovo valore di EP
- Alloca le variabili locali

## Call/Return



### Chiamato (epilogo)

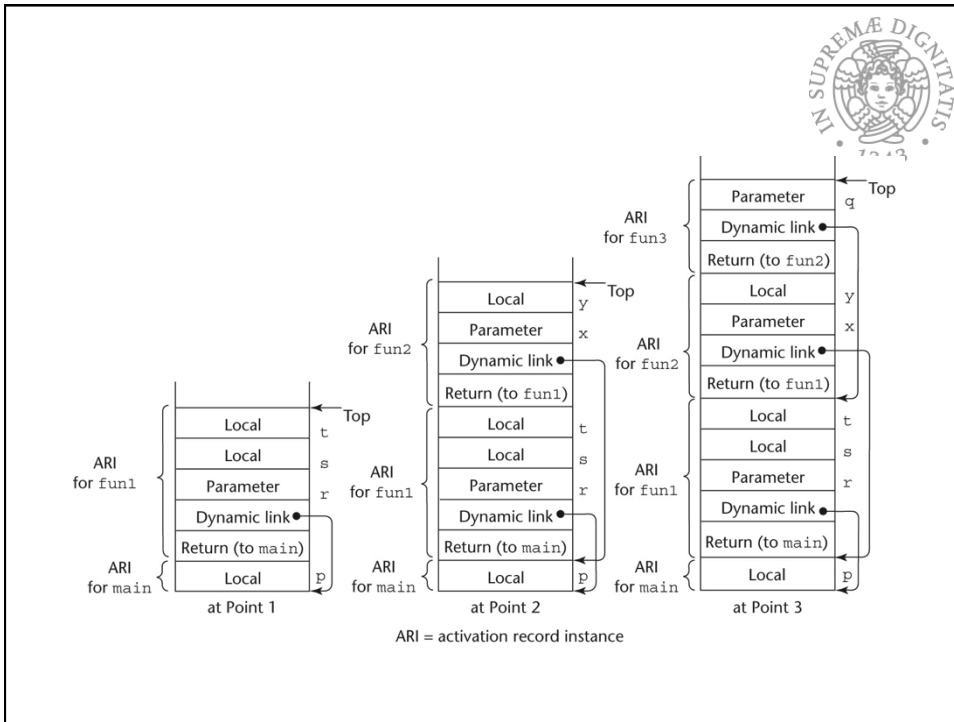
- Passaggio di valori eventuali a casusa della modalita' di passaggio dei paramenti (lo vediamo dopo)
- Il valore calcolato dalla funzione viene trasferito al chiamante
- Ripristina le informazioni di controllo (il vecchio valore di EP salvato come link dinamico)
- Ripristina lo stato di esecuzione del chiamante
- Trasferisce il controllo al chiamante

## Esempio



```
void fun1(float r) {  
    int s, t;  
    ...  
    fun2(s);  
    ...  
}  
void fun2(int x) {  
    int y;  
    ...  
    fun3(y);  
    ...  
}  
void fun3(int q) {  
    ...  
}  
void main() {  
    float p;  
    ...  
    fun1(p);  
    ...  
}
```

```
main calls fun1  
fun1 calls fun2  
fun2 calls fun3
```



## Introduciamo le funzioni nel linguaggio funzionale



```
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
         | Fun of ide list * exp
         | Appl of exp * exp list
         | Rec of ide * exp
```





```
type exp = ...
  | Fun of ide list * exp
  | Appl of exp * exp list
  | Rec of ide * exp
```

- le funzioni hanno (oltre ai parametri)
  - ✓ identificatori nel costrutto di astrazione
  - ✓ espressioni nel costrutto di applicazione
- Per ora non ci occupiamo di modalità di passaggio dei parametri
  - o le espressioni parametro attuale sono valutate (`eval` oppure `dval`) ed i valori ottenuti sono legati nell'ambiente al corrispondente parametro formale
- in un primo momento ignoriamo il costrutto `Rec`
- con l'introduzione delle funzioni, il linguaggio funzionale è completo
  - o lo ritorcheremo solo per discutere alcune modalità di passaggio dei parametri
- un linguaggio funzionale reale (tipo ML) ha in più i tipi, il pattern matching e le eccezioni

33

## Giochiamo con la semantica (1)



```
type eval = | Int of int | Bool of bool | Unbound
            | Funval of efun
and efun = exp * eval env
```

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | ...
  | Fun(ii, aa) ->
      Funval(e, r)
  | Apply(e1, e2) -> match sem(e1, r) with
      | Funval(Fun(ii, aa), r1) ->
          sem(aa, bindlist(r1, ii, semlist(e2, r)))
```

- la definizione del dominio `efun` e la corrispondente semantica della applicazione mostrano che
- il corpo della funzione viene valutato nell'ambiente ottenuto
  - ✓ legando i parametri formali ai valori dei parametri attuali
  - ✓ nell'ambiente `r1` che è quello in cui era stata valutata l'astrazione

34

## Giochiamo con la semantica (2)



```
type eval = | Int of int | Bool of bool | Unbound
           | Funval of efun
and efun = expr
```

```
let rec sem (e:expr) (r:eval env) =
  match e with
  | ...
  | Fun(ii, aa) ->
      Funval(e)
  | Apply(e1, e2) -> match sem(e1, r) with
      | Funval(Fun(ii, aa)) ->
          sem(aa, bindlist(r, ii, semlist(e2, r)))
```

- la definizione del dominio efun e la corrispondente semantica dell'applicazione mostrano che
- il corpo della funzione viene valutato nell'ambiente ottenuto
  - ✓ legando i parametri formali ai valori dei parametri attuali
  - ✓ nell'ambiente  $r$  che è quello in cui avviene la applicazione

35

## Le regole di scoping



```
type efun = expr * eval env
| Apply(e1, e2) -> match sem(e1, r) with
  | Funval(Fun(ii, aa), r1) ->
      sem(aa, bindlist(r1, ii, semlist(e2, r)))
```

- scoping statico (lessicale): l'ambiente non locale della funzione è quello esistente al momento in cui viene valutata l'astrazione

```
type efun = expr
| Apply(e1, e2) -> match sem(e1, r) with
  | Funval(Fun(ii, aa)) ->
      sem(aa, bindlist(r, ii, semlist(e2, r)))
```

- scoping dinamico: l'ambiente non locale della funzione è quello esistente al momento in cui avviene l'applicazione

36

# Valutazione



- ✎ scoping statico in cui l'ambiente non locale della funzione è quello esistente al momento in cui viene valutata l'astrazione
  - è migliore di quello dinamico
    - affidabilità, possibilità di effettuare analisi statiche
      - ✓ errori rilevati "a tempo di compilazione"
    - ottimizzazioni possibili nell'implementazione
- ✎ nel linguaggio didattico, adottiamo lo scoping statico
  - discuteremo lo scoping dinamico successivamente
- ✎ il confronto critico fra i due meccanismi verrà effettuato verso la fine del corso con tutti gli elementi disponibili

37

# La semantica operativa



```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> applyenv(r,i)
  | Iszero(a) -> iszero(sem(a, r))
  | Eq(a,b) -> equ(sem(a, r),sem(b, r))
  | :
  | Ifthenelse(a,b,c) -> let g = sem(a, r) in
    if typecheck("bool",g) then
      (if g = Bool(true) then sem(b, r) else sem(c, r))
    else failwith ("nonboolean guard")
  | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
  | Fun(i,a) -> makefun(Fun(i,a), r)
  | Appl(a,b) -> applyfun(sem(a, r), semlist(b, r))
  | Rec(i,e) -> makefunrec(i, e, r)
and semlist (el, r) = match el with
| [] -> []
| e::el1 -> sem(e, r)::semlist(el1, r)
val sem : exp * eval env -> eval = <fun>
val semlist: exp list * eval env -> eval list
```

38



```
type efun = exp * eval env

and makefun ((a:exp),(x:eval env)) =
  (match a with
   | Fun(ii,aa) -> Funval(a,x)
   | _ -> failwith ("Non-functional
                    object"))
and applyfun((ev1:eval),(ev2:eval list)) =
  ( match ev1 with
   | Funval(Fun(ii,aa),r) ->
     sem(aa, bindlist( r, ii, ev2))
   | _ -> failwith ("attempt to apply
                    a non-functional
                    object"))
```

39



```
rec semc(While(e, cl),
  (r:dval env), (s: mval store)) =
  let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then
      semc((cl @ [While(e, cl)]), r, s)
    else s)
  else failwith ("nonboolean guard")
```

inizione che esprime il comportamento del while in termini di se stesso

Equazione ricorsiva

40

## Punti fissi



➤ Consideriamo la funzione  $f: \mathbb{N} \rightarrow \mathbb{N}$  definita nel modo seguente

➤  $f(x) = 0$  if  $x = 0$

➤  $f(x) = f(x-1) + 2x - 1$  altrimenti

➤ Una equazione che definisce le proprietà che la funzione deve soddisfare

41

## Punti fissi



➤  $f(x) = 0$  if  $x = 0$

➤  $f(x) = f(x-1) + 2x - 1$  altrimenti

➤ L'unica soluzione di questa equazione è la funzione  $g(x) = x * x$

42

## Soluzione di equazioni di punto fisso



- ✎ La soluzione dell'equazione si può ottenere mediante approssimazioni successive.
- ✎ Ogni approssimazione si avvicina alla soluzione dell'equazione
- ✎ Nel nostro esempio partiamo a trovare la soluzione dell'equazione dalla funzione  $f_0$  non definita (in termini di insiemi di coppie la  $f_0$  è la funzione che non contiene coppie)

43

✎  $f_0 = \_$

✎  $f_1 = 0$  se  $x = 0$  else  $f_0(x-1) + 2x - 1 = \{(0,0)\}$

✎  $f_2 = 0$  se  $x = 0$  else  $f_1(x-1) + 2x - 1 = \{(0,0), (1,1)\}$

✎  $f_3 = 0$  se  $x = 0$  else  $f_2(x-1) + 2x - 1 = \{(0,0), (1,1), (2,4)\}$

✎ Limite di questa sequenza è la funzione  $g(x) = x*x$



44

## Cosa abbiamo fatto?



- Il processo di approssimazione utilizza strumenti funzionali di ordine superiore, ovvero abbiamo usato un funzionale  $F$  ( $F: (N \rightarrow N) \rightarrow (N \rightarrow N)$ ) che prende l'approssimazione  $f_i$  e restituisce la approssimazione  $f_{i+1}$
- La soluzione e' pertanto una funzione  $f$  tale che  $F(f) = f$

45

## Fixpoint iteration



- $f = \text{fix}(F)$   
=  $f_0, f_1, f_2, f_3, \dots$   
=  $\lambda x. F(f_0), F(f_1), F(f_2), \dots$   
=  $\bigcup_i F^i(-)$

46

## Matematica



- ✎ In matematica diverse costruzioni definiscono le condizioni per l'esistenza dei punti fissi.
- ✎ Teorema di Tarski: Una funzione monotona crescente su un reticolo completo ha un reticolo completo di punti fissi.
- ✎ Teorema di Banach su spazi metrici
- ✎ .....

47

## Cosa ci serve



- ✎ Metodo per costruire la soluzione di una equazione di punto fisso nella definizione della semantica dei linguaggi di programmazione

48



## E le funzioni ricorsive?



come è fatta una definizione di funzione ricorsiva?

- o espressione Let in cui
  - ✓  $i$  è il nome della funzione (ricorsiva)
  - ✓  $e1$  è una astrazione nel cui corpo ( $aa$ ) c'è una applicazione di Den  $i$

```
Let("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 4]
  )
)
```

49

Guardiamo la semantica

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Let(i,e1,e2) ->
      sem (e2, bind (r ,i, sem(e1, r))
  | Fun(ii, aa) -> Funval(Fun(ii,aa), r)
  | Appl(a,b) ->
      match sem(a, r) with
      Funval(Fun(ii,aa), r1) ->
        sem(aa,bindlist(r1, ii, semlist(b r)))
```



il corpo "aa" (che include Den "fact" ) è valutato in un ambiente che è quello (r1) in cui si valutano sia l'espressione Let che l'espressione Fun esteso con una associazione per i parametri formali "ii" tale ambiente non contiene l'associazione tra il nome "Fact" e la funzione la semantica di Den "fact" restituisce Unbound

50



**MORALE:**

per permettere la ricorsione bisogna che il corpo della funzione venga valutato in un ambiente in cui è già stato inserita

l'associazione tra il nome e la funzione  
Un diverso costrutto per "dichiarare" (come il let rec di ML) oppure un diverso costrutto per le funzioni ricorsive

## makefunrec



```
type eval = | Int of int | Bool of bool  
           | Unbound | Funval of efun  
and efun = expr * eval env  
and makefunrec (i, e1, (r:eval env)) =  
  let functional (rr: eval env) =  
    bind(r, i, makefun(e1,rr)) in  
  let rec rfix =  
    function x -> functional rfix x  
    in makefun(e1, rfix)
```

l'ambiente calcolato da functional contiene l'associazione tra il nome della funzione e la chiusura con l'ambiente soluzione della definizione

## Esempio di ricorsione



```
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 4]))
```

- Letrec(*i*, *e1*, *e2*) può essere visto come una notazione per Let(*i*, Rec(*i*,*e1*), *e2*)


53

## Semantica Iterativa



- non servono strutture dati diverse da quelle già introdotte per gestire i blocchi
  - la applicazione di funzione crea un nuovo frame invece di fare una chiamata ricorsiva a sem
- pila dei records di attivazione realizzata attraverso tre pile gestite in modo "parallelo"
  - envstack pila di ambienti
  - cstack pila di pile di espressioni etichettate
  - tempvalstack pila di pile di eval
- introduciamo due "nuove" operazioni 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)

54



```

let pushargs ((b: exp list),(continuation:
labeledconstruct stack) =
let br = ref(b) in
  while not(!br = []) do
    push(Expr1(List.hd!br),continuation);
    br := List.tl !br
  done

```


```

let getargs ((b: exp list),(tempstack: eval
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

```

55

## makefun, applyfun, makefunrec



```

let makefun ((a:exp),(x:eval env)) =
  (match a with
  | Fun(ii,aa) -> Funval(a,x)
  | _ -> failwith ("Non-functional object"))
let applyfun ((ev1:eval),(ev2:eval list)) =
  ( match ev1 with
  | Funval(Fun(ii,aa),r) -> newframes(aa,bindlist(r, ii, ev2))
  | _ -> failwith ("attempt to apply a non-functional object"))
let makefunrec (i, e1, (r:eval env)) =
  let functional (rr: eval env) =
    bind(r, i, makefun(e1,rr)) in
  let rec rfix = function x -> functional rfix x in
    makefun(e1, rfix)

```

56

## L'interprete iterativo 1

```
let sem ((e:expr), (rho:eval env)) =
  push(emptystack(1,Unbound),tempvalstack);
  newframes(e,r);
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      let continuation = top(cstack) in
      let tempstack = top(tempvalstack) in
      let rho = topenv() 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)
        | Let(i,e1,e2) -> push(Expr1(e1),continuation)
        | Appl(a,b) -> push(Expr1(a),continuation); pushargs(b,continuation)
        | _ -> ()))
```

57



## 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(applyenv(rho,i),tempstack)

| Fun(i, a) -> push(makefun(Fun(i, a), rho), tempstack)
| Rec(f, e) -> push(makefunrec(f, e, rho), 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(egu(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)
| Let(i,e1,e2) -> let arg=top(tempstack) in pop(tempstack);
  newframes(e2, bind(rho, i, arg))

| Appl(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=getargs(b, tempstack) in applyfun(firstarg,sndarg)

| 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"))))
done;
let valore= top(top(tempvalstack)) in pop(top(tempvalstack));
  popenv(); pop(cstack); pop(tempvalstack); push(valore,top(tempvalstack));
done;
let valore= top(top(tempvalstack)) in pop(top(tempvalstack)); pop(tempvalstack); valore;
```

58



## E' un vero interprete?



- ✚ nella implementazione attuale abbiamo una pila di ambienti relativi alle varie attivazioni
  - ognuno di questi ambienti è l'ambiente complessivo, rappresentato attraverso una funzione
- ✚ in una implementazione reale ogni attivazione ha
  - l'ambiente locale (ed un modo per reperire il resto dell'ambiente visibile)
  - l'ambiente locale dovrebbe essere "implementato" al prim'ordine (con una struttura dati)
- ✚ troveremo una situazione simile per il linguaggio imperativo con sottoprogrammi
  - dove il discorso riguarderà anche l'implementazione mediante strutture dati della memoria

59

## Digressione sullo scoping dinamico



- ✚ scoping dinamico
  - l'ambiente non locale della funzione è quello esistente al momento in cui avviene l'applicazione
- ✚ cambiano
  - efun, makefun e applyfun
- ✚ si semplifica il trattamento della ricorsione

60

# efun, makefun, applyfun



```
type efun = exp
```

```
let rec makefun (a:exp) =
  (match a with
  | Fun(ii,aa) -> Funval(a)
  | _ -> failwith ("Non-functional object"))
and applyfun ((ev1:eval),(ev2:eval list), (r:eval env)) =
  ( match ev1 with
  | Funval(Fun(ii,aa)) -> sem(aa, bindlist( r, ii, ev2))
  | _ -> failwith ("attempt to apply a non-functional object"))
```

61

# La semantica operativa



```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> applyenv(r,i)
  | Iszero(a) -> iszero(sem(a, r))
  | Eq(a,b) -> equ(sem(a, r),sem(b, r))
  | Prod(a,b) -> mult(sem(a, r), sem(b, r))
  | Sum(a,b) -> plus(sem(a, r), sem(b, r))
  | Diff(a,b) -> diff(sem(a, r), sem(b, r))
  | Minus(a) -> minus(sem(a, r))
  | And(a,b) -> et(sem(a, r), sem(b, r))
  | Or(a,b) -> vel(sem(a, r), sem(b, r))
  | Not(a) -> non(sem(a, r))
  | Ifthenelse(a,b,c) -> let g = sem(a, r) in
    if typecheck("bool",g) then
      (if g = Bool(true) then sem(b, r) else sem(c, r))
    else failwith ("nonboolean guard")
  | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
  | Fun(i,a) -> makefun(Fun(i,a))
  | Appl(a,b) -> applyfun(sem(a, r), semlist(b, r), r)
val sem : exp * eval env -> eval = <fun>
```

62

## Si riescono a trattare funzioni ricorsive?

```
Let("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 4]))
```

guardando la semantica dei tre costrutti che ci interessano

```
let rec sem ((e:exp), (r:eval env)) =  
  match e with  
  | Let(i,e1,e2) -> sem (e2, bind (r ,i, sem(e1, r))  
  | Fun(ii, aa) -> Funval(Fun(ii,aa))  
  | Appl(a,b) -> match sem(a, r) with Funval(Fun(ii,aa)) ->  
    sem(aa, bindlist(r, ii, semlist(b, r)))
```

vediamo che

- o il corpo (che include l'espressione Den "fact" ) è valutato in un ambiente che è
  - ✓ quello in cui si valuta la Appl ricorsiva
  - ✓ esteso con una associazione per i parametri formali "ii"
- o tale ambiente contiene l'associazione tra il nome "Fact" e la funzione, perché la Appl ricorsiva viene eseguita in un ambiente in cui ho inserito (nell'ordine) le seguenti associazioni
  - ✓ fact (semantica del Let)
  - ✓ ii (parametri formali della prima chiamata)

per permettere la ricorsione non c'è bisogno di un costrutto apposta

- o si ottiene gratuitamente

63

## Interprete iterativo con scoping dinamico

quasi identico a quello con scoping statico

dato che sono diverse makefun e applyfun, cambia il modo di invocarle nella "seconda passata" dell'interprete

```
let rec makefun (a:exp) =  
  (match a with  
  | Fun(ii,aa) -> Funval(a)  
  | _ -> failwith ("Non-functional object"))  
and applyfun ((ev1:eval),(ev2:eval list), (r:eval env)) =  
  ( match ev1 with  
  | Funval(Fun(ii,aa)) -> newframes(aa,bindlist(r, ii, ev2))  
  | _ -> failwith ("attempt to apply a non-functional object"))  
  
let sem ((e:exp), (rho:eval env)) =  
  ...  
  | Expr2(x) -> (pop(continuation); (match x with  
  | ...  
  | Fun(i, a) -> push(makefun(Fun(i, a)), tempstack)  
  | ...  
  | Appl(a,b) -> let firstarg=top(tempstack) in pop(tempstack);  
    let sndarg=getargs(b, tempstack) in applyfun(firstarg,sndarg,rho)  
  | ...  
  done; ....
```

64



## Scoping statico e dinamico

- ☛ la differenza fra le due regole riguarda l'*ambiente non locale*
  - l'insieme di associazioni che nel corpo di una funzione (o di un blocco) sono visibili (utilizzabili) pur appartenendo all'*ambiente locale* di altri blocchi o funzioni
- ☛ per le funzioni, l'*ambiente non locale* è
  - se lo scoping è statico, quello in cui occorre la astrazione funzionale, determinato dalla struttura sintattica di annidamento di blocchi (Let) e astrazioni (Fun e Rec)
  - se lo scoping è dinamico, quello in cui occorre la applicazione di funzione, determinato dalla struttura a run time di valutazione di blocchi (Let) e applicazioni (Apply)
- ☛ vengono "ereditate" tutte le associazioni per nomi che non vengono ridefiniti
  - (scoping statico) in blocchi e astrazioni più interni (nella struttura sintattica)
  - (scoping dinamico) in blocchi e applicazioni successivi (nella sequenza di attivazioni a tempo di esecuzione)
- ☛ un riferimento non locale al nome  $x$  nel corpo di un blocco o di una funzione e viene risolto
  - se lo scoping è statico, con la (eventuale) associazione per  $x$  creata nel blocco o astrazione più interni fra quelli che sintatticamente "contengono" e
  - se lo scoping è dinamico, con la (eventuale) associazione per  $x$  creata per ultima nella sequenza di attivazioni (a tempo di esecuzione)
- ☛ in presenza del solo costruito di blocco, non c'è differenza fra le due regole di scoping
  - perché non c'è distinzione fra definizione e attivazione
    - ✓ un blocco viene "eseguito" immediatamente quando lo si incontra

65

## Scoping statico e dinamico: verifiche

- ☛ un riferimento non locale al nome  $x$  nel corpo di un blocco o di una funzione e viene risolto
  - se lo scoping è statico, con la (eventuale) associazione per  $x$  creata nel blocco o astrazione più interni fra quelli che sintatticamente "contengono" e
  - se lo scoping è dinamico, con la (eventuale) associazione per  $x$  creata per ultima nella sequenza di attivazioni (a tempo di esecuzione)
- ☛ scoping statico
  - guardando il programma (la sua struttura sintattica) siamo in grado di
    - ✓ verificare se l'associazione per  $x$  esiste
    - ✓ identificare la dichiarazione (o il parametro formale) rilevanti e conoscere quindi l'eventuale informazione sul tipo
  - il compilatore può "staticamente"
    - ✓ determinare gli errori di nome (identificatore non dichiarato, unbound)
    - ✓ fare il controllo di tipo e rilevare gli eventuali errori di tipo
- ☛ scoping dinamico
  - l'esistenza di una associazione per  $x$  ed il tipo di  $x$  dipendono dalla particolare sequenza di attivazioni
  - due diverse applicazioni della stessa funzione, che utilizza  $x$  come non locale, possono portare a risultati diversi
    - ✓ errori di nome si possono rilevare solo a tempo di esecuzione
    - ✓ non è possibile fare controllo dei tipi statico

66

## Scoping statico: ottimizzazioni



- ✦ un riferimento non locale al nome  $x$  nel corpo di un blocco o di una funzione ~~e viene risolto~~
  - con la (eventuale) associazione per  $x$  creata nel blocco o astrazione più interni fra quelli che sintatticamente “contengono” e
- ✦ guardando il programma (la sua struttura sintattica) siamo in grado di
  - verificare se l’associazione per  $x$  esiste
  - identificare la dichiarazione (o il parametro formale) rilevanti e conoscere quindi l’eventuale informazione sul tipo
- ✦ il compilatore potrà ottimizzare l’implementazione al prim’ordine dell’ambiente (che non abbiamo ancora visto)
  - sia la struttura dati che lo implementa
  - che l’algoritmo che permette di trovare l’entità denotata da un nome
- ✦ tali ottimizzazioni, come vedremo, sono impossibili con lo scoping dinamico

67

## Regole di scoping e linguaggi



- ✦ lo scoping statico è decisamente migliore
- ✦ ~~l’unico linguaggio importante che ha una regola di scoping dinamico è LISP~~
  - questo spiega alcune delle caratteristiche “strane” di LISP, come la scarsa attenzione ai tipi ed alla loro verificabilità
- ✦ alcuni linguaggi non hanno regole di scoping
  - l’ambiente è locale oppure globale
  - non ci sono associazioni ereditate da altri ambienti locali
  - PROLOG, FORTRAN, JAVA
- ✦ avere soltanto ambiente locale ed ambiente non locale con scoping statico crea problemi rispetto alla modularità ed alla compilabilità separata
  - PASCAL
- ✦ soluzione migliore
  - ambiente locale, ambiente non locale con scoping statico e ambiente globale basato su un meccanismo di moduli

68



## LE CHIUSURE

69



## Chiusura

- ✎ Una chiusura e' una astrazione funzionale con associato l'ambiente in cui e' stata definita
- ✎ Necessario avere a disposizione l'ambiente perche' l'astrazione funzionale puo' essere invocata da un qualunque altro punto del programma (e si devono risolvere i riferimenti non locali)

# Chiusura



JavaScript closure:

```
function makeAdder(x) {  
    return function(y) {return x + y;}  
}  
...  
var add10 = makeAdder(10);  
var add5 = makeAdder(5);  
document.write("add 10 to 20: " + add10(20) +  
    "<br />");  
document.write("add 5 to 20: " + add5(20) +  
    "<br />");
```

- Chiusura = funzione anonima restituita da makeAdder

# Chiusure



 C#

- Func<int, int> (the return type) specifies a delegate that takes an int as a parameter and returns an int

```
static Func<int, int> makeAdder(int x) {  
    return delegate(int y) {return x + y;};  
}  
...  
Func<int, int> Add10 = makeAdder(10);  
Func<int, int> Add5 = makeAdder(5);  
Console.WriteLine("Add 10 to 20: {0}", Add10(20));  
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```