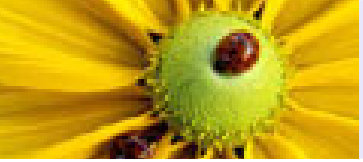
Linguaggi di Programmazione I – Lezione 3

Prof. Marco Faella
mailto://m.faella@unina.it
http://wpage.unina.it/mfaella
Materiale didattico elaborato con i Proff. Sette e Bonatti

18 marzo 2024

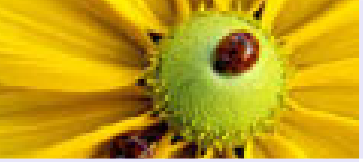


Panoramica della lezione

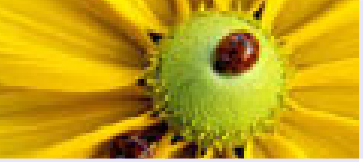
Procedure come astrazioni

Propagazione dei data object

Bibliografia

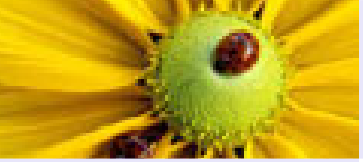


Procedure come astrazioni



Procedure

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

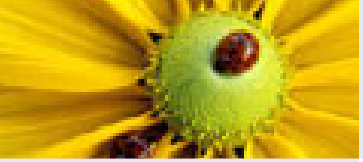


Procedure

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.

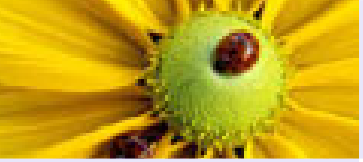


Procedure

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.

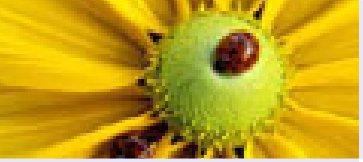


Procedure

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

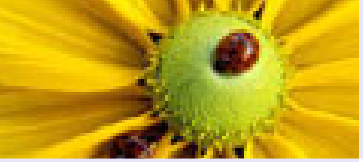
Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.
- Riusabilità di brani di programmi; riduzione errori.



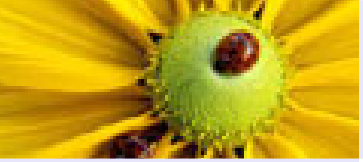
Astrazione procedurale

- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce



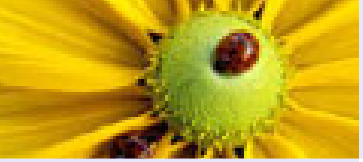
Astrazione procedurale

- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce
- **astrazione procedurale** la rappresentazione di una unità di esecuzione attraverso un'altra unità più semplice (l'invocazione)



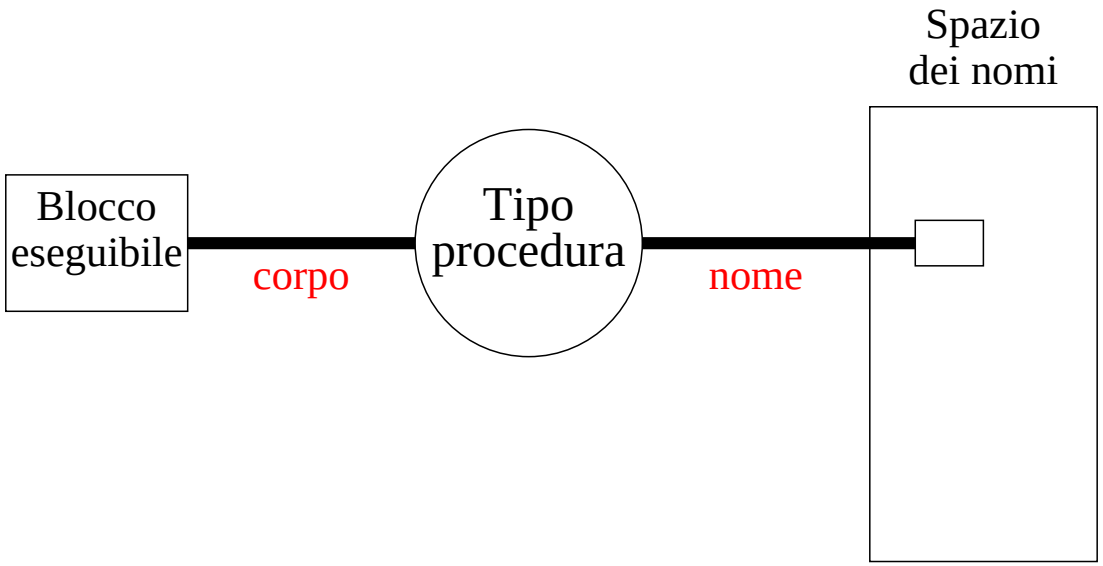
Astrazione procedurale

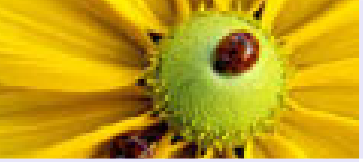
- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce
- **astrazione procedurale** la rappresentazione di una unità di esecuzione attraverso un'altra unità più semplice (l'invocazione)
- In pratica è la rappresentazione di un blocco attraverso un enunciato o una espressione.



Dichiarazione di procedura

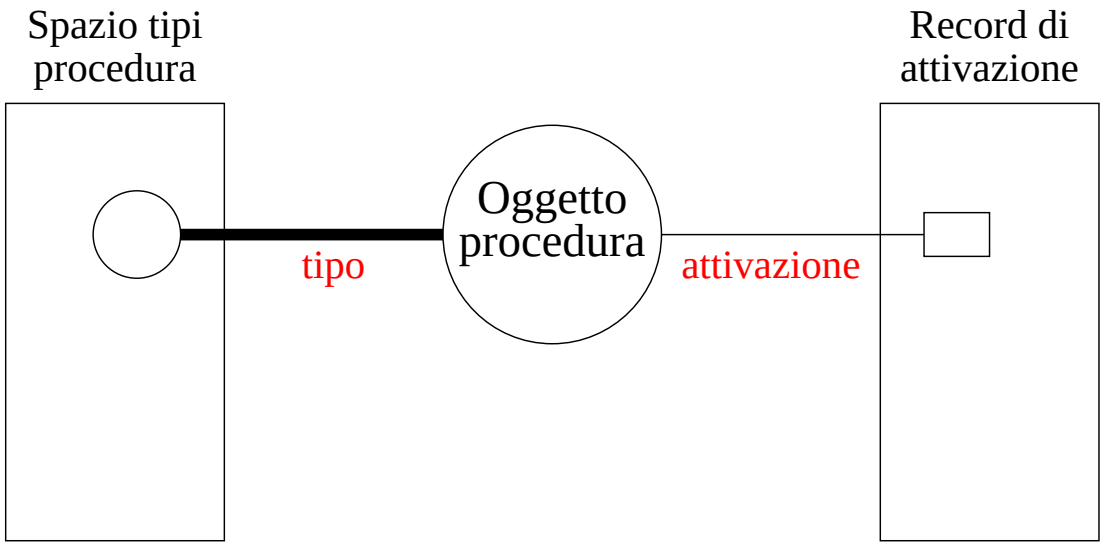
- Causa la generazione di un legame tra il nome della procedura e il corpo della procedura (oltre a parametri, tipo di ritorno, etc.)
- Chiamiamo questo legame un oggetto "Tipo procedura"
- Il legame viene stabilito durante la compilazione

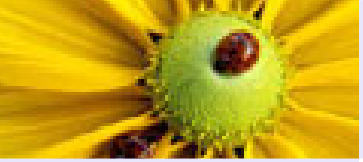




Invocazione di una procedura

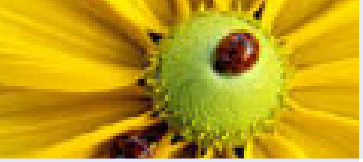
- Causa la generazione di un legame tra un Tipo procedura e un Record di attivazione
- Chiamiamo questo legame un "Oggetto procedura"
- Il legame avviene durante l'esecuzione, nel momento in cui c'è l'invocazione della procedura
- Ogni invocazione diversa della stessa procedura causa la generazione di un nuovo "Oggetto procedura" con lo stesso legame di tipo, ma con diverso record di attivazione





Ambiente di esecuzione

Analogamente a quanto avveniva per un blocco in-line il **record di attivazione (RdA)** rappresenta l'intero ambiente di esecuzione di una procedura o funzione

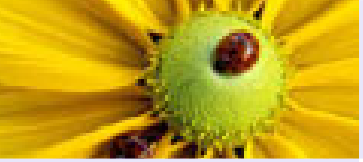


Ambiente di esecuzione

Analogamente a quanto avveniva per un blocco in-line il **record di attivazione (RdA)** rappresenta l'intero ambiente di esecuzione di una procedura o funzione
A differenza del blocco anonimo, una procedura o funzione...

- Ha un nome e può essere invocata in vari punti del programma
- Può avere parametri
- Può avere un valore di ritorno (se funzione)

Quindi, l'RdA di una funzione contiene più informazioni di quello di un blocco in-line

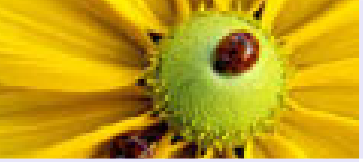


Ambiente di esecuzione

L'RdA di una funzione contiene:

1. Puntatore di catena dinamica (link al precedente RdA)
2. Puntatore di catena statica (new!) (opzionale, serve per scoping statico)
3. Indirizzo di ritorno (new!)
4. Indirizzo del risultato (new!)
5. Parametri (new!)
6. Ambiente locale (variabili locali e spazio per risultati intermedi)

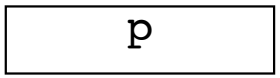
L'ambiente non locale di una funzione viene recuperato tramite il puntatore di catena dinamica (scoping dinamico) o statica (scoping statico)



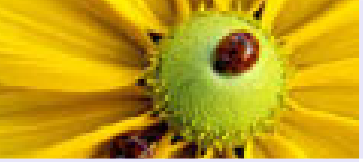
Esempio

Stack esecuzione

p: prima di chiamare s



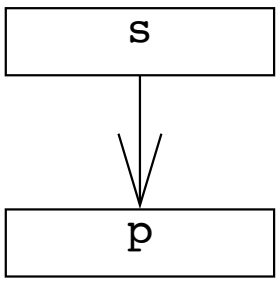

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



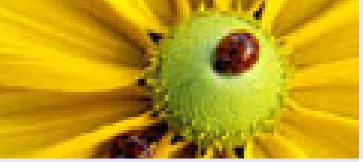
Esempio

Stack esecuzione

s: prima di chiamare r



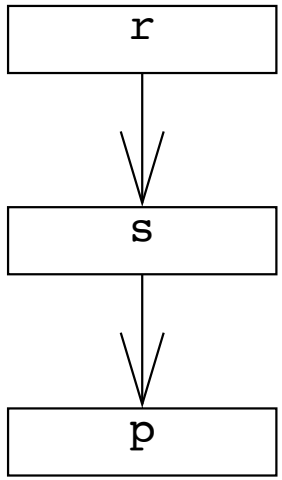

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



Esempio

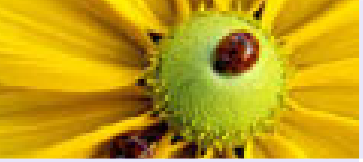
Stack esecuzione

r: prima di chiamare r



XS

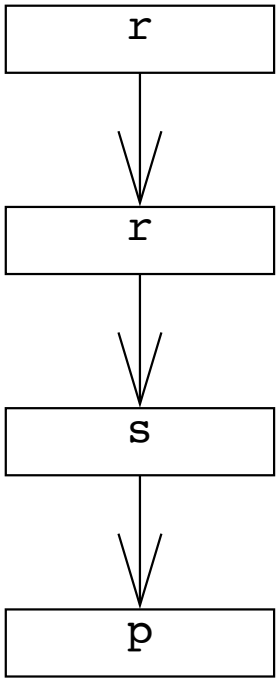
```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



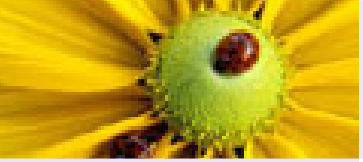
Esempio

Stack esecuzione

r: prima di chiamare q



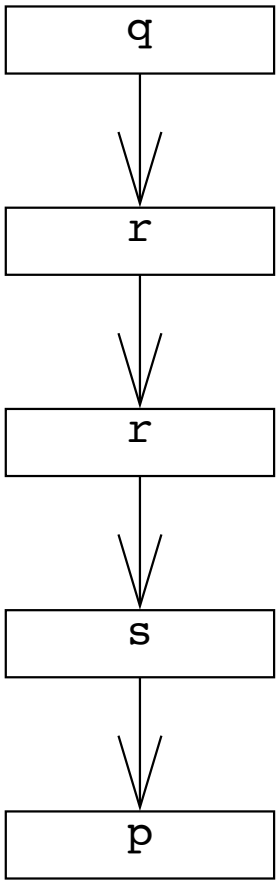

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



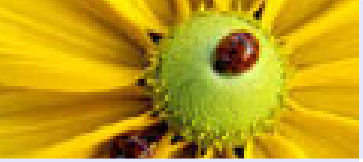
Esempio

Stack esecuzione

q: prima di terminare



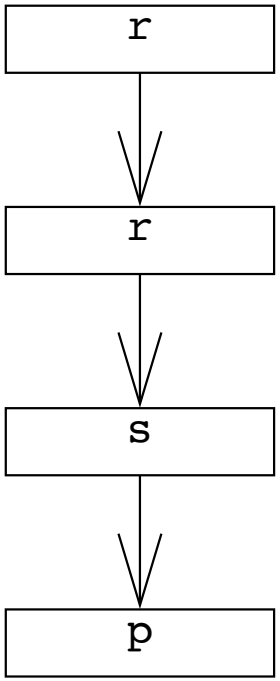

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



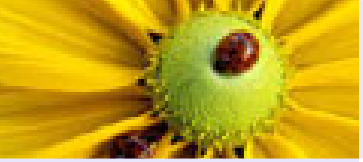
Esempio

Stack esecuzione

r: prima di terminare



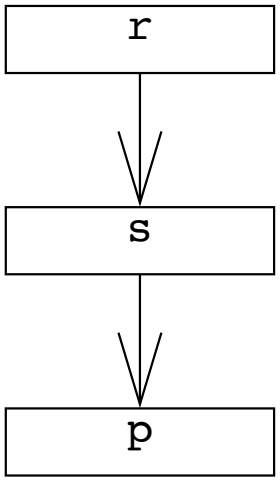

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



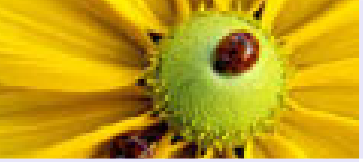
Esempio

Stack esecuzione

r: prima di terminare



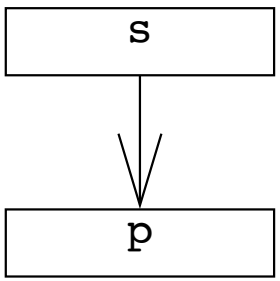

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



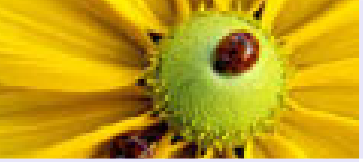
Esempio

Stack esecuzione

s: prima di q



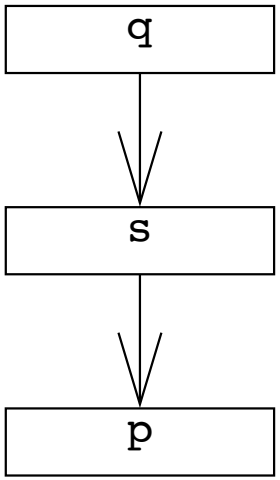

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



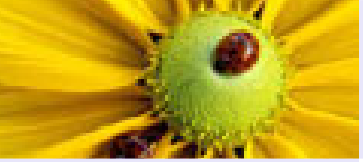
Esempio

Stack esecuzione

q: prima di terminare



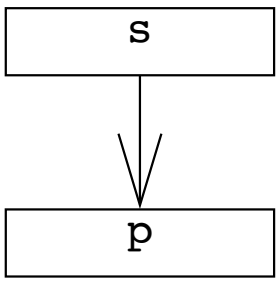

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



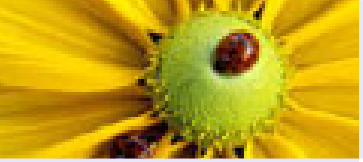
Esempio

Stack esecuzione

s: prima di terminare



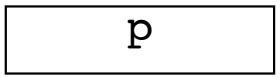

```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

  s

end.
```



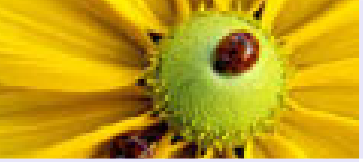
Esempio

Stack esecuzione

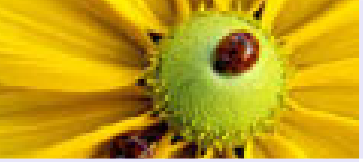
p: prima di terminare




```
program p;
var i;

  procedure q;
  begin
    ...
  end;

  procedure r;
  begin
    i:= i-1;
    if i>0 then
      r
    else
      q
  end;

  procedure s;
  begin
    r;
    q
  end;

begin

  i:= 2;

○ s

● end.
```

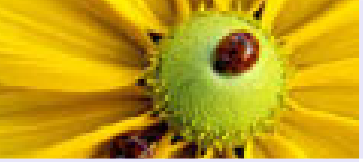


Propagazione dei data object



Ambiente non locale

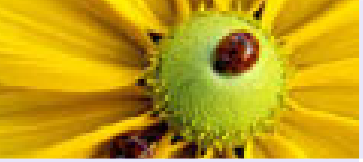
- L'*ambiente non locale* è l'insieme di tutti quei nomi a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):

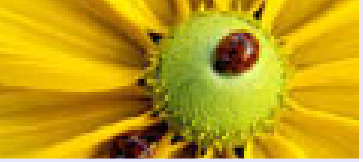
  1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):

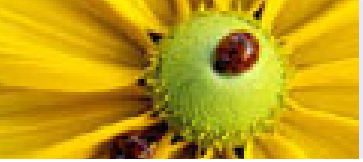
1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.
2. Propagazione in ambito dinamico (scoping dinamico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura chiamante.



Ambiente non locale

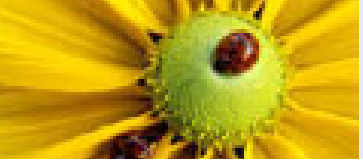
- L'*ambiente non locale* è l'insieme di tutti quei nomi a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):

1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.
2. Propagazione in ambito dinamico (scoping dinamico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura chiamante.
3. Nessuna propagazione (in generale, l'uso di ambienti non locali è scoraggiato perché produce effetti collaterali non facilmente prevedibili)



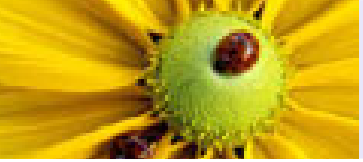
Implementazione

- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object



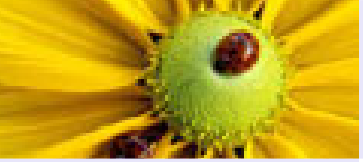
Implementazione

- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object
  - Scoping statico: viene usato il puntatore di catena statica
  - Scoping dinamico: viene usato il puntatore di catena dinamica
- Se viene richiesto l'accesso ad un dato che non è definito localmente, esso viene ricercato ricorsivamente seguendo la catena appropriata



Implementazione

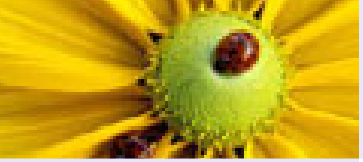
- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object
  - Scoping statico: viene usato il puntatore di catena statica
  - Scoping dinamico: viene usato il puntatore di catena dinamica
- Se viene richiesto l'accesso ad un dato che non è definito localmente, esso viene ricercato ricorsivamente seguendo la catena appropriata



Il caso dei blocchi in-line

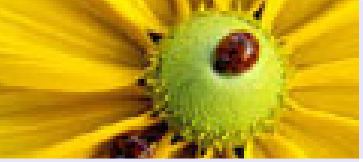
L'RdA di un blocco in-line non contiene il puntatore di catena statica (anche in presenza di scoping statico) perché viene eseguito sempre nel contesto del blocco in cui è contenuto sintatticamente

Quindi, l'eventuale puntatore di catena statica *coinciderebbe con quello di catena dinamica*



Il caso del C

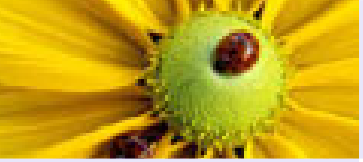
- Scoping statico
- Non supporta funzioni annidate
- Solo blocchi in-line (anonimi) annidati
- Quindi, non serve il puntatore di catena statica
- Lo scope non locale contiene: i blocchi esterni a quello corrente, poi la funzione corrente e infine lo scope globale



Scoping statico con procedure annidate

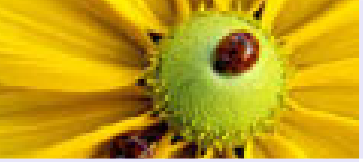
Pseudo-codice:

```
program p;
  var a, b, c: integer;

  procedure q;
    var a, c: integer;
    procedure r;
      var a: integer;
      begin r          {variabili: a da r; b da p; c da q;
        ...             procedure: q ed s da p; r da q}
      end r;
    begin q            {variabili: a da q; b da p; c da q;
       ...              procedure: q ed s da p; r da q}
    end q;

  procedure s;
    var b: integer;
    begin s            {variabili: a da p; b da s; c da p;
      ...              procedure: q ed s da p}
    end s;

begin p                {variabili: a, b, c da p;
  ...                   procedure: q, s da p}
end p.
```



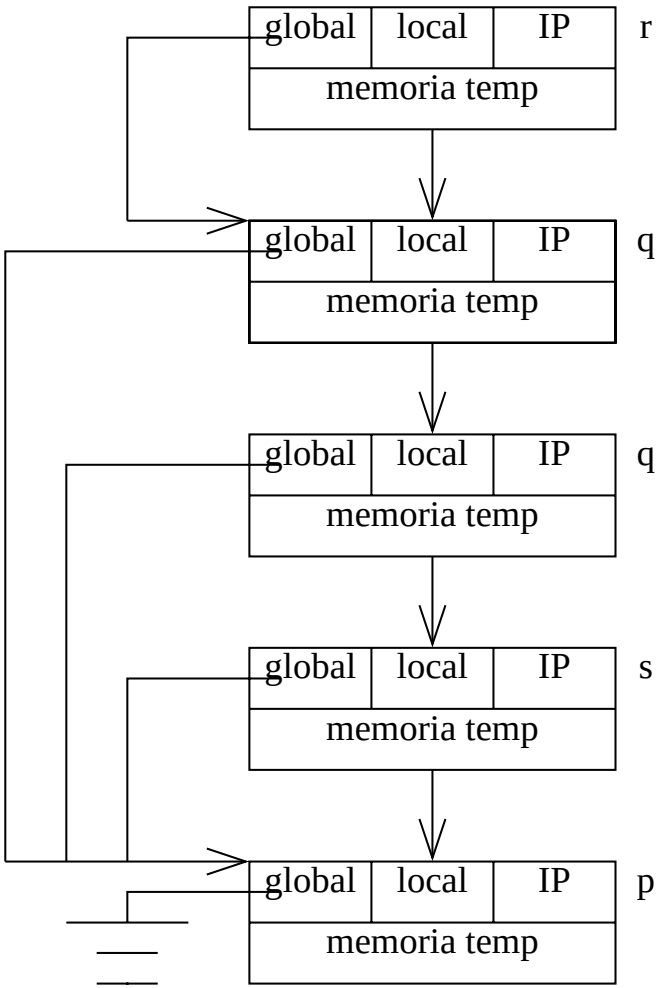
Scoping statico (2)

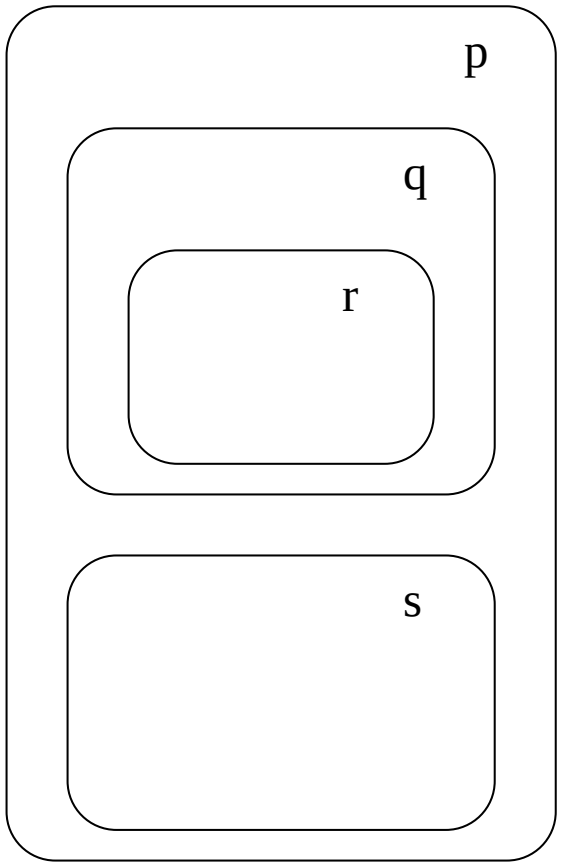
Supponendo una sequenza di attivazione (`p`, `s`, `q`, `q`, `r`), lo stack di esecuzione ha questa forma:

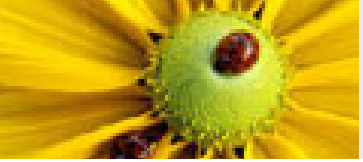


Scoping statico (2)

Supponendo una sequenza di attivazione (`p`, `s`, `q`, `q`, `r`), lo stack di esecuzione ha questa forma:



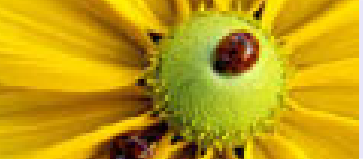




Osservazioni su scoping dinamico

Nello scoping dinamico:

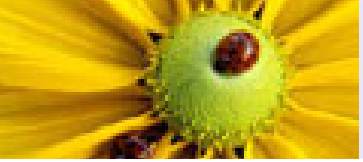
- il puntatore di catena statica non è necessario
- è sufficiente il puntatore di catena dinamica



Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
- è sufficiente il puntatore di catena dinamica
- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)

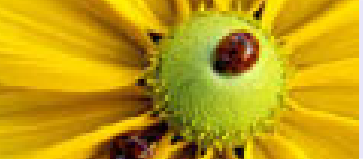


Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
- è sufficiente il puntatore di catena dinamica

- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)
- Svantaggio: è praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.

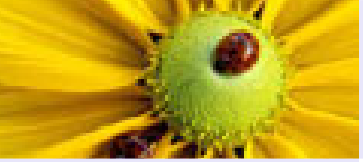


Osservazioni su scoping dinamico

Nello scoping dinamico:

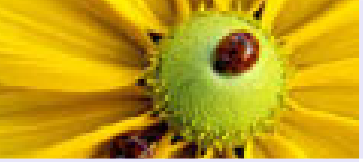
- il puntatore di catena statica non è necessario
- è sufficiente il puntatore di catena dinamica

- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)
- Svantaggio: è praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.



Esempio di scoping dinamico

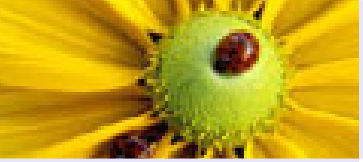

```
program p;
  var a: integer;
  procedure q;
    begin q              {vars: a da p o da r; procs: q, r da p}
      ...
    end q;
  procedure r;
    var a: integer;
    begin r              {vars: a da r; procs: q, r da p}
      ...
    end r;
begin p                  {vars: a da p; procs: q, r da p}
  ...
end p.
```



Esempio di scoping dinamico

Linguaggio BASH (shell Linux):

```
> x=3
> echo $x
3

> func1 () { echo "in func1: $x"; }
> func1
in func1: 3

> func2 () { local x=9; func1; }
> func2
in func1: 9
```

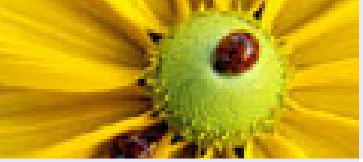


Linguaggi con procedure annidate

I seguenti linguaggi supportano l'annidamento di procedure/funzioni:

- I linguaggi funzionali (ML, Scheme, Common LISP, etc.)
- Javascript
- Kotlin
- C# (dalla versione 7.0)
- Estensione GCC del C

Altri linguaggi OO, come Java e C++, supportano l'annidamento di *classi*



Bibliografia

Linguaggi di Programmazione, principi e paradigmi, di Gabbrielli e Martini (2a edizione):

- Capitolo 7 ("La gestione della memoria")
- Capitolo 9 ("Astrarre sul controllo")