

Variabili

- LP imperativi = astrazione di macchina di v.N. $\left\langle \begin{array}{l} \text{memoria} \rightarrow \text{variabile} \\ \text{processore} \end{array} \right\rangle \left\langle \begin{array}{l} \text{intero} \\ M(10, 20, 30) \end{array} \right\rangle$
- Variabile(nome, indirizzo, tipo, valore, lifetime, scope)

Nome

- Scelte di progetto $\left\{ \begin{array}{l} \text{Lunghezza max} \\ \text{Case sensitivity} \\ \text{Vincoli sulle parole speciali} \end{array} \right.$
- Parole speciali $\left\langle \begin{array}{ll} \text{keyword} & \rightarrow \text{parola che è speciale solo in certi contesti} \\ \text{reserved word} & \rightarrow \text{parola non usabile come nome} \\ \text{predefined word} & \rightarrow \text{significato predefinito ma ridefinibile} \end{array} \right.$

Variabili (ii)

- **Keyword:** (FORTRAN)

```
REAL LUNG  
REAL = 3.14
```

```
INTEGER REAL  
REAL INTEGER
```

- **Reserved:** if, then, else, while, repeat, ...

- **Predefined**
 - Ada: tipi built-in $\left\langle \begin{array}{l} \text{INTEGER} \\ \text{FLOAT} \end{array} \right.$
 - Pascal: “identificatori standard”: nomi sottoprogrammi I/O $\left\langle \begin{array}{l} \text{readln} \\ \text{writeln} \end{array} \right.$

Variabili (iii)

Indirizzo

- Associazione $\left\langle \begin{array}{l} \text{nome} \\ \text{indirizzo} \end{array} \right\rangle$ in generale non semplice



stesso nome associato a diversi indirizzi in diversi $\left\langle \begin{array}{l} \text{luoghi (subprog)} \\ \text{tempi (ricorsione)} \end{array} \right\rangle$

- **Alias**: quando \exists più nomi (*alias*) che referenziano lo stesso indirizzo



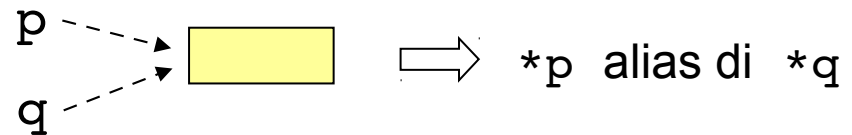
Variabili (iv)

- \exists svariati modi per creare alias

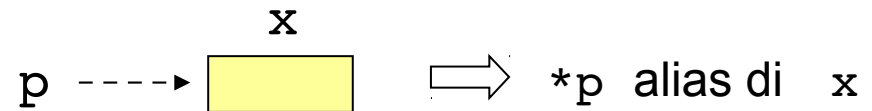
- Ada/Pascal: record con variante

- C/C++ $\left\langle \begin{array}{l} \text{union} \\ \text{puntatori} \end{array} \right.$

```
p = alloc()  
q = p
```



```
p = &x
```



- Reference

- Meccanismo di passaggio dei parametri nei sottoprogrammi

- Alcune giustificazioni nell'uso di alias: obsolete

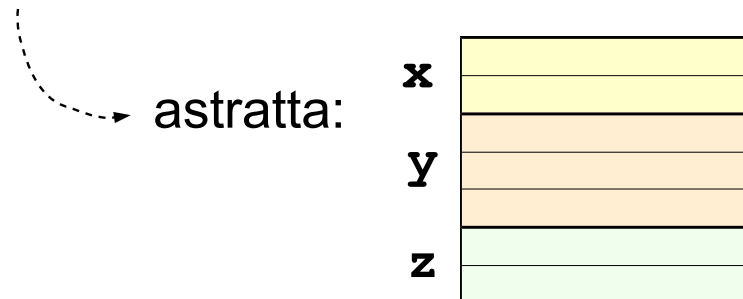
Es: record con variante \rightarrow risparmio di memoria: stessa locazione usata da tipi diversi in tempi diversi
 \rightarrow raggirare le regole sui tipi

Variabili (v)

Tipo $\equiv \{ \text{valori} \} \cup \{ \text{operazioni} \}$

FORTRAN: **INTEGER** $\left\{ \begin{array}{l} \text{valori: } [-32768 \dots 32767] \\ \text{operazioni} \left\{ \begin{array}{l} \text{aritmetiche } (+, -, *, /, **) \\ \text{f di libreria (valore assoluto, ...)} \end{array} \right. \end{array} \right.$

Valore \equiv contenuto della “cella di memoria” associata alla variabile



- Differenza $\left\{ \begin{array}{l} \text{l-value} \\ \text{r-value} \end{array} \right. \rightarrow$ per accedere, bisogna prima determinare l-value

Binding

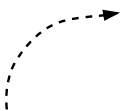
- **Binding** \equiv associazione (entità, attributo) Es: (variabile, indirizzo)
- **Binding time** \equiv istante in cui si realizza il binding

<i>Binding time</i>	<i>Esempio</i>
Progetto del LP	(*, moltiplicazione)
Progetto del compilatore	(INTEGER, {valori}) (FORTRAN)
Compilazione	(variabile, tipo) (C)
Linking	(chiamata a sub di libreria, codice sub)
Loading	(variabile globale, indirizzo)
Esecuzione	(variabile locale, indirizzo)

- Binding $\left\{ \begin{array}{l} \text{statico: se} \\ \text{dinamico: altrimenti} \end{array} \right. \left\{ \begin{array}{l} \text{si realizza prima dell'esecuzione} \\ \text{non cambia durante l'esecuzione} \end{array} \right.$

Binding (*variabile, tipo*)

- Binding(*var, tipo*): necessario prima che la variabile sia referenziata

- Aspetti importanti:
 - Come è def. il tipo di var $\left\{ \begin{array}{l} \text{implicitamente: FORTRAN: inizio} \left\{ \begin{array}{l} \text{I, J, K, L, M} \rightarrow \text{INTEGER} \\ \text{altro} \rightarrow \text{REAL} \end{array} \right. \\ \text{esplicitamente} \end{array} \right.$ 
 - Quando si realizza il binding (*var, tipo*) $\left\{ \begin{array}{l} \text{staticamente} \\ \text{dinamicamente} \end{array} \right.$
- Distinzione $\left\{ \begin{array}{l} \text{dichiarazione} \rightarrow (var, tipo) \\ \text{definizione} \rightarrow (var, \{tipo, indirizzo\}) \end{array} \right.$: unica!

\Rightarrow anche per funzioni $\left\{ \begin{array}{l} (fun, tipo) \\ (fun, \{tipo, codice\}) \end{array} \right.$

Binding (*variabile, tipo*) (ii)

- Binding dinamico (*var, tipo*): si realizza al primo assegnamento

$v := expr$

Vantaggio: flessibilità (Es: programma generico per manipolare liste)

APL: `lista ← 1.5 3.2 8.7` `array [3] of real`
`lista ← 12` `int` \Rightarrow cancellamento del tipo precedente!

Svantaggi: 1. Diminuzione della capacità del compilatore di filtrare errori

`i, x: integer`
`y: array of real` \Rightarrow `i := y` \Rightarrow errore non riconosciuto!
invece di `x` (errore di battitura)

2. Costo $\left\{ \begin{array}{l} \text{tempo} \rightarrow \text{controllo dei tipi a tempo di esecuzione} \\ \text{spazio} \rightarrow \text{descrittore di tipo} \\ \text{complessità allocazione} \rightarrow \text{dimensione flessibile} \end{array} \right.$

- Binding (*var, tipo*) $\left\{ \begin{array}{l} \text{dinamico} \rightarrow \text{interpretazione} \\ \text{statico} \rightarrow \text{compilazione} \end{array} \right. \left\{ \begin{array}{l} \text{difficile cambiare tipo con codice macchina} \\ \text{costo ulteriore "nascosto" di interpretazione} \end{array} \right.$

Binding (*variabile, tipo*) (iii)

Inferenza di tipi: ML (1990) $\left\langle \begin{array}{l} \text{funzionale} \\ \text{imperativo} \end{array} \right\rangle \Rightarrow$ minimizzazione delle dichiarazioni

```
fun area(r) = 3.14 * r * r;
```

 $\left\{ \begin{array}{l} r: \text{reale} \\ \text{risultato: reale} \end{array} \right.$

```
fun molt10(x) = 10 * x;
```

 $\left\{ \begin{array}{l} x: \text{intero} \\ \text{risultato: intero} \end{array} \right.$

```
fun quadrato(x) = x * x;
```

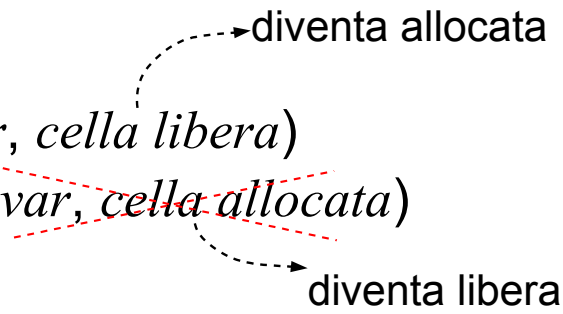
 \Rightarrow **errore!**

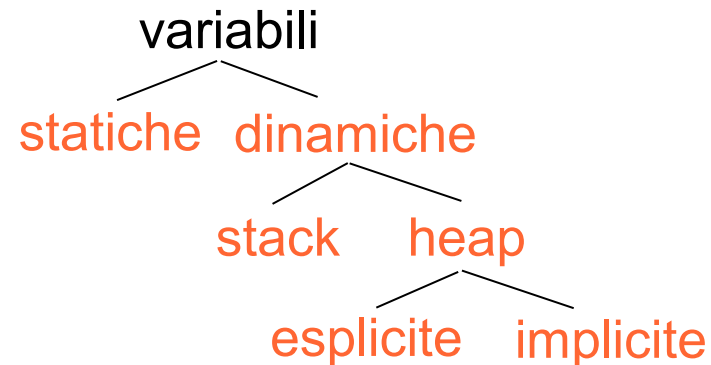
\Downarrow necessità di qualche informazione per il compilatore

```
fun quadrato(x):int = x * x;  
fun quadrato(x: int) = x * x;  
fun quadrato(x) = (x:int) * x;  
fun quadrato(x) = x * (x:int);
```

Lifetime

- **Lifetime** \equiv intervallo temporale in cui “vive” una variabile

- Memoria = {celle libere} \cup {celle allocate} < $\begin{matrix} \text{allocazione: } (var, \text{cella libera}) \\ \text{deallocazione: } (var, \text{cella allocata}) \end{matrix}$

- Classificazione delle variabili in base al loro lifetime:



Variabili Statiche

- Lifetime del binding (*var statica, indirizzo*) = intervallo di esecuzione del programma
- Vantaggi:
 1. Var **globali** accessibili durante tutta l'esecuzione del programma
 2. Var **locali** “sensibili alla storia” → conservazione del valore tra chiamate diverse
 3. Efficienza → \nexists sovraccarico runtime per allocazione/deallocazione
- Svantaggi:
 1. Ridotta flessibilità (\nexists ricorsione)
 2. Non condivisione di memoria da parte di diverse variabili

Es: $\left\{ \begin{array}{l} \text{sub1} \rightarrow \text{A1} \\ \text{sub2} \rightarrow \text{A2} \end{array} \right\}$ grossi array

Variabili Dinamiche Stack

- Binding $\begin{cases} (var, tipo) \rightarrow \text{statico} \\ (var, indirizzo) \rightarrow \text{dinamico (durante l'elaborazione della definizione)} \end{cases}$
- Tipicamente: variabili locali (Pascal, Ada, C, C++, ...)
- Vantaggi: $\begin{cases} \text{condivisione di memoria tra diversi sottoprogrammi} \\ \text{ricorsione} \rightarrow \text{copia } \forall \text{ record di attivazione} \end{cases}$
- Svantaggi: $\begin{cases} \text{overhead di allocazione/deallocazione} \\ \text{insensibilità alla storia} \rightarrow \text{C/C++ : static} \end{cases}$

Variabili Dinamiche Heap Esplicitite

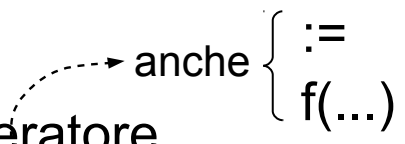

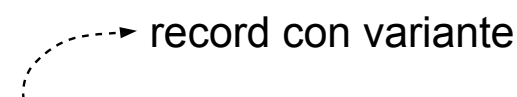
- Celle di memoria anonime allocate/deallocate esplicitamente in esecuzione
- Referenziate mediante variabili $\left\langle \begin{array}{l} \text{puntatore} \\ \text{reference} \end{array} \right\rangle \Rightarrow$ **heap** = collezione non strutturata di celle di memoria
- Creazione $\left\langle \begin{array}{l} \text{operatore (C++)} \\ \text{funzione di sistema (C)} \end{array} \right\rangle \Rightarrow$ binding (*var*, *tipo*): statico
- Distruzione $\left\langle \begin{array}{l} \text{esplicita (C++)} \\ \text{implicita} \rightarrow \text{garbage collection (Java)} \end{array} \right\rangle$

```
int *p;  
...  
p = new int;  
...  
delete p;
```
- Vantaggio: supporto a strutture dati dinamiche (alberi, grafi, ...)
- Svantaggi $\left\langle \begin{array}{l} \text{ridotta ergonomia (puntatori)} \\ \text{costo} \left\langle \begin{array}{l} \text{allocazione} \\ \text{accesso} \\ \text{deallocazione} \end{array} \right\rangle \end{array} \right\rangle$

Variabili Dinamiche Heap Implicite

- Allocate nell'assegnamento → in generale, binding di tutti gli attributi
- Vantaggio: massimo grado di flessibilità → supporto a codice altamente generico
- Svantaggi <
 - sovraccarico in esecuzione (mantenimento degli attributi dinamici)
 - perdita di capacità di individuazione degli errori

Type Checking

- Attività che controlla la compatibilità degli operandi di un operatore 
- Def: Tipo **compatibile** con l'operatore < **legale** per l'operatore
implicitamente convertibile in un tipo legale 
- Def: Errore di tipo \equiv Applicazione di un operatore ad un operando non compatibile
- Se binding $(var, tipo)$ 
 - statico \rightarrow type checking quasi tutto statico \rightarrow ridotta flessibilità
 - dinamico \rightarrow type check. dinamico \rightarrow ridotta capacità di individuazione degli errori

Strong Typing

- Idea nata nel contesto della programmazione strutturata (1970s)
- Def: LP è **fortemente tipato** se permette sempre l'individuazione di errori di tipo



tipi degli operandi sempre noti a tempo di < compilazione
esecuzione

<i>LP</i>	<i>Fortem. tipato?</i>	<i>Perchè</i>
FORTTRAN	no	∄ controllo compatibilità parametri formali/attuali
Pascal	quasi	record con variante
Ada	quasi	UNCHECKED_COVERSION(x): restituisce la stringa di bit di x
C/C++	no	union funzioni senza controllo compatibilità formali/attuali
ML	si	tipi noti staticamente (dichiarazioni + regole inferenziali)

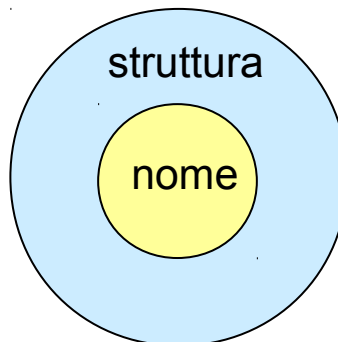
- Regole di coercizione: possono inibire il controllo dei tipi!

Compatibilità dei Tipi

- Influenza la progettazione delle operazioni (espressioni miste)

- Compatibilità $\begin{matrix} \text{type}(v_1) \\ \text{type}(v_2) \end{matrix} \Rightarrow \begin{cases} v_1 := v_2 \\ v_2 := v_1 \end{cases}$

- Compatibilità per $\begin{cases} \text{nome} \rightarrow \text{variabili} \begin{cases} \text{nella stessa dichiarazione} \\ \text{dichiarazioni che usano stesso nome di tipo} \end{cases} \\ \text{struttura} \rightarrow \text{variabili con tipi con struttura identica} \end{cases}$



Compatibilità per Nome

- Caratteristiche < facile da implementare
restrittiva

```
type tipo_indice = 1..100;  
var cont: integer;  
    ind: tipo_indice;
```

} non compatibili → non assegnabili

- Pb: passaggio parametri strutturati → tipo definito un'unica volta globalmente



non possibile definire il tipo in termini locali (come il nome delle variabili!)

Compatibilità per Struttura

- Necessario confrontare l'intera struttura del tipo

- Questioni progettuali

1. Record con $\left\langle \begin{array}{l} \text{stessa struttura} \\ \text{diversi nomi dei campi} \end{array} \right\rangle \Rightarrow ?$

2. Array con $\left\langle \begin{array}{l} \text{stessi} \left\{ \begin{array}{l} \text{tipo di elemento} \\ \text{dimensione} \end{array} \right. \\ \text{diversi range dell'indice} \left\langle \begin{array}{l} 0..9 \\ 1..10 \end{array} \right\rangle \end{array} \right\rangle \Rightarrow ?$

3. Enumerativi con $\left\langle \begin{array}{l} \text{stessa numero di elementi} \\ \text{diversi nomi degli elementi (letterali)} \end{array} \right\rangle \Rightarrow ?$

4. Impossibile differenziare tipi con stessa struttura

```
type celsius = real;  
    fahrenheit = real;
```

Scope

- Def: **Scope** di una variabile $v \equiv \{ \text{segmenti di codice in cui si può referenziare } v \}$

- Variabile
 - locale in
 - unità di programma \Rightarrow dichiarata in essa
 - blocco
 - non locale: visibile ma non dichiarata in essa

- **Regole di scope**: stabiliscono il modo in cui l'occorrenza di un nome è associata ad una variabile

- Scope
 - statico
 - dinamico

Scope Statico

- Introdotto da ALGOL 60: scope creati da unità di programma → gerarchia di scope
- Pb fondamentale: quando il compilatore trova la referenza ad una var
→ determinazione dei suoi attributi → dichiarazione?

```
procedure P;  
  var x: integer;  
  procedure P1;  
  begin  
    ... x ...  
  end;  
  
  procedure P2;  
  var x: integer;  
  begin  
    ...  
  end;  
begin  
  ...  
end;
```

nascosta → $\left\{ \begin{array}{l} \text{Ada : Prog.x} \\ \text{C++ : ::x} \end{array} \right.$ -----> globale

```
program Prog;  
  var x: integer;  
  procedure P1;  
  var x: integer;  
  begin  
    ... x ...  
  end;  
  
begin  
  ...  
end;
```

Scope Dinamico

- Basato sulla sequenza delle chiamate, invece che sulla relazione spaziale

```
procedure P;  
  var x: integer;  
  procedure P1;  
    begin  
    ... x ...  
    end;  
  procedure P2;  
    var x: integer;  
    begin  
    ...  
    end;  
  begin  
    ...  
  end;
```



significato di x in P1: dinamico



ricerca negli antenati dinamici

- Esempi:
1) $P \rightarrow P2 \rightarrow P1$: x trovato in P2
2) $P \rightarrow P1$: x trovato in P

Scope Dinamico (ii)

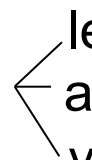
- Problemi:

1. Programmi meno affidabili: var locali di un subprog in exec potenzialmente visibili da qualsiasi altro subprog in exec
2. Impossibile type checking statico (di variabili non locali)
3. Programmi meno leggibili (necessità di seguire la catena di chiamate)
4. Inefficienza nel meccanismo di accesso a variabili non locali

- Vantaggio: Var locali del chiamante implicitamente visibili nel chiamato



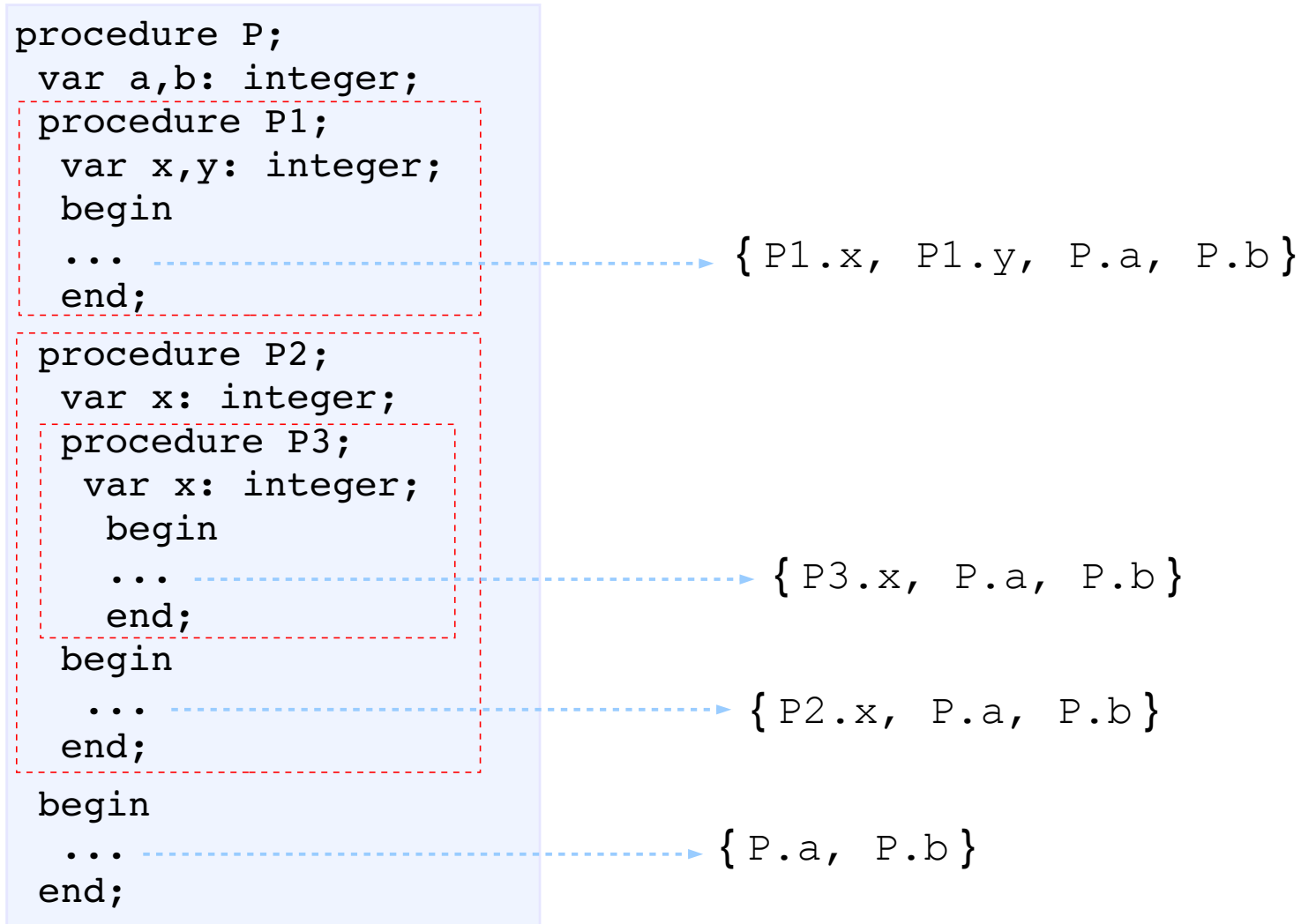
non necessario passaggio dei parametri

Conclusione: Programmi con scope statico: più  leggibili
affidabili
veloci in esecuzione

Ambiente di Referenziazione

Ambiente di referenziazione di una istruzione $\equiv \{ \text{variabili visibili nell'istruzione} \}$

1. LP con scope statico:



Ambiente di Referenziazione (ii)

2. LP con scope **dinamico**: $AR = \{ \text{var locali} \} \cup \{ \text{var degli altri subp antenati attivi} \}$

