LINGUAGGI

Un **linguaggio di programmazione** è una **notazione** per **descrivere** algoritmi e dati.
Un **programma** è una **frase** di un linguaggio di programmazione, una **specifica finita** di un algoritmo. Per ogni algoritmo ho quindi infiniti programmi.
Un PL mi permette di scrivere i passi computazionali necessari per rappresentare un algoritmo. Le proprietà che deve avere un PL sono:

a. **leggibilità**: sintassi chiara, minimo overloading di operatori, parole chiave significative
b. **scrivibilità**: pochi costrutti, interfacce, numero di operazioni
c. **affidabilità**: testing per errori di tipo, gestione delle eccezioni, aliasing
d. **costo**: di scrittura, di formazione, di compilazione, di manutenzione
e. **portabilità**: indipendenza dalla macchina
f. **astrazione**: polimorfismo, astrazione dei dati e del controllo
g. **well-definedness**: semantica chiara

Macchina Astratta (MA): dato un **linguaggio L** di programmazione $\mathbf{M_L}$ la macchina astratta di L, è un **insieme di strutture dati ed algoritmi** che permettono di **memorizzare ed eseguire** programmi di L.
La macchina può quindi essere vista così: M_L= memoria + interprete; è possibile realizzare la MA in HW (++veloce, --flessibile), in FIRMWARE (+veloce,-flessibile) o in SW (-veloce, ++flessibile).
Linguaggio Macchina: data una macchina astratta M, il **linguaggio $\mathbf{L_M}$** che ha come stringhe legali **tutte** quelle **interpretabili** da M è detto **linguaggio macchina**.
Implementare un linguaggio L significa quindi **realizzare** una macchina astratta $\mathbf{M_L}$.

Per realizzare M_L da eseguire sulla macchina astratta Mo_{Lo} (che conosce un altro linguaggio, appunto Lo) ho principalmente due opzioni:

1. **traduzione implicita**: esecuzione di codice Lo dato codice L (mediante una M_L) → interpretato
2. **traduzione esplicita**: traduzione in Lo dato codice L → compilato

NOTAZIONI:
$\mathbf{Prog^L}$ = insieme dei **programmi** scritti in L
D = insieme dei dati in **input e output**
$\mathbf{P^L \in Prog^L}$, $\mathbf{P^L : D \rightarrow D}$ parziale ricorsiva tale che $\mathbf{P^L(Input) = Output}$

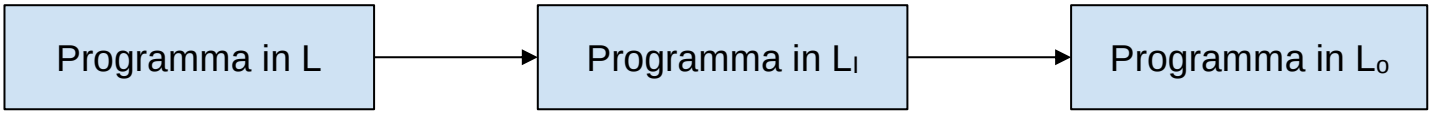
Un **interprete per un linguaggio L**, scritto nel linguaggio Lo, è un **programma** che realizza
$I^{Lo,L}$: ($Prog^L$ * D) → D tale che $\mathbf{I^{Lo,L}(P^L, Input) = P^L(Input)}$

Un **compilatore da un linguaggio L a Lo**, è un **programma** che realizza
$C^{L,Lo}$: $Prog^L$ → $Prog^L$ tale che dato $P^L \in Prog^L$, $\mathbf{C^{L,Lo}(P^L) = P^{Lo}}$ **e** $\mathbf{P^L(Input) = P^{Lo}(Input)}$. La compilazione è composta dalle fasi:

a. **scanner**: analisi lessicale, spezza un programma nei componenti sintattici primitivi, detti tokens
b. **parser**: analisi sintattica, crea un albero della sintassi del programma

INTERPRETE	COMPILATORE
Nessuna traduzione, esecuzione lenta	Lenta traduzione, veloce esecuzione
Scarsa efficienza della macchina M_L	Difficile da implementare
Buona flessibilità e portabilità	Buona efficienza
Buona interazione run - time	Scarsa flessibilità, bassa portabilità

Oltre all'implementazione di un interprete puro e di un compilatore, esiste una terza opzione, per poter trarre vantaggi da entrambi i metodi precedenti, una **soluzione ibrida**. Questa soluzione prevede che venga aggiunta una **macchina intermedia tra M_o e M_L**, denominata **M_I**. Il processo di comprensione viene quindi modificato nel seguente modo: un interprete puro si occupa di trasformare L in L_I. Fatto ciò entra in gioco un compilatore che si occupa di tradurre L_I in un linguaggio comprensibile a M_o (cioè L_o). Questo approccio è **usato da Java**.



SINTASSI

Parola: **stringa** di caratteri su un alfabeto
Frase: **sequenza** ben formata di **parole**
Linguaggio: **insieme** di **frasi**
Lessema (terminale): **unità sintattica** di più basso livello di un linguaggio (es. for, sum, +, *...)
Token (non terminale): **categoria** di lessemi (es. identificatori)

La sintassi è composta da un vocabolario e da delle regole di composizione (grammatica). Un tipo particolare di grammatica è la CFG, inventata da Chomsky (ed equivalente alla BNF).

CFG (context free grammar)

Una grammatica libera dal contesto (**CF**) è una quadrupla **G = <V, T, P, S>** dove:

- **V** è un insieme finito di simboli **non terminali** (elementi della frase)
- **T** è un insieme finito di simboli **terminali** (vocabolario)
- **P** è un insieme finito di **produzioni**, nella forma $A \rightarrow_1 \alpha$ dove
  - $A \in V$
  - $\alpha \in (V \cup T)^*$
- **S** è il **simbolo iniziale** (categoria delle frasi)

Una grammatica CF mi permette di descrivere la sintassi di qualsiasi linguaggio di programmazione.

BNF (Backus-Naur Form)

Progettata nel 1959, prevede:

- **parole** come **simboli terminali**
- i **non terminali** racchiusi tra **parentesi angolari**
- la **non terminalità** di un **simbolo iniziale**

<A> ::= α <B> γ | α
<B> ::= ε | β_1 | β_2

Per semplificare la notazione, si usa la **Extended BNF (EBNF)** che aggiunge:

- **[ ]** indica **0 o 1** occorrenza di quanto contenuto
- **{ }** indica **0 o più** occorrenze di quanto contenuto
- la **,** per esprimere **più opzioni** in or

<A> ::= α [β_1 , β_2] γ | α

La EBNF sarà il modo in cui descriveremo il linguaggio che costruiremo durante questo corso.

SEMANTICA

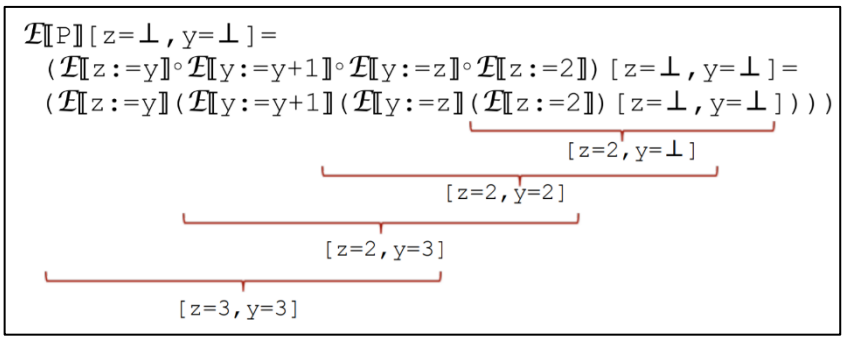
La **semantica** attribuisce un **significato** (entità autonoma) ad ogni **frase sintatticamente corretta**. La semantica ha il compito di **controllare** e di **garantire**: una **corrispondenza** tra il **numero di parametri attuali** e il **numero di parametri formali**, l'**inizializzazione** di un **identificatore** prima dell'uso, la **compatibilità** dei **tipi** di un assegnamento e di **rimuovere** le **ambiguità** grammaticali

A seconda di chi legge la semantica, è utile **descrivere significati diversi**; esistono quindi diversi tipi di semantica, focalizzate su diversi aspetti: **semantica denotazionale** (funzionalità di **I/O**), **semantica operazionale** (**trasformazioni di stato**) e **semantica assiomatica** (**proprietà**).

Semantica denotazionale

La semantica **denotazionale** è basata sulla **teoria della ricorsione** ed è il **modello matematico dei programmi**. Per costruirne una devo

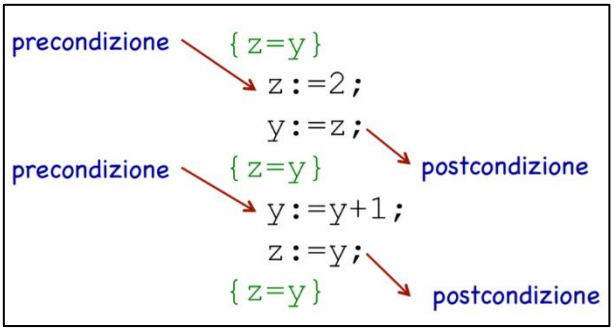
1. definire un oggetto matematico per ogni entità
2. definire una funzione che mappi le istanze delle entità negli oggetti matematici



Quindi **un programma** corrisponde ad **una funzione**, ottenuta tramite **l'equivalenza di funzioni più piccole**; questo permette di utilizzare l'oggetto creato tramite la semantica denotazionale per dimostrare la **correttezza di un programma**. Poco utilizzata per l'**alta complessità**.

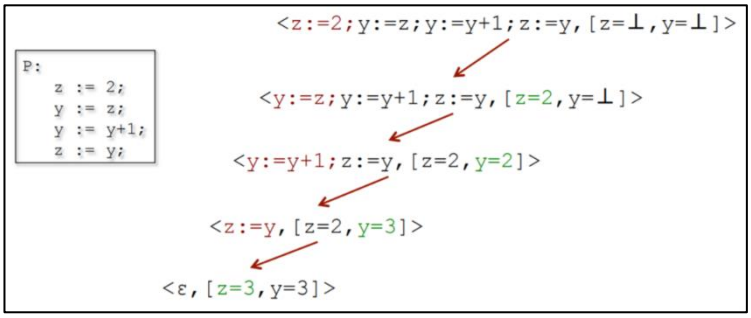
Semantica assiomatica

Basata sulla **logica formale**, la semantica **assiomatica** è stata progettata per fornire una **verifica formale** dei programmi. Consiste in **assiomi** e **regole di inferenza** per ogni comando del linguaggio Le **espressioni logiche** sono chiamate **asserzioni** e definiscono i vincoli (sia **pre-condizioni** che **post-condizioni**).



Semantica Operazionale

La semantica **operazionale** descrive **come** il programma **effettua un'operazione** eseguendola su una macchina reale o simulata; il **cambio di stato** della macchina quindi definisce il **significato del comando**.



Introduciamo ora alcuni concetti: la **composizionalità** è la **proprietà** che implica che il **significato di un programma** debba essere **funzione del significato dei costituenti immediati**. L'**equivalenza** tra programmi è definita dalla seguente caratteristica: due programmi sono equivalenti se la loro **semantica è equivalente**. Questo si traduce in una equivalenza **a livello di funzionalità, esecuzione step-by-step e proprietà**. Con l'equivalenza tra programmi possiamo quindi **valutare** la **correttezza** di un programma e l'**efficienza** di un programma.

CATEGORIE SINTATTICHE

Le **categorie sintattiche** sono gli **elementi non terminali della grammatica classificati** in funzione di cosa **denotano/producono/ottengono rispetto** ad uno **stato di computazione**; ne fanno parte le **espressioni**, i **comandi** e le **dichiarazioni** di un linguaggio. Lo **stato di computazione** è dettato da due fattori, l'**ambiente** (*environment*, l'**insieme dei bindings** tra **identificatori** e **denotazioni**) e la **memoria** (*store*, l'**insieme di effetti sugli identificatori** causati dagli assegnamenti).

Espressioni

Le **espressioni** vanno **valutate** per restituire il **valore che denotano**. Due espressioni **sono equivalenti** se **vengono valutate** nello stesso valore **in tutti gli stati di computazione**.

Comandi

I **comandi** rappresentano **funzioni da memoria a memoria**. Devono essere **eseguiti** per effettuare la **trasformazione della memoria** (che è **irreversibile**). Due comandi **sono equivalenti** se **per ogni stato** della memoria **in input** producono **lo stesso stato** della memoria **in output**.

Dichiarazioni

Le **dichiarazioni** descrivono i **legami tra** gli **identificatori** e le **denotazioni** e servono per **creare/modificare** gli **ambienti**. Devono essere **elaborate** per poter agire **sui legami** (**reversibilmente**). Due dichiarazioni **sono equivalenti** se producono lo **stesso ambiente** e la **stessa memoria** in tutti gli stati di computazione.

SISTEMI DI TRANSIZIONE

Il **sistema di transizione TS** specifica **cosa viene calcolato** per induzione sulla struttura sintattica ed è definito: un sistema di transizione è **una struttura** (Γ, →), dove **Γ** è un **insieme** di elementi **γ** chiamati **configurazioni** e la **relazione binaria** → **⊆ Γ x Γ** è chiamata **relazione di transizione**.

ESPRESSIONI

La **valutazione** di un'**espressione restituisce un valore** esprimibile con un **intero** o con un **booleano**. I **costituenti elementari** sono i **letterali**, **composti** mediante **operatori**.

Espressioni aritmetiche

Sono espressioni **costituite da operatori**, **operandi**, **parentesi** e **chiamate** di procedura. Sono caratterizzate da un **numero di operatori** (*arietà*), delle **regole di precedenza** e di **associatività**, un **ordine di valutazione degli operandi**, una presenza di **side - effects** e un **overloading degli operatori**. Queste espressioni possono essere **denotate in modo diverso** variando la posizione dell'operatore: otteniamo così la notazione **pre - fissa** (*+ a b*), la notazione **post - fissa** (*a b +*) e la notazione **in - fissa** (*a + b*). La notazione **post - fis**sa è in assoluto la **più semplice**, **non necessitando** né di **regole di precedenza**, né di **regole di associatività** e di **parentesi**. Un'espressione scritta in questo modo può essere risolta semplicemente **usando una pila**: *leggo un simbolo e lo metto nella pila. Se il simbolo letto nell'espressione è un operatore, lo applico ai due simboli precedenti in pila*. La notazione **pre - fissa** è molto più semplice di quella in - fissa. Anche lei, come la post - fissa, **non ha bisogno** di **regole di precedenza**, di **associatività** o di **parentesi** e può essere valutata **con una pila**. Nella notazione **in – fissa**, invece, è necessario stabilire le **regole di precedenza** e quelle di **associatività** tra gli operatori. Le **espressioni** vengono internamente **rappresentate tramite un albero**. Da questo albero il compilatore produce il codice intermedio o l'interprete valuta l'espressione.

Valutazione delle espressioni - possibili problemi

I linguaggi di programmazione adottano una **valutazione** degli operandi **lazy** (o *corto circuito*) o **eager**: la prima presuppone che vengano **valutati solo** gli operandi strettamente **necessari**, la seconda che vengano **valutati tutti**. Prendiamo ad esempio l'**espressione** *a == 0 || b/a > 2*: se *a == 0* con una **valutazione lazy** avremo un risultato **vero**; con una valutazione **eager**, invece, otterremo un **possibile errore** perchè valutando *b/a > 2*

dividiamo per 0. Altra situazione da considerare è quella riguardante i **side – effect**, che si ottiene quando una **funzione** riferita in una espressione **cambia un altro parametro della stessa espressione**.

Valutazione ed equivalenza

La funzione $Eval: Exp \to Con$, che descrive il comportamento dinamico delle espressioni restituendo il valore è definita da $Eval(e) = k \leftrightarrow e \to_e^* k$ (**valutazione**)
L'equivalenza di espressioni $\equiv \subseteq Exp \times Exp$ è definita da $e_0 \equiv e_1 \leftrightarrow Eval(e_0) = Eval(e_1)$ (**equivalenza**)

DICHIARAZIONI

Identificatori

Gli **identificatori** sono **sequenze di caratteri** usati per **denotare** o **rappresentare un altro elemento** (come **variabili**, **costanti**, **espressioni** o **metodi**). In un programma gli identificatori sono usati per **riferirsi agli oggetti senza conoscerne il valore** a priori. Non tutte le parole possono essere utilizzate come identificatori: esistono infatti alcune *keyword*, **parole riservate** (dipendenti dal linguaggio) che **non possono essere usate** dall'utente per definire i nomi. Il linguaggio, poi, impone altri **vincoli sui nomi** come la **lunghezza massima**, il **case sensitive** o la **presenza o meno di caratteri speciali**. L'**aliasing** è la possibilità di avere **più nomi** per identificare **lo stesso elemento**. Il **polimorfismo**, invece, è la possibilità per **un nome** di **identificare elementi diversi** in **momenti diversi**.

Binding

Binding significa **associare l'identificatore al suo significato**; deriva dalla matematica dove possiamo trovare i concetti di ***binding occurrences*** (la creazione di binding e quindi l'associazione), ***applied occurrences*** (l'uso del nome per accedere al significato) e ***free occurrences*** (l'uso di un identificatore non definito). Col concetto di binding viene definito anche il concetto di **scope**, ovvero lo **spazio** in cui si può **usare un** determinato **nome** per **rappresentare il significato**.

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{m} a_{ij} \,..\, b_{ij}\right)$$

scope di j

scope di i

A seconda di cosa vogliamo denotare, dobbiamo creare diversi binding. Un **binding** è **statico** se **viene effettuato prima dell'esecuzione** e **rimane invariato per tutta la vita del programma** (es. i *tipi delle variabili*). Un **binding** è **dinamico** se **occorre durante l'esecuzione e può cambiare** (es. *i valori delle variabili*). Il binding può essere effettuato a **tempo di compilazione** (*early binding*) o a **tempo di esecuzione** (*late binding*): l'**early binding** è molto **veloce** da essere eseguito e si basa sull'**uso dello stack**; **non è flessibile**. Il **late binding**, al contrario, è più **lento** da essere eseguito ma è **molto flessibile** e si basa sullo **heap**. La scelta di un determinato tipo di binding è prettamente una **questione semantica**.
Possiamo ora definire altri due concetti, collegati al binding degli identificatori: un **ambiente dinamico** è un elemento dello spazio di funzioni $Env = \cup_{I \subseteq_f Id} Env_I$ dove $Env_I : I \to DVal \cup \{\perp\}$ ha metavariabile ρ
Più semplicemente, un **insieme delle associazioni fra nomi e oggetti denotabili** (valori riferibili ad un identifiactore) esistenti a **run – time** in uno specifico punto del programma ed in uno specifico momento dell'esecuzione. La **dichiarazione** è il meccanismo (implicito o esplicito) col quale si **crea un'associazione** nell'ambiente.

Overload di operatori

Gli **operatori** sono **identificatori globali**; overload significa un **diverso binding** per lo **stesso nome** e nello **stesso scope** (es. *l'operatore + per int e float*). Questo genere di operazione può comportare diversi problemi quali la **difficoltà** del compilatore nel **rilevare** gli **errori sintattici** o la **scarsa leggibilità**.

Identificatori liberi

Un **identificatore** è in **posizione libera** (*free occurrence*) se il **suo uso** (*applied occurrence*) **non è nello scope** di **una definizione**. Un identificatore in posizione libera è detto *identificatore libero*. Un identificatore libero **non ha significato** all'interno di una espressione, **deve essere associato ad un ambiente esterno**. Formalmente, possiamo dire che la funzione $FI : Exp \rightarrow Id$ che ad ogni espressione associa l'insieme degli identificatori liberi in essa contenuti è definita per induzione da:

$FI(k) = \emptyset$

$FI(id) = \{id\}$

$FI(e_0\ bop\ e_1) = FI(e_0) \cup FI(e_1)$

$FI(uop\ e) = FI(e)$

In un linguaggio (con identificatori), **un termine** in cui **non ci sono identificatori liberi** è detto *chiuso*; un termine in cui **non ci sono identificatori** è detto *ground*. Un identificatore che può assumere solo i **valori di un insieme predefinito** è detto **costante** (*non modificabile*). Una **costante** con nome è una **variabile** legata ad un valore solo quando viene legata ad una cella. Questo aumenta di molto la leggibilità.

Tipo

Il **tipo** determina l'**insieme di valori** che **condividono** una certa **proprietà strutturale** e indica l'insieme di valori che un **identificatore può denotare**. I tipi forniscono diversi vantaggi quali un **aumento dell'organizzazione** dell'informazione, un aiuto concreto nell'**individuazione** e nella **prevenzione di errori** e alcune ottimizzazioni.

Type binding

Il **type binding**, ovvero l'**associazione** di un **identificatore** ad un particolare **tipo**, può avvenire in momenti diversi. Un type binding **statico** si dice **esplicito** se **esiste un comando** che permette di **dichiarare il tipo** delle variabili; se invece esiste un **meccanismo di default** che specifica il tipo si dice **implicito**. Un type binding **implicito**, al contrario di uno esplicito, è **molto più scrivibile ma molto meno affidabile**. Il **type binding dinamico**, invece, viene **specificato** attraverso un **comando di assegnamento**. È molto **flessibile** ma ha un **costo molto alto**. Il type binding, quindi, introduce l'**ambiente statico**: un ambiente statico (o di tipi) è un **elemento** dello spazio di funzioni definito da $TEnv = \cup_{I \subseteq_f Id} TEnv_I$ dove $TEnv_I : I \rightarrow DTyp$ ha metavariabile Δ e DTyp è l'insieme dei tipi denotabili

Più semplicemente, l'**ambiente statico associa ogni identificatore al suo tipo**.

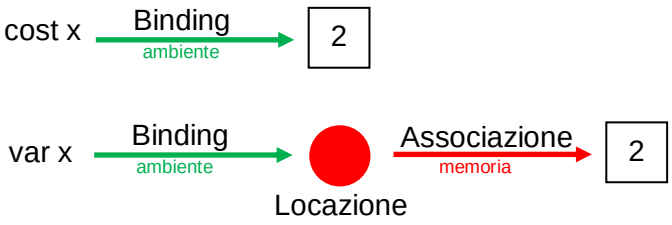
Sia ρ : I un **ambiente dinamico** e Δ : I un ambiente statico con $I \subseteq_f Id$. Gli ambienti ρ e Δ sono compatibili (ρ : Δ) se e soltanto se $\forall id \in I, \Delta(id) = \tau \wedge \rho(id) = \tau$, ovvero **due ambienti sono compatibili se associano all'identificatore lo stesso tipo**. La **semantica statica associa un tipo ad ogni espressione corretta**.

La funzione $Elab : Dic \rightarrow Env$ che descrive il comportamento dinamico delle dichiarazioni a partire da una memoria σ restituendo l'ambiente che esse generano, è definita da $Elab(d) = \rho \leftrightarrow \langle d, \sigma\rangle \rightarrow_d^* \langle \rho, \sigma'\rangle$ (**elaborazione**). L'equivalenza di dichiarazioni, $\equiv \subseteq Dic \times Dic$, è definita da $d_0 \equiv d_1 \leftrightarrow \forall \sigma, Elab(d_0, \sigma) = Elab(d_1, \sigma)$ (**equivalenza**)

	Binding statico	**Binding dinamico**
	Non può processare i dati senza conoscerne il loro tipo	Fornisce più flessibilità alla programmazione: permette di scrivere un programma senza conoscere ancora il tipo delle variabili
	Il compilatore può rilevare gli errori di tipo ma alcuni linguaggi hanno delle regole di traduzione automatica per convertire il tipo a destra (RightHandSide) con il tipo a sinistra (LeftHandSide)	Riduce l'affidabilità: la capacità del compilatore di rilevare errori di tipo è più bassa.
Costo dell'implementazione	Il controllo del tipo è stato eseguito prima dell'esecuzione, quindi non vi è alcun costo per il controllo dei tipi durante l'esecuzione del programma.	Il controllo del tipo deve essere eseguito in fase di esecuzione, il che richiede chela memoria utilizzata per una variabile sia variabile (non so cosa ci andrà dentro)
	Solitamente compilato	Solitamente interpretato
Alcuni linguaggi	C/C++, Java, Pascal, Fortran, BASIC	JavaScript, PHP, Python

COMANDI

I **comandi** sono una categoria sintattica i cui elementi **producono delle trasformazioni irreversibili** di stato (ovvero le **associazioni identificatore – valore**). I costrutti tipici dei paradigmi funzionali e logici sono l'**assegnamento**, la **composizione** e il **controllo di flusso**. L'**assegnamento** è il **comando base** che si limita a **modificare la memoria**; è composto da due parti, il **target** (un *identificatore*) e il **source** (che può essere *composto da identificatori*). Per far funzionare correttamente l'assegnamento introduciamo il concetto di



locazione, ovvero una **fase intermedia dello stato**, un *contenitore per i valori che può essere inutilizzato, indefinito o definito*. La locazione modella l'indirizzo di memoria a cui la variabile è associata, e può essere **creata da una dichiarazione** (*creazione statica*) o **da comandi** (*creazione dinamica*). La **memoria** è una **collezione di associazioni** con identificatori che vengono aggiornati dinamicamente, che ci permette di **catturare** i **cambiamenti non reversibili dello stato** (*in pratica gli identificatori di variabili vengono associati alle rispettive locazioni che ne contengono il valore*).

Variabile

Una **variabile** è un'**astrazione di una cella di memoria**. È caratterizzata da sei elementi: un **nome** (*identificatore*), un **tipo**, uno **scope**, un **indirizzo** (*locazione*), un **valore** ed un **tempo di vita**.

Strutture di controllo

Per rendere il linguaggio di programmazione **potente e flessibile**, dobbiamo introdurre due strutture di controllo, un **comando di selezione** e un **comando di iterazione**.

Comandi di selezione

```
if control_expr
   then clause
   else clause
```

Un **comando di selezione** fornisce lo strumento per **scegliere tra più cammini di esecuzione**. I comandi di selezione si dividono in **selettori a due vie** e **selettori a più vie**. Nela **forma generale** del comando condizionale troviamo la **valutazione di controllo** (nella maggior parte dei casi *booleana altrimenti aritmetica*) e le **clause** (che possono essere *comandi composti*). Possono essere presenti anche varie **parentesi** (*graffe e tonde*) per delimitare meglio sia l'espressione che i comandi. È possibile inserire **un selettore all'interno di un altro**, stando ben attenti ad evitare le ambiguità. Il **selettore a più vie**, permette la **selezione di un comando** (o di un insieme di comandi) **da un gruppo di comandi**. La sua forma varia da linguaggio a linguaggio. Significativo il fatto che **possono essere eseguiti più segmenti** nella stessa esecuzione (*no break*).

```
switch (control_expr){
   case const_expr1: stmt1;
   case const_expr2: stmt2;
   …
   default: stmtx;
}
```

Iterazione

L'**esecuzione ripetuta** di un comando (singolo o composto) si ottiene mediante **iterazione** e **ricorsione**; questi sono i due meccanismi che permettono di **raggiungere la Turing completezza**. Esistono due tipi di iterazione: **indeterminata** (*cicli controllati logicamente*, con il *numero di iterazioni sconosciuto*, come ***while*** e ***repeat***) e

```
for ([expr1]; [expr2]; [expr3]) statement
```

determinata (*cicli controllati numericamente*, con il *numero di iterazioni conosciuto a priori*, come ***do*** e ***for***). A differenza di una iterazione indeterminata, un costrutto **determinato deve sempre terminare**. Una iterazione determinata ha una **variabile di ciclo** e si basa sulla specifica dei **parametri di ciclo**,

```
for index := start to end by step do
```

ovvero di un **valore iniziale**, di uno **finale** e di un **passo d'incremento**; queste variabili **non sono modificabili all'interno del loop**. Quando si

raggiunge il costrutto *for*, si valutano le espressioni di *start* e di *end* e si congelano i valori ottenuti. Si inizializza *index* al valore di inizio e il ciclo termina se *index* > *end* (*step* positivo); altrimenti si esegue il *body* e si incrementa *index*.

Un **altro formato** molto diffuso di scrivere il for, è quello espresso in C; qui **non esiste una variabile esplicita di ciclo** e può essere **modificato tutto** all'interno del ciclo (**anche i parametri**). Inoltre, la **prima espressione** viene **valutata solo una volta**, mentre il **resto** viene valutato **ad ogni iterazione**. L'iterazione **indeterminata**, invece, basa il controllo della ripetizione su un'**espressione booleana** (anche se alcuni linguaggi permettono una condizione aritmetica); dell'iterazione indeterminata esistono **due forme**: la **pre – test** e la **post – test**.

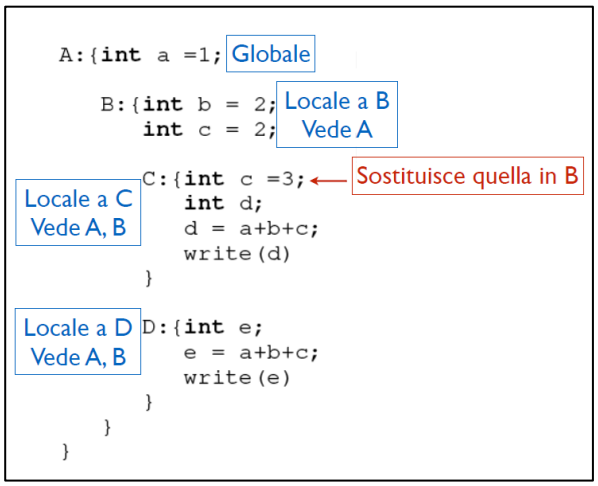
do *body* **while** (*control_expr*)

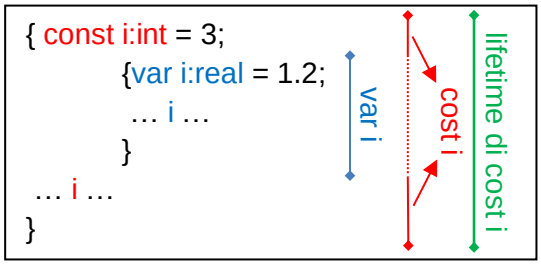
while (*control_expr*) **do** *body*

In alcuni linguaggi (es. C) la forma indeterminata è un **particolare caso della forma determinata**.

Scope

Come abbiamo visto precedentemente, le memorie dipendono dagli ambienti. È necessario quindi **identificare l'ambiente** in cui un **comando viene eseguito**: il blocco. Il blocco è quindi una **parte di programma** che **contiene diversi comandi**; garantisce una **gestione locale dei nomi**, una **ottimizzazione dello spazio di memoria** e la **ricorsione**. È permesso **annidare i blocchi** (ma solo **completamente**, ovvero "inserendoli" come comandi in un altro blocco). Ogni blocco viene **delimitato da** delle **parentesi**; queste, **stabiliscono** anche **lo scope** di una **dichiarazione**, **limitandola al blocco e ai suoi sottoblocchi** (salvo ridefinizioni). Per **scope** ci si riferisce all'**area del codice** nel quale tutte **le occorrenze applicate** di un **identificatore** si riferiscono alla **stessa occorrenza di binding**; questo particolare scope è detto **scope statico** (ovvero il **range** di comandi nel quale **una variabile è visibile**). Dal punto di vista opposto, l'**ambiente di riferimento** di un comando è la **collezione** di tutti i **nomi** che sono **visibili al comando**. Otteniamo una **distinzione** in **locali** e **non locali** per **variabili** e **ambienti**. Le variabili locali sono **dichiarate** nel **blocco in esame**, le non locali sono **dichiarate** in un **blocco** che contiene quello in analisi; le variabili **globali**, ovvero **visibili** in **tutto** il programma, sono una **sottocategoria delle non locali**. Un **ambiente locale** è l'**insieme** delle **associazioni create all'ingresso del blocco** (quindi contenente le variabili locali), mentre un **ambiente non locale** è l'**insieme delle associazioni ereditate da altri blocchi**. Esiste anche in questo caso l'**ambiente globale**, un ambiente **non locale** relativo alle **associazioni comuni in tutti i blocchi**. Un altro concetto introdotto dallo scope è quello di **tempo di vita** (*lifetime*): il **tempo di esecuzione** nel quale tutte **le occorrenze** applicate di un identificatore si **riferiscono alla stessa locazione di memoria**. Scope e lifetime sono **concetti simili** ma di natura diversa: mentre lo **scope** statico è un **concetto spaziale** e viene **definito a tempo di compilazione**, il lifetime è un **concetto temporale** e viene **definito a tempo di esecuzione**.

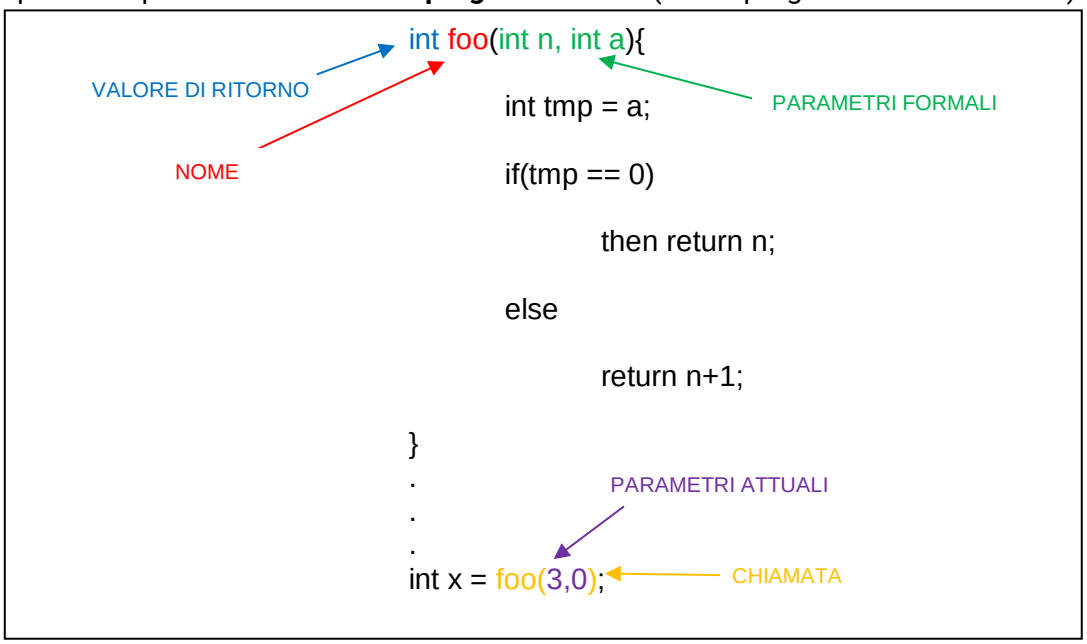




Sottoprogrammi

Per poter **abbreviare** la scrittura di programmi che contengono **sequenze di codice ripetute** ma operanti **su dati diversi**, possiamo **legare** negli ambienti una determinata **sequenza di comandi** (*corpo*) ad un **identificatore** (*nome*) a cui si potrà fare riferimento; una volta **richiamato l'identificatore** passando gli opportuni **dati** (*parametri*), il programma **chiamante viene messo in pausa** e il **corpo** associato al nome **viene eseguito**. Alla **fine** dell'esecuzione del corpo, il **controllo torna al chiamante**. Questa particolare astrazione dà quindi vita ai **sottoprogrammi**. Oltre che ad abbreviare la scrittura del codice, un sottoprogramma permette anche di **nascondere** i **dettagli implementativi**, mostrando solo **nome** e **parametri** (*interfaccia dell'astrazione*). I sottoprogrammi si dividono in **due categorie**: le **procedure** e le **funzioni**. Le **procedure non restituiscono**

nessun valore al **contrario delle funzioni**. I **sottoprogrammi** sono considerati **blocchi all'interno del codice**, per cui sottostanno alle **regole di scoping** descritte sopra. Un **parametro formale** è una **variabile** listata nella definizione e che **verrà poi utilizzata nel sottoprogramma**. Il **parametro attuale**, invece, rappresenta il **valore o** un **indirizzo usato** nel comando di chiamata. L'**ambiente di riferimento** di un sottoprogramma, con **scoping statico** è composto dalle **variabili locali** e dalle **variabili dei blocchi esterni**; nello **scoping dinamico**, dalle **variabili locali** più tutte quelle **visibili nei sottoprogrammi attivi** (i sottoprogrammi non terminati).



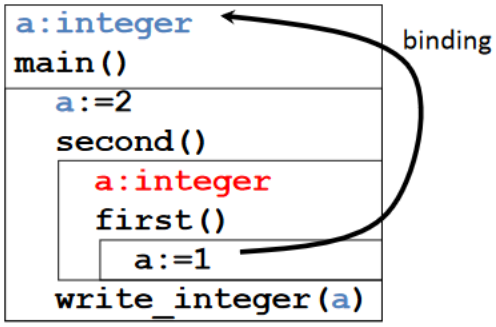
Tipi di scope

Esistono **due tipi di scope**: scope **statico** e scope **dinamico**. Lo **scope statico**, come precedentemente detto, permette che un **identificatore non locale sia risolto nel blocco che testualmente lo racchiude**. Questa tipologia di scoping è eseguita a **compile – time** e per questo è **molto efficiente**, anche se più **complessa da implementare**. Molto importante è ricordare che la **posizione di una chiamata non** ne **determina l'ambiente di riferimento** (che viene definito dal blocco); un metodo per evitare di sbagliare è **utilizzare sempre l'ambiente valido al momento della definizione della funzione interessata**. Lo **scope dinamico**, al contrario, consente che un **nome non locale sia risolto nella chiamata più recente attiva**. In pratica, le **referenze delle variabili** sono **risolte** (collegate) alle dichiarazioni **cercando all'indietro nella catena delle chiamate** i sottoprogrammi che hanno portato a quel punto l'esecuzione. Questo scope è molto **semplice da implementare** ma **diminuisce** sensibilmente la **leggibilità** del programma. Per utilizzarlo sempre nel modo corretto, bisogna **valutare** sempre l'**ambiente valido al momento della chiamata della funzione interessata**. La **differenza** tra scope statico e dinamico è **visibile solo in presenza congiunta** di **ambienti locali** e **non locali** e di **procedure**.

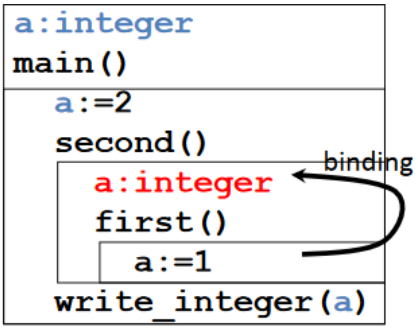
```
{ //scoping statico
  int x = 0;
  void pippo(int n){
        x = n + 1;
  }
  pippo(3);
  write(x); → 4
        { int x = 0;
          pippo(3);
          write(x); → 0
        }
  write(x); → 4
}
```

```
{ //scoping dinamico
  int x = 0;
  void pippo(int n){
        x = n + 1;
  }
  pippo(3);
  write(x); → 4
        { int x = 0;
          pippo(3);
          write(x); → 4
        }
  write(x); → 4
}
```

```
a:integer
procedure first(){
  a:=1}
procedure second(){
  a:integer
  first()}
procedure main(){
  a:=2
  second()
  write_integer(a)}
```



Stampa 1



Stampa 2

Allocazione della memoria

L'uso delle procedure implica un'**allocazione della memoria** particolare: è infatti necessario **salvare i dati relativi al sottoprogramma** quali le **informazioni di sistema** (registri e info di debug), gli **indirizzi di ritorno**, i **parametri**, le **variabili locali** e i **risultati intermedi**. I metodi di allocazione della memoria sono **allocazione statica** (memoria allocata a **compile – time**) e **allocazione dinamica** (memoria allocata a **run – time**).

ALLOCAZIONE STATICA	
PRO	Molto veloce Indirizzo degli oggetti fisso
CONTRO	Lifetime più lungo Niente ricorsione
ALLOCA COSA	Variabili globali Variabili locali di sottoprogrammi Costanti determinabili a compile – time Tabelle di supporto
ALLOCA DOVE	In zone di memoria protette

L'allocazione **statica** consiste nell'**allocare a monte** (compile – time) lo **spazio per gli identificatori**. Come conseguenze, avremo un **lifetime** della durata dell'**intero programma** e **nessuna possibilità** di avere la **ricorsione** (non so a priori quanta memoria dovrò allocare) affiancati da un'**esecuzione molto veloce**. L'allocazione statica viene usata per allocare **variabili globali**, variabili **locali di sottoprogrammi**, **costanti** determinabili a compile – time e **tabelle di supporto** a run – time (type checking, garbage collection…) in **zone di memoria protette**. L'**allocazione dinamica**, invece, viene implementata tramite l'utilizzo di una **pila** su cui vengono salvati, **per ogni istanza di un sottoprogramma**, i **record di attivazione** (*RdA*), ovvero le **informazioni** relative a tale istanza. Analogamente, **ogni blocco** ha un suo **record di attivazione**. Ogni RdA ha un **indirizzo** che non è noto compile – time e che **viene puntato dal RdA successivo**. I campi di un RdA possono essere acceduti aggiungendo un **offset** all'indirizzo del **puntatore**; l'offset specifico per accedere alle variabili locali è chiamato *local_offset* e può essere determinato a compile – time. Il *puntatore RdA* (o **SP**) punta al RdA del **blocco attivo** (quindi **in cima** alla pila). Si forma così la **catena dinamica** (*o catena delle chiamate*), la lista di **link dinamici nello stack in un determinato punto dell'esecuzione**. Poter richiamare un sottoprogramma implica una serie di azioni da eseguire per gestire il cambio di blocco. Il chiamante deve:

1. Creare un'**istanza** del RdA
2. Salvare il proprio **status corrente** dell'esecuzione
3. Passare i **parametri**
4. Passare l'**indirizzo di ritorno**
5. Trasferire il **controllo**

Il sottoprogramma chiamato, terminata la propria esecuzione, deve:

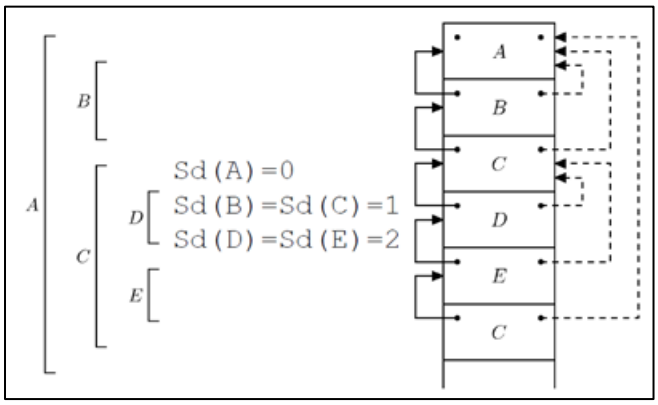
1. I parametri per valore e per valore – risultato devono **restituire** il loro valore
2. **Deallocazione dello stack** per le variabili non locali
3. **Recupero dello stato** di esecuzione del chiamante
4. Ritorno del **controllo al chiamante**

Puntatore di Catena Dinamica
Variabili Locali
Parametri
Indirizzo di ritorno

Struttura di un RDA

Implementazione dello scoping statico

Nello **scoping statico** il metodo più comune di **implementazione** per supportare la **nidificazione dei sottoprogrammi** utilizza i **RdA** ed è la **catena statica**; la catena statica (**CS**) aggiunge un **puntatore al RdA** ed è una catena che **connette il record di attivazione all'istanza del record di attivazione del blocco padre**. Mentre il link dinamico (e quindi la catena dinamica) dipende dalla sequenza di esecuzione del programma, la catena statica **dipende dall'annidamento statico delle dichiarazioni dei sottoprogrammi**. Grazie a questa catena, possiamo definire il concetto di **profondità statica** (**Sd**) il cui valore è la **profondità di annidamento** di quello scope. Il **link del chiamato** viene determinato dal chiamante: quando **Ch chiama P**, il valore sarà definito da **k = Sd(Ch) – Sd(P) + 1**. Se **k = 0**, Ch passa a P il **proprio puntatore**; se **k > 0**, Ch **risale la catena statica di k passi** e consegna il puntatore trovato a P. La continua lettura della catena statica comporta un **elevato costo** che è possibile **ridurre** ad una costante utilizzando il **display**. Il display non è altro che un **array rappresentante la CS**, dove l'**i-esimo valore** equivale al **puntatore dell'RdA del sottoprogramma di livello di annidamento i**. Se il sottoprogramma corrente si trova al livello *i*, un oggetto in uno scope esterno di *h* livelli può essere trovato semplicemente facendo *j = i – h*.

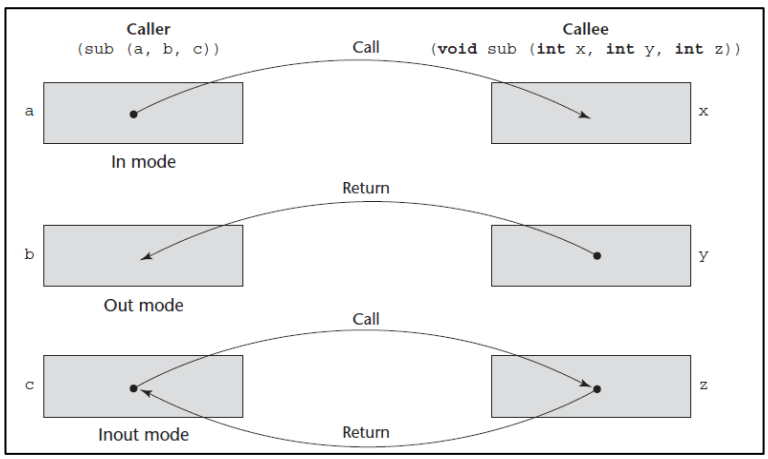


Implementazione dello scoping dinamico

Lo **scoping dinamico**, come già visto, viene gestito seguendo una semplice regola: l'**associazione corrente per un nome è quella determinata per ultima nell'esecuzione** (e non ancora distrutta). Seguendo questo ragionamento, l'implementazione negli RdA prevede che all'interno di ogni blocco sia presente **un campo** dove vengono **memorizzati** i nomi degli **identificatori appartenenti al blocco**. In caso di chiamata di un particolare identificatore, viene **ricercato per nome risalendo la pila**. Sulla base di questo, possiamo definire due diverse implementazioni: **deep access** e **shallow access**. L'idea di deep access è di **mantenre uno stack** con le **variabili attive**, **shallow access** una **memoria centralizzata** con uno slot per variabile. Deep access prevede che le **variabili non locali** vengano trovate **cercando negli RdA** lungo la catena dinamica: se le variabili locali sono inserite nei record di attivazione, le variabili non locali possono essere trovate cercando tra i RdA degli altri sottoprogrammi attivi (partendo dal basso). Per poter velocizzare il meccanismo di ricerca, i **binding** possono essere **memorizzati in una struttura apposita**, gestita come una pila (LIFO), chiamata **A – list** (*association – list*). Lo shallow access, invece, utilizza la **CRT**. La CRT (*central reference tables*) è una **tabella** che mantiene una **distinct di tutti i nomi**; ad ogni elemento di questa distinct è **associata una lista**: il **primo valore** contenuto è il **più recente**. Usare una CRT invece che una A – list riduce il **tempo di accesso medio** da lineare a costante e permette di occupare **meno memoria**. Una CRT però è **più complessa** da gestire rispetto ad una A – list.

Parametri

Come precedentemente visto, i parametri servono per **usare la stessa computazione in contesti differenti** e si dividono in **formali e attuali**; il binding tra queste due tipologie può essere **posizionale** o **keyword**. In una corrispondenza posizionale, il binding tra parametri attuali e formali è **dato dalla posizione**. In una corrispondenza keyword, invece, il parametro attuale **specifica a quale formale si riferisce**. Questo **non necessita** di un **passaggio ordinato** di parametri, ma costringe l'utente a **conoscere il nome dei formali**. Nella maggior parte dei linguaggi, i parametri sono passati in modo posizionale; l'**implementazione** di un



passaggio di parametro può essere effettuata in diversi modi e si basa (principalmente) sulla scelta di **spostare fisicamente un valore** o di **spostare un accesso al valore**.

Il **passaggio per valore**, implementando la **pragmatica in – mode**, prevede che il **valore del parametro attuale** sia usato per **inizializzare il parametro formale**, che viene gestito come una **variabile locale**. Questo significa che le **modifiche al formale non passano all'attuale**. **Molto semplice** da implementare, il suo più grande svantaggio è che si va copiare il valore trasmesso, **aumentando l'utilizzo di memoria**. Alternativamente, è possibile fornire al sottoprogramma chiamato un **path per accedere al valore**, ma bisogna implementare anche una **protezione sulla scrittura della variabile**.

```
void foo (int x) {
   x = x + 1; //poi distrutto
}
…
y = 1;
foo(y+1); //foo(1+1)
```

Il **passaggio per risultato** implementa la **pragmatica out – mode**; il parametro **formale** funziona come **variabile locale** ma, appena prima della fine del sottoprogramma, viene **trasmesso al parametro attuale del chiamante**. Questo metodo ha gli **stessi vantaggi e svantaggi del passaggio per valore**, a cui si aggiungono un **utilizzo ancora maggiore della memoria** (ho una copia extra) ed una **possibile collisione di parametri**: supponiamo che un metodo chiamante passi due volte lo **stesso parametro attuale** e che questo venga salvato con **due nomi diversi** nel sottoprogramma. Se i due parametri formali **vengono modificati** in modo diverso, a seconda dell'**ordine di assegnamento**, avremo un valore o un altro. Altro problema presente è la **tempistica di valutazione degli indirizzi**: quando viene valutato un parametro, **all'inizio o poco prima della terminazione** del metodo?

```
void fixer(int x, int y) {
  x = 17;
  y = 35;
}
…
f.fixer(a,a); //17 o 35?
```

```
void doIt(int x, int index) {
  x = 17;
  index = 42;
}
…
sub = 21;
f.doIt(list[sub],sub);
//17 in posizione 21 o in posizione 42?
```

```
void foo (int x) {
   x = 8; //poi distrutto
}
…
y = 1;
foo(y); //qui y = 8
```

Un altro modo per passare i parametri è implementato seguendo la **pragmatica inout mode** ed è il **passaggio valore – risultato**. Questa tecnica, chiamata anche *passaggio per copia*, è una **combinazione delle precedenti** implementazioni e quindi prende **pregi e difetti di entrambi**; i **parametri locali** vengono usati per **inizializzare il corrispondente parametro formale**, che **funziona come una variabile locale**. Alla terminazione del sottoprogramma i **parametri vengono restituiti al parametro attuale**.

```
void foo (int x) {
   x = x + 1; //poi distrutto
}
…
y = 8;
foo(y); //y = 9
```

Sempre usando una **pragmatica inout mode**, possiamo implementare il metodo di **passaggio per riferimento** (*chiamato passaggio per condivisione*). Questo consiste nel fornire al sottoprogramma il **puntatore alla cella** contenente il valore. I vantaggi principali sono la **velocità** e il **poco spazio richiesto** per il passaggio di indirizzi. Gli svantaggi, invece, sono un **accesso lento ai parametri formali**, **potenziali collisioni** (modifiche al valore del parametro attuale) e rischio di **creazione di alias**.

```
void foo (int x) {
   x = x + 1; //poi distrutto
   //il legame tra x e y
}
…
y = 1;
foo(y); //y = 2
```

L'ultimo metodo di passaggio dei parametri è il **passaggio per nome**. In questo metodo i **parametri attuali** sono **sostituiti ai parametri formali** in **tutte le occorrenze** del sottoprogramma. Fornisce un'**ottima flessibilità** nel late binding ma richiede che l'**ambiente di riferimento del chiamante sia passato** (per calcolare gli indirizzi).

Funzioni come parametri

Alcuni linguaggi permettono di **passare funzioni come argomenti** di sottoprogrammi e di **restituire funzioni come risultato** di procedure (*lambda expression*). In entrambi i casi è necessario **gestire l'ambiente della funzione**: se la funzione viene passata come argomento, occorre avere un **puntatore al RdA**; nel caso la funzione sia **ritornata da una chiamata** di procedura, è necessario **mantenere il RdA della funzione restituita**. Il riferimento che si crea tra **nome** (parametro formale) **e procedura** (parametro attuale) può essere gestito con diverse **politiche di binding**.

Ricorsione

Un modo alternativo all'iterazione per ottenere il **potere espressivo delle MdT** è la **ricorsione**, ovvero una **funzione che è definita in termini di sé stessa**. La ricorsione è **applicabile in ogni linguaggio** che permetta delle **funzioni che si possono chiamare da sole** e una **gestione dinamica della pila**. Essendo un'alternativa all'iterazione, la ricorsione può sempre essere **tradotta in un ciclo** (e **viceversa**). Una tecnica ricorsiva comune è la **ricorsione in coda**; la ricorsione in coda (*tail recursion*) si verifica quando, in una procedura e/o funzione ricorsiva, la **chiamata ricorsiva viene operata come ultimo passo**. Una ricorsione in coda ha l'enorme vantaggio che **non necessita di un'allocazione dinamica della memoria**, ma basta un **singolo RdA**; questo la rende **estremamente efficiente**.

```
int fatt (int n, int res) {
  if (n <= 1 )
    return 1;
  else
    return fatt (n-1, n * res);
} // elimino RdA, ho il val in res
```

Strutture dati

Un **tipo di dato** definisce una **collezione di oggetti** e un **insieme di operazioni** applicabili su di essi. Un **oggetto** rappresenta una **istanza di un tipo di dato** astratto **definito dall'utente**. I dati "base" sono i **tipi scalari**: **booleani**, **caratteri**, **interi**, **reali** e il tipo **void**. Quest'ultimo ha **un solo valore** e **non è possibile** effettuare **nessuna operazione** su di lui. Viene implementato solamente per **definire le istruzioni che modificano lo stato senza restituire alcun valore**: il tipo di queste funzioni, appunto, è void. Esiste anche una famiglia di **dati composti** (o *non scalari*); di questa ne fanno parte i **record**, i **record varianti**, gli **array** (di cui fanno parte le **stringhe**), gli **insiemi** e i **puntatori**. I **record sono collezioni di campi di tipi diversi**; vengono usati per **manipolare** in modo unitario **dati di tipo eterogeneo**. Non sono presenti in Java (esistono le **classi**) ma sono presenti in C (le **struct**). Una volta creato un record, è possibile **accedere ad ogni suo singolo campo**. È possibile avere una **nidificazione di record**. Un record può essere **memorizzato in due modi**: salvando tutti i campi in **modo sequenziale** (**spreco di memoria ma facile accesso ai dati**) o **non sequenziale** (*packed records*, **risparmio di memoria ma accesso costoso**). I **record varianti** sono particolari record dove **solo alcuni campi sono attivi in un determinato istante** (campi alternativi tra loro). In questo caso, i **campi alternativi** possono **condividere** la stessa **locazione di memoria**. Gli **array** sono una **collezione** dove **ogni elemento è identificato da un indice intero**. Esistono in tutti i linguaggi, anche in più versioni (**mono o multidimensionale**). La principale operazione permessa sugli array è la **selezione di un elemento** (la modifica non è un'operazione sull'array ma sulla locazione di un elemento); in alcuni linguaggi esiste anche l'**operazione di slicing** (letteralmente *affettare*) che permette di **prendere parti contigue di un array**. La **memorizzazione** degli array può essere effettuata in diversi modi: utilizzando **locazioni contigue** (con **elementi salvati in ordine di riga o di colonna**) o **discontinue** (tramite un algoritmo per il **calcolo degli indirizzi**). La **forma** (*shape*) degli array indica il **numero delle dimensioni e l'intervallo dell'indice della lista**; può essere fissata **dinamicamente**, **staticamente** o al momento dell'**elaborazione della dichiarazione**. I **puntatori** sono dei **riferimenti ad oggetti** di un altro tipo e permettono di **riferirsi** (e **modificare**) un **valore senza dereferenziarlo**. Un puntatore può essere **creato**, **dereferenziato** (accesso al dato, *p) e può essere **confrontato**.

Equivalenza

Due tipi T e S sono **equivalenti** se **ogni oggetto di tipo T è anche oggetto di tipo S**, e **viceversa**. In molti linguaggi, l'equivalenza tra tipi viene **implementata** tramite una **equivalenza per nome** (se hanno lo stesso nome, sono equivalenti). Un altro possibile metodo per gestire l'equivalenza tra tipi è l'**equivalenza strutturale**: l'equivalenza strutturale tra tipi è la relazione d'equivalenza che **soddisfa le proprietà**:

- un **nome** di tipo è **equivalente a sé stesso**
- se un **tipo T** è introdotto con una definizione **type T = expr**, allora **T è equivalente a expr**
- se due tipi sono costruiti applicando lo **stesso costruttore di tipo** allora **sono equivalenti**

In breve, due tipi sono strutturalmente equivalenti se la loro struttura è equivalente.

Compatibilità e conversione

T è compatibilie con S quando gli **oggetti di T** possono essere **usati in un contesto** dove ci si attende **valori S**. Questa definizione dipende molto dal linguaggio, infatti può essere che: **T e S sono equivalenti**, i **valori di T sono un sottoinsieme dei valori di S**, tutte le **operazioni sui valori di S sono possibili anche su T** (*estensione*), i **valori di T corrispondono ad alcuni valori di S**, i **valori di T possono essere fatti corrispondere ad alcuni valori di S**; proprio quest'ultima possibilità ci introduce nella **conversione di tipi**. Una conversione **narrowing** (*ristretta*) consiste nel **convertire un oggetto ad un tipo che non include tutti i valori del tipo originale** (*float to int*). Al contrario, una conversione **widening** (*estesa*), implica la **conversione di un oggetto ad un tipo che include almeno tutti i valori del tipo originale** (*int to float*). Quindi anche se **T è compatibile con S**, **occorre una conversione**; la conversione può essere **implicita** (*coercizione*) o **esplicita** (*cast*). Una conversione è **implicita** se la **macchina astratta** inserisce la conversione **senza** che ve ne sia **traccia a livello linguistico**; può essere usata se i due tipi hanno gli **stessi valori** e la **stessa rappresentazione** (*tipi uguali, nomi diversi*), se hanno **valori diversi** ma **stessa rappresentazione nell'intersezione** (*intervalli e interi*) o se hanno **valori e rappresentazione diversi** (*interi e reali*). Una conversione **esplicita** consiste nell'**inserire esplicite conversioni di tipo**; è presente in C e in Java. **Non tutte le conversioni esplicite sono consentite**.

Astrazione dei dati

Un'**astrazione** è una **rappresentazione di un'entità** che include solo gli **attributi più significativi**. L'astrazione sui dati ha il compito di **nascondere** le decisioni di **rappresentazione delle strutture dati** e sull'**implementazione delle operazioni**. Da queste motivazioni sono nati i **tipi di dati astratti** dati **definiti dall'utente** la cui rappresentazione è nascosta e le cui dichiarazioni del tipo e dei protocolli delle operazioni sono **contenute in un'unica unità sintattica** (*capsula*). Avere la rappresentazione non visibile **aumenta la leggibilità** e **diminuisce complessità** di scrittura e **conflitti di nome** di variabili. Avere le dichiarazioni di tipo e le operazioni in un'unica unità sintattica, invece, favorisce un'**organizzazione del programma**, aumentando la **flessibilità**. Con l'astrazione entra in gioco il concetto di **modularità** che viene attuato tramite **componenti** (unità di programma come **funzioni**, **strutture**, **moduli** …), **interfacce** (**tipi** e **operazioni** definiti in un componente non visibili al di fuori), **specifiche** (funzionamento del componente espresso tramite **proprietà osservabili dall'interfaccia**) e **implementazioni** (**strutture dati** e funzioni definite dentro al componente e non visibili fuori).

COMPONENTE: coda a priorità
INTERFACCIA: **PrioQueue**
empty: PrioQueue
insert: ElemType * PrioQueue PrioQueue
deletemax: PrioQueue ElemType*PrioQueue
SPECIFICA: **insert** aggiunge all'insieme di elementi memorizzati
deletemax restituisce l'elemento a max priorità e la coda degli elementi rimanenti

Per poter definire un **ADT** (*abstract data type*), un linguaggio deve avere un'**unità sintattica** che possa **incapsulare** la **definizione del tipo**, un **metodo** per rendere il **nome del tipo visibile agli utenti** nascondendone la complessità e alcune **operazioni primitive**. Soprattutto l'incapsulamento serve per ottenere l'**indipendenza dalla rappresentazione**: l'obiettivo è che l'utente non riesca a distinguere due implementazioni del dato astratto.