

## ASTRAZIONE (del codice)

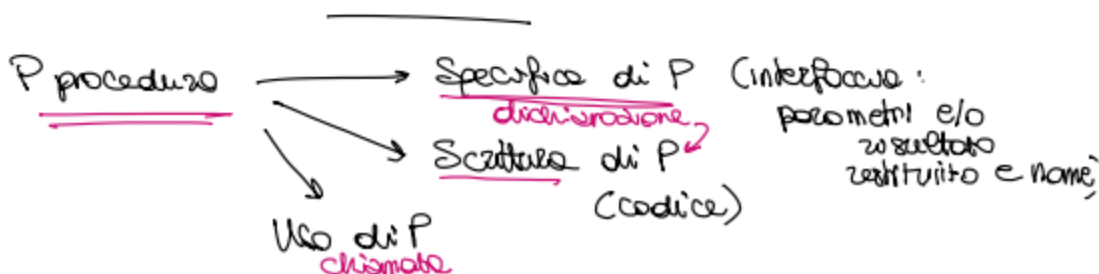
- ↳ ignorare dettagli non rilevanti
- ↳ identificare proprietà di interesse

- variabili = astrazione della cella di memoria
- procedure = astrazione del codice

⇒ All'astrazione si attribuisce un NOME

- Parametri: permettono di rappresentare con un solo codice una funzione dipendente da valori di input  
unico codice = più computazioni possibili
- Corpo: insieme delle istruzioni che vengono eseguite quando si riferisce il nome della procedura

- Una procedura ha una singola entrata
- L'esecuzione del codice chiamante viene sospesa per passare il controllo alla procedura chiamata
- Al termine della procedura il controllo torna sempre al chiamante che riprende l'esecuzione (processo automatico)



Abbiamo bisogno di strumenti  
Sintattici per fare questi passaggi

parametri formali (tipo e nome)

specific → double P (int x) {  
↑  
informazioni  
necessarie  
per chiamare  
e usare  
correttamente P

tipo del  
val. di  
 ritorno

double z;  
/\* corpo \*/  
return exp;

Amh locale  
definito  
nella procedura

→ sono porte  
dell'ambiente  
locale di P

corpo della procedura

di dichiarazione

Uso di P → chiamata

main ...

double y;

:

y = P(3);

parametri attuali

main

:

int x

:

P(x+1);

Q(-)

void P (int y) {

:

z = x + y;

{

non definita dentro P

⇒ Quale valore  
usare per x?

Regole di visibilità sono necessarie per risolvere  
i riferimenti non locali (non definiti nella procedura  
e non parametri formali)

⇒ rimangono le stesse viste: Ogni dichiarazione è  
visibile nel blocco in cui è definita e nei  
blocchi annidati che non la sovrascrivono

statico

Annidamento  
della definizione  
del blocco

dinamico

Annidamento  
della chiamata  
e quindi dell'esecuzione

⇒ Dove si trova  
la dichiarazione

Ambiente riferimento



regole di scope  
statico

⇒ Dove si trova  
la chiamata

Ambiente riferimento



regole di  
scoping dinamico

⇒ Regole per l'ambiente di riferimento



Regole di scope

```
int x = 10;
```

```
void foo () {  
  x++;  
}
```

```
void fie () {
```

```
  int x = 0;
```

```
  foo ();
```

```
  fie ();  
}
```

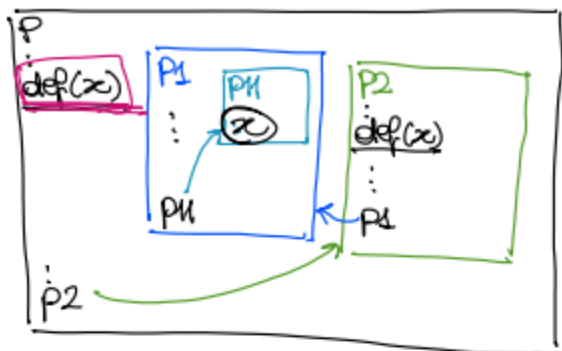
Ambiente che contiene  
la definizione  
(statico)

l'ambiente non dipende  
dalla chiam.  
x non locale  
dove trova significato?

Ambiente che contiene  
la chiamata  
(dinamico)

L'ambiente di riferimento  
comba (potenzialmente)  
per qui chiamato

# SCOPING STATICO



Sequenza di nomi

$P \rightarrow P_2 \rightarrow P_1 \rightarrow P_{11}$

Scheme definizioni



catena statica

```

int x = 0;
void pippo (int m) {
    x = m + 1;
}

pippo(3);
4 ← write(x);

int x = 0;
pippo(2);
write(x);
3 ← write(x);
    
```

*Annotations:*  
 - `x` non locale (pointing to the `x` in the function scope)  
 - `4` ← `write(x)`: `4` is the value of `x` at the call site.  
 - `3` ← `write(x)`: `3` is the value of `x` at the call site.  
 - `write(x)` in the function scope: `x` is global, modified only once.  
 - `write(x)` in the global scope: `x` is global, modified only once.  
 - `write(x)` in the function scope: `x` is global, modified only once.  
 - `write(x)` in the function scope: `x` is global, modified only once.

⇒ Scoping statico  
 ovvero `x` dentro `pippo`  
 fa sempre riferimento  
 alla `x` globale

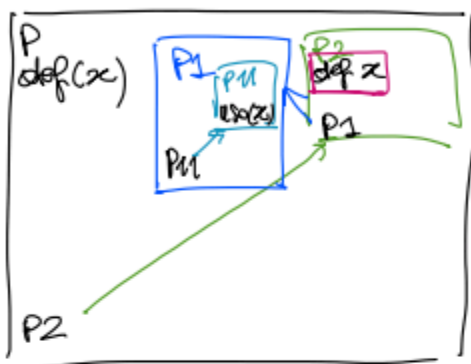
Nello scoping statico l'ambiente di riferimento  
 è indipendente dalla posizione della chiamata

NO:

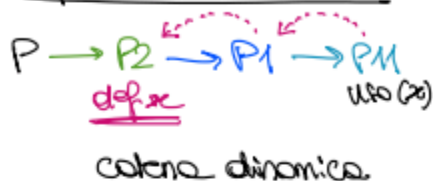
`x = 1;`  
`int x;`

SI: `void f();`  
`f();`

## SCOPING DINAMICO



Sequenza di chiamate



```
{ int x=0;
```

```
void pippo (int m) {
    x = m+1;
}
```

• pippo(3); porta la x globale a 4

4 ← write(x);

```
{ int x=0;
  → pippo(2);
  write(x);
}
```

porta la x locale a 3

4 ← write(x);