

Nomi

nomi, ambiente, blocchi, regole di scope

M. Gabbielli, S. Martini

Linguaggi di programmazione:

principi e paradigmi

McGraw-Hill Italia, 2005

Evoluzione dei linguaggi

- Anni 40-50 Preistoria: (linguaggio macchina e assembler)
- Anni 50-60: Primi linguaggi alto livello:
 - **FORTRAN** (^{FORMULA - TRANSLATOR} Backus 1957) per calcolo numerico-scientifico (su IBM704): uso di notazione matematica in espressioni (es. $i+2*j$)
 - **ALGOL (58, 60, W)** (Naur 1958) come linguaggio universale, per esprimere algoritmi: Indipendenza dalla macchina, vicinanza con la notazione matematica, call by name, **funzioni ricorsive**, type systems, data structures (anche dinamiche)
 - **LISP** (^{Unico tipo di dato: lista} McCarthy 59-60) per intelligenza artificiale (List Processor): funzioni su una certa classe di espressioni simboliche (S-Expressions), ordine superiore (**gennitori dei linguaggi funzionali** antesignano del paradigma funzionale)
 - **COBOL** (59-60) per applicazioni commerciali (record, archivi)
 - **ALGOL 68** linguaggio barocco, complessa progettazione, ma contiene anche idee importanti (**struttura a blocchi**)
 - **SIMULA** (Nygaard & Dahl): da ALGOL, primo linguaggio con classi e oggetti (**antesignano del paradigma object-oriented**)

Evoluzione dei linguaggi II

- Anni 70:
 - **PL/I** (IBM): Fortran + COBOL, primo esempio di linguaggio multi-uso (PL deriva da Multi-Purpose Prog. Lang.) Scarso successo
 - **Pascal** (*uno dei primi ad usare il codice intermedio (P-code)*): Evoluzione (e semplificazione) di Algol W. Essenzialmente linguaggio didattico fino agli anni 80 (poi rimpiazzato da C e Java). Portabilità grazie a codice intermedio (P-Code)
 - **C** (Ritchie): Programmazione di sistema (Unix). Ha avuto notevole successo fino ai nostri giorni.
 - **Prolog**: (Colmerauer, Kowalski): uso esplicito della logica (di primo ordine) come base di un linguaggio di programmazione (primo linguaggio logico)
 - **SmallTalk**: Incapsulamento, oggetti e classi. O-O. (*primo vero linguaggio object-oriented*)
 - **ML** (Milner): Funzionale. Nato come Meta Linguaggio per un sistema semi-automatico di prova di proprietà dei programmi. Sistema di tipi evoluto.

Evoluzione dei linguaggi III

- Anni 80:

- ADA: Linguaggio real-time di US Dep. Of Defense.
Linguaggio per sviluppo di programmi molto grandi da parte di team. Include quasi tutto quello che esisteva al tempo, inclusi costrutti per il controllo della concorrenza
 - Postscript: page description language nell'area di desktop publishing elettronico
 - C++ Estensione Object Oriented di C
 - Standard ML
 - CLP (linguaggi logici con vincoli). Linguaggi che permettono di manipolare relazioni su opportuni domini. Usati per problemi combinatori, ottimizzazione, intelligenza artificiale etc.

Evoluzione dei linguaggi IV

- Anni 90: → nasce il World Wide Web
 - Java (Gosling): Altamente portabile; object-oriented; programmi spediti sulla rete (applet). Elevata portabilità (Write once, run anywhere!) --> Java Virtual Machine e ByteCode *è eseguibile su qualsiasi tipo di macchina*
 - PERL: Linguaggio per applicazioni di text-processing nato come linguaggio di shell-scripting in UNIX
 - HTML linguaggi di marcatura per ipertesti (usato per creare pagine web)
 - XML linguaggio per la rappresentazione di dati semi-strutturati (ad esempio, pagine web).

Evoluzione dei linguaggi V

- Anni dal 2000 in poi:
 - service-oriented computing: paradigma che usa i “servizi” (di solito web services), come unità di base di computazione, per progettare ed implementare applicazioni integrate di business.
 - **WSDL**: web services description language
 - **WS-BPEL**: Web Services Business Process Execution Language
 - **Jolie (sviluppato a Bologna!)**
 - Qual'è il linguaggio più potente? Sono tutti equipotenti perché è possibile realizzare una macchina di Turing
 - **Agent Based Programming** (vedi corso di Sacerdoti Coen in LM) Go, Erlang, Rust, ecc..
 - **Scalable Cloud Programming** (vedi corso di Zavattaro) – Scala e Spark

TURING - COMPLETO \rightarrow formalismo che indica
che un certo linguaggio
abbia la stessa potenza della
macchina di Turing

La macchina di TURING è un formalismo di calcolo com'è macchina di Turing

un mostro infinito con una testina, può:

- leggere un carattere
- andare avanti e indietro

- \rightarrow c'è il dispositivo di calcolo più potente
(cioè la macchina di Turing può realizzare una
macchina che realizza il mostro algoritmo)
- \rightarrow Congettura di Churchill: Non può esistere un
dispositivo più potente
della macchina di Turing

Esistono linguaggi di programmazione NON
TURING COMPLETI? SÌ

Esempio: linguaggio di programmazione fatto dalla sola
istruzione Skip, per esempio

\rightarrow Per dire che il calcolatore non può fare tutto

N.B. Molti funzioni sono NON CALCOLABILI \rightarrow Non è possibile scrivere
un algoritmo che calcoli
quella funzione

Es. $F = \{f \mid f: \mathbb{N} \rightarrow \{0, 1\}\}$ \rightarrow Insieme delle funzioni booleane
(NON è NUMERABILE)

Linguaggi imperativi e linguaggi dichiarativi

- **Linguaggi imperativi** → chiamati così perché sono simili alle frasi imperative espresse in ^{linguaggio naturale} _{costruito di base: ASSEGNAZIONE}
- Basati sulla nozione di **stato** (insieme di locazioni di memoria contenenti dei valori) → in tali locazioni di memoria
- Le istruzioni sono **comandi** che cambiano lo **stato**.
- Es. $X := X + 1; X := X + 2 \dots$
- C, Pascal, FORTRAN, COBOL, ..., C++ (per alcuni aspetti)
- Analogia con le frasi *imperative* dove il soggetto è implicito (e.g. taglia quella mela)

- legato al fatto che "dichiarano" delle funzioni*
- **Linguaggi dichiarativi** → esprimono delle funzioni _{NO ASSEGNAZIONE, NO VARIABILI MODIFICABILI (COSTANTI)}
 - Basati sulla nozione di **funzioni o relazione**.
 - Le istruzioni sono **dichiarazioni** di nuovi valori, in modo diretto oppure per composizione di funzioni o relazioni.
 - Es. $(\text{fun}(X). X + 2) 3;$
 - Analogia con le frasi *dichiarative* (e.g., quella mela è tagliata)
→ devo dichiarare come si fa la soluzione, non come risolvere un certo problema

Linguaggi dichiarativi

- Linguaggi **funzionali**: ^{→ basati sulla funzione} Lisp, Scheme, ML, Haskell
 - Basati sulla nozione di **funzione**: risultato del programma = valore esplicito di una espressione
 - applicazione e definizione di funzioni, ricorsione
 - Programmare = costruire la funzione che calcola il risultato
- Linguaggi **logici** (e con vincoli): Prolog, CLP
 - Basati su **relazioni**: risultato = valore di alcune variabili determinato da alcune relazioni
 - Istruzioni = implicazioni logiche fra opportune formule, che possono essere viste come regole di riscrittura
 - Programmare = specificare la relazione che definisce il valore delle variabili di interesse (sostanzialmente ``dire come è fatta la soluzione'')
- → Nei linguaggi logici ho **DEFINIZIONI** e **APPLICATIONI** di **DEFINIZIONI** (e sono TURING-completi)

Nomi

- nome (in un linguaggio di alto livello)

DEF – sequenza di caratteri usata per denotare qualche cos'altro
Artificio che usiamo per indicare qualcosa (siamo usate un nome)

– const pi = 3.14;

– int x;

– void f(...);

nomi

oggetto denotato:

la costante 3.14

una variabile

la definizione di f

- Nei linguaggi i nomi sono spesso **identificatori** (token alfano numerici)
 - ma possono essere anche altro (+, := ...)
- L'uso di un nome serve ad indicare l'oggetto denotato
- Oggetti simbolici più facili da ricordare
- Astrazione
 - dati (variabili, tipi ecc.)
 - controllo (sottoprogrammi)

→ i nomi sono un meccanismo di astrazione.

Oggetti denotabili

- **Oggetto denotabile**
– quando può essergli associato un nome
sono quegli oggetti a cui puo'
- Nomi definiti dall'utente
 - variabili, parametri formali, procedure (in senso lato), tipi definiti dall' utente, etichette, moduli, costanti definite dall' utente, eccezioni
- Nomi definiti dal linguaggio
 - tipi primitivi, operazioni primitive, costanti predefinite.
Ese. int è un nome: indica il tipo di dato intero
definito dal linguaggio
- Terminologia:
 - Legame (binding), o associazione, tra nome e oggetto
lingame/cio' che si instaura

Binding time

- **Statico** → Tutto quello che avviene prima dell'esecuzione del programma
 - Progettazione del linguaggio
 - Tipi primitivi, nomi per operazioni e costanti predefinite ecc. Es. ~~int è tipo primitivo~~ associato a quel tipo di dato nel momento in cui definisco il linguaggio

- Scrittura del programma
 - definizione di alcuni nomi (variabili, funzioni ecc.) il cui legame sarà completato più tardi
- Compilazione (+collegamento e caricamento)
 - legame di alcuni nomi (var globali)

- **Dinamico** → A run-time, cioè durante l'esecuzione del programma
 - Esecuzione
 - legame definitivo di tutti i nomi non ancora legati (pe. variabili locali ai blocchi)

Terminologia

In italiano usiamo:

- *binding* = legame = associazione
- *environment* = ambiente
- *scope* = portata, estensione (anche: ambito, campo d'azione)
↳ *Scope di una dichiarazione: è la parte di codice in cui tale dichiarazione vale*
- *lifetime* = vita, o tempo di vita

Esempio

(In pseudocodice di un linguaggio imperativo)

blocco: ← variabile: (valore, ora non ci interessa)

nomi dei blocchi
(in alcuni linguaggi
può essere dato)

Cosa c'è nell'ambiente
in questo punto?

O equivalememente

Quali variabili sono
visibili e cosa demando
fino a questo punto
delle esecuzioni?

(risposta sopra in verde)

A: { int a = 1;

B: { int b = 2;
int c = 2;

C: { int c = 3;
int d;
d = a+b+c;
write (d)

D: { int e;
e = a+b+c;
write (e)

AMBIENTE
→ insieme di
associaz.

A ← a
B ← b
C ← c
B ← c

1

2

3

MASCHERATA

...

quando esco dal blocco C, d viene distruito

Sono qua

Durante l'esecuzione

Ambiente

Ambiente:

DEF insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione

cioè in un punto preciso dell'esecuzione del programma

Dichiarazione: → parti del programma che vengono eseguite a run-time

esecuz.
a run-time

meccanismo (implicito o esplicito) col quale si crea un' associazione in ambiente

EFFETTO : modifica l'ambiente, va a modificare il binding tra nomi e variabili

```
int x;  
int f () {  
    return 0;  
}  
type T = int;
```

(Ricordo che stiamo parlando di linguaggi di alto livello)

Ambiente, 2

Lo stesso nome può denotare oggetti distinti

in punti diversi del programma

→ { int b=5
 int b = 10
 }

Aliasing

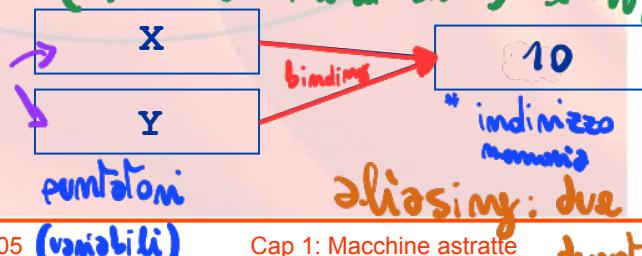
→ Non si verifica quando due variabili hanno lo stesso valore
ma quando due nomi di variabile denotano lo stesso oggetto
fenomeno dove nomi diversi denotano lo stesso oggetto
(si dice con i puntatori)

passaggio per riferimento

puntatori
ecc.

```
int *X, *Y;           // X, Y puntatori a interi
X = (int *) malloc (sizeof (int));
                    // allocata la memoria puntata
*X = 5;
                    // * dereferenzia
Y=X;
                    // Y punta alla stessa memoria di X
*Y=10;
                    // Y punta alla stessa memoria di X
write (*X);
```

nomi di
variabili
(puntatori)



aliasing: due nomi di VARIABILE

denotano lo stesso indirizzo
di memoria

A cosa serve?

Blocchi → costituto che permette di **STRUTTURARE** l'ambiente

- Nei linguaggi moderni l'ambiente è **strutturato**

- **Blocco:** → è una regione testuale contigua (pezzo di codice) compresa tra un segnale di inizio e uno di fine, che può contenere dichiarazioni locali a quella regione
 - regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni *locali* a quella regione

- **begin...end**
- **{...}**
- **(...)**
- **let...in...end**

Algol, Pascal

C, Java

Lisp

ML

- anonimo (o in-line)
- associato ad una procedura

→ il blocco può avere un nome nel momento in cui corrisponde al corpo di una funzione (nome blocco = nome funzione)

Perché i blocchi → perché permettono di strutturare l'ambiente

- ~~Senza blocchi non avremmo la ricorsione~~

- Gestione locale dei nomi

```
{int tmp = x;  
 x=y;  
 y=tmp  
}
```

- chiarezza
 - ognuno può scegliere il nome che vuole

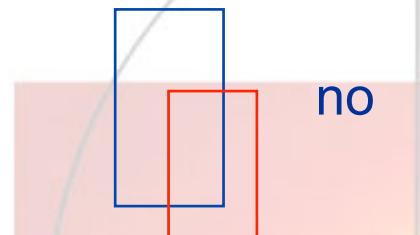
→ posso ridefinire nomi già usati in altri blocchi

- Con un'opportuna allocazione della memoria
(=> vedi dopo):

- ottimizzano l'occupazione di memoria
 - permettono la ricorsione

Annidamento → i blocchi possono essere annidati

- Blocchi sovrapposti solo se annidati



no



→ Se il blocco rosso è annidato nel blocco blu si il blocco rosso deve terminare prima del blocco blu
(Non si dà verificazione perché a livello di implementazione usiamo una pila)

- Regola di visibilità (preliminare)

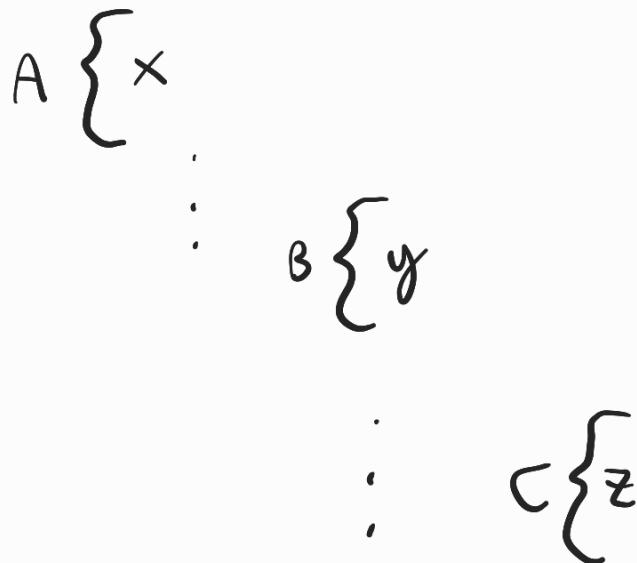
- Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che *nasconde*, o maschera, la precedente).

→ dove è ridefinita, si vede la variabile nuova e non quella vecchia (viene mascherata)

Suddividiamo l'ambiente

- L'ambiente (in uno specifico blocco) può essere suddiviso in
 - ambiente locale: associazioni create all'ingresso nel blocco
 - variabili locali
 - parametri formali
 - ambiente non locale: associazioni ereditate da altri blocchi
che NON sono state definite nel blocco in cui si sta parlando e nell' ambiente globale
 - ambiente globale: quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi
 - dichiarazioni esplicite di variabili globali
 - dichiarazioni del blocco più esterno → perché riguardano TUTTI i blocchi
(sono associaz. comuni a tutti i blocchi)
 - associazioni esportate da moduli ecc.

Esempio:



Blocco C:

ambiente locale: associaz. tra z (nominata) e l' ogg. denominato da z

amb. globale: x

amb. non locale: y

Esempio

(In pseudocodice di un linguaggio imperativo)

```
A: {int a = 1;
```

```
B: {int b = 2;  
    int c = 2;
```

```
C: {int c = 3;  
    int d;  
    d = a+b+c;  
    write(d)  
    }  
se doppio C = 10; (blocco B c'è ancora) → quando esco dal blocco C,  
l'oggetto denotato con il nome d viene  
distruito
```

```
D: {int e;  
    e = a+b+c;  
    write(e)  
    }  
}
```

BLOCCO D: (vedo le variabili in D)
amb. locale: e
amb. globale: a
amb. non locale: b, c (ereditati da blocco B)

Esempio (In pseudocodice di un linguaggio imperativo)

```
A: {int a =1;
```

```
    B: {int b = 2;  
        int c = 2;
```

```
    C: {int c =3;  
        int d;  
        d = a+b+c;  
        write (d)  
    }
```

```
    D: {int e;  
        e = a+b+c;  
        write (e)  
    }
```

```
}
```

Operazioni sull'ambiente

- **Creazione** associazione nome-oggetto denotato (naming)
 - dichiarazione locale in blocco
- **Riferimento** oggetto denotato mediante il suo nome (referencing)
 - uso di un nome
- **Disattivazione** associazione nome-oggetto denotato
 - una dichiarazione maschera un nome
- **Riattivazione** associazione nome-oggetto denotato
 - uscita da blocco con dichiarazione che maschera
- **Distruzione** associazione nome-oggetto denotato (unnaming)
 - uscita da blocco con dichiarazione locale

Operazioni sugli oggetti denotabili

- Creazione
 - Accesso
 - Modifica (se l'oggetto è modificabile)
 - Distruzione
-
- Creazione e distruzione di un oggetto non coincidono con creazione e distruzione dei legami per esso

Es: Ho un ogg. denotato e un nome



Elimino l'ogg. denotato senza eliminare il binding



Alcuni eventi fondamentali

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

Il tempo tra 1 e 7 è la **vita** (o il tempo di vita: *lifetime*)
dell'oggetto

Il tempo tra 2 e 6 è la **vita dell'associazione**

vita

La vita di un oggetto non coincide con la vita dei legami per quell'oggetto

- Vita dell'oggetto più **lunga** di quella del legame:

variabile passata ad una proc per riferimento (Pascal: var)

```
procedure P (var X:integer); begin... end;  
...  
var A:integer;  
...  
P (A) ;
```

Durante l'esecuzione di P esiste un legame tra X e un oggetto che esiste prima e dopo tale esecuzione.

$\text{free}(x)$

$$\left\{ \begin{array}{l} x \rightarrow \boxed{\quad} \\ \vdots \\ \left. \right\} \end{array} \right.$$

vita, 2

- Vita dell'oggetto più **breve** di quella del legame:

area di memoria dinamica deallocated

```
int *X, *Y;  
...  
X = (int *) malloc (sizeof (int));  
Y=X;  
...  
free (X);  
X=null;
```

Dopo la `free` non esiste più l'oggetto, ma esiste ancora un legame (“pendente”) per esso (`Y`): *dangling reference*

Regole di scope

- Come deve essere interpretata la regola di visibilità?

Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde, o maschera, la precedente)

- in presenza di procedure
cioè di blocchi che sono eseguiti in posizioni diverse dalla loro definizione
- in presenza ambiente non locale (e non globale)

Qual è lo scope?

```
{int x=10;  
void foo () {  
    x++;  
}  
void fie () {  
    int x = 0;  
    foo();  
}  
fie();  
}
```

quale x incrementa foo?

un riferimento non-locale in un blocco B può essere risolto:

nel blocco che *include sintatticamente* B

nel blocco che è *eseguito immediatamente prima* di B

scope statico

scope dinamico

Cosa stampa questo programma ?

REGOLA DI SCOPE STATICO:

```
int x = 0;  
void pippo(int n) {  
    x = n+1;  
}  
pippo(3);  
write(x);  
  
{ int x = 0;  
  pippo(3);  
  write(x);  
}  
write(x);
```

dov'è dichiarata
la x in modo
che sia visibile per
pippo

Formale (è il parametro
che compare
nella def. di
funzione)

attuale (parametro
chiamata che compare nella
di funzione)

perché la x torna
ad essere quella del blocco
sopra, questa x nella chiamata pippo
non esiste più (grado il
punto in cui è definita)

SCOPE
STATICO → quando la
struttura

Scope statico

- Un nome non locale è risolto nel blocco che *testualmente* lo racchiude

risolto dal punto di vista del riferimento

parto da quella variabile, vado all'esterno e
appena trovo il valore mi riferisco a quell'

```
{int x = 0;  
void pippo(int n) {  
    x = n+1;  
}  
pippo(3);  
write(x);  
{int x = 0;  
pippo(3);  
write(x);  
}  
write(x);  
}
```

stampa 4

stampa 0

$$\left\{ \begin{array}{l} x = 10 \end{array} \right.$$

$\left\{ \begin{array}{l} x = 0 \end{array} \right.$

valido all'esterno (del blocco)
(la prima occorrenza che incontra è valido)

$\left\{ \begin{array}{l} \text{void pippo (int } n) \{ \\ \quad x = n+1 \\ \} \\ \text{pippo (3)} \end{array} \right.$

Scope dinamico

- Un nome non locale è risolto nel blocco attivato più di recente e non ancora disattivato

in senso
tempo dell'esecuzione

REGOLA DI SCOPE DINAMICO

N.B. Il blocco deve essere ancora attivo
(la prima occorrenza
trovata deve essere attiva)

stampa 4

```
{int x = 0;  
void pippo(int n){  
    x = n+1;  
}  
pippo(3);  
write(x);  
{int x = 0;  
pippo(3);  
write(x);  
}  
write(x);  
}
```

si guarda la
STRUTTURA DINAMICA
del programma
(durante l'esecuzione)

x = 0

x = 0

x = 3

→ più usato, per il copire a quale variabili ci si riferendo

Scope statico: indipendenza dalla posizione

```
{int x=10;  
void foo () {  
    x++;  
}  
void fie () {  
    int x=0;  
    foo();  
}  
fie();  
foo();  
}
```

- il corpo di `foo` è parte dello scope della `x` più esterna
- la chiamata di `foo` è compresa nello scope della `x` più interna
- `foo` può essere chiamata in molti contesti diversi
- l'unico modo in cui `foo` può essere compilata in modo univoco è che il riferimento a `x` sia sempre quello più esterno

La chiamata di `foo` interna a `fie` e quella nel main accedono alla stessa variabile: la `x` esterna

Scope statico: indipendenza dai nomi locali

```
{int x=10;  
void foo () {  
    x++;  
}  
void fie () {  
    int y =0; → cambio x in y  
    foo();  
}  
fie();  
foo();  
}
```

la modifica del locale **x** in **y** dentro **fie**

- modifica la semantica del programma in scope **dinamico**
- non ha alcun effetto in scope **statico**

perché
con scope
dinamico
quella che prima
era la prima
occorrenza di x
ora è di y

Un principio di indipendenza:

Ridenominazioni consistenti dei nomi locali di un programma non devono avere effetto sulla semantica del programma stesso

Scope dinamico: specializzare una funzione

- **visualizza** è una procedura che rende a colore sul video una certa forma

```
...
{var colore = rosso;
 visualizza(testo);
}
```

Scope statico vs dinamico

- **Scope statico (*scoping statico, statically scoped, lexical scope*).**
 - informazione completa dal testo del programma
 - le associazioni sono note a tempo di compilazione
 - dunque: principi di indipendenza
 - concettualmente più complesso da implementare ma più efficiente
 - Algol, Pascal, C, Java, ...
- **Scope dinamico (*scoping dinamico, dynamically scoped*).**
 - informazione derivata dall'esecuzione
 - spesso causa di programmi meno ``leggibili''
 - concettualmente più semplice da implementare, ma meno efficiente
 - Lisp (alcune versioni), Perl
- Differiscono solo in presenza congiunta di
 - ambiente non locale e non globale
 - procedure

Attenzione: C

- Algol, Pascal, Ada, Java permettono di annidare blocchi di sottoprogrammi
non possibile in C:
 - funzioni definite solo nel blocco più esterno
 - dunque in una funzione l'ambiente è partizionato in locale e globale
 - non si presenta il problema dei non-locali

Questo non vuol dire che la regola di scoping (statico o dinamico) sia indifferente in C !

Significa solo che è semplice determinare dove risolvere un non-locale:

*ogni non-locale viene risolto nell'ambiente globale
(ovvero: ci sono solo locali e globali)*

misultato scope statico ↗
 ≠
misultato scope dinamico

```
int x=10;
void foo () {
    x++;
}
void fie() {
    int x =0;
    foo();
}
main() {
    fie();
    foo();
}
```

Attenzione: C

Una situazione di “scope dinamico” si realizza in C con le “macro”:

```
int x=10;
int next_x(int delta) { return x + delta; }
#define NEXT_X(delta) x+delta
main() {
    int x=5;
    printf("%d, %d\n", next_x(4), NEXT_X(4));
}
```

La definizione di C dice che **define** corrisponde ad una sostituzione testuale.

Ma poteva anche dire:

la semantica di **define** corrisponde ad una chiamata in scoping dinamico e lasciare poi all’implementazione come realizzarla

Determinare l'ambiente

- L'ambiente è dunque determinato da
 - regola di **scope** (statico o dinamico)
 - regole specifiche, p.e.
 - quando è visibile una dichiarazione nel blocco in cui compare?
- discuteremo più avanti
- regole per il **passaggio dei parametri**
 - regole di **binding** (shallow o deep) → *regole di visibilità di funzioni passate come parametri*
 - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

Alcune regole specifiche

- Dov'è visibile una dichiarazione all'interno del blocco in cui essa compare?

- a partire dalla dichiarazione e fino alla fine del blocco

- Java: dichiarazione di una variabile

```
{a=1; // no!
int a;
...
}
```

normalmente No!
per campi e variabili No!
per metodi sì!

coi metodi
lor posso
fare

- sempre (dunque anche *prima*) della dichiarazione

- Java: dichiarazione di un metodo

- permette metodi mutuamente ricorsivi

```
void f(){
    g(); // si
}
void g(){
    f(); // si
}
...
```

E alcuni problemi

- Pascal

- lo scope di una dichiarazione è l'intero blocco dove essa appare - eccetto i buchi
- ogni identificatore deve essere dichiarato prima di essere usato

```
const a = -1;  
  
procedure pippo;  
  const b = a;  
  ...  
  const a = 0;
```

errore di semantica statica
o in alcuni casi **b = -1 !**

analogamente

- *miglior mettere le dichiarazioni all'inizio del blocco*
- Pascal
 - lo scope di una dichiarazione è l'intero blocco dove essa appare - eccetto i buchi
 - ogni identificatore deve essere dichiarato prima di essere usato

```
procedure pippo  
...  
end (*pippo*)  
...  
procedure A;  
...  
procedure B  
begin  
...  
    pippo;  
    end (*B*)  
    ...  
    ?  
procedure pippo;  
begin...end
```

errore di semantica statica

Mutua ricorsione

Mutua ricorsione (funzioni, tipi) in linguaggi dove un nome deve essere dichiarato prima di essere usato?

- rilasciare tale vincolo per funzioni e/o tipi
 - Java per i metodi
 - Pascal per tipi puntatore

Pascal

```
type lista = ^elem; → definisco il tipo elemento  
      elem = record → e lo specifico sotto  
        info : integer;  
        next : lista;  
      end
```

Java

```
{void f(){  
    g();  
}  
void g(){  
    f();  
}
```

- definizioni *incomplete*

Ada

```
type elem; → elemento tipo è stato introdotto  
           come tipo  
type lista is access elem; → puntatore  
                           ad elemento  
type elem is record → qui specifico  
        info: integer; elem  
        next: lista;   (che ho già  
                      definito, e  
                      dico cosa è  
                      fatto (per  
                      specifico))  
      end
```

C

```
struct elem;  
struct elem{  
    int info;  
    elem *next;  
}
```