*Roberto Russo*
*Paradigmi di Programmazione*
*2009*

# *Capitolo 1 – Macchine astratte.*

Dato un linguaggio di programmazione *L*, una macchina astratta per questo linguaggio sarà *ML*, un insieme di algoritmi e strutture dati che permetteranno la memorizzazione ed esecuzione di programmi scritti in *L*. Una macchina astratta è composta sostanzialmente da *memoria* ed *interprete.*

L'interprete compie determinate azioni in base al linguaggio che interpreta appunto. Nonostante esistano *infiniti* linguaggi le azioni compiute dall'interprete sono principalmente quattro.

1. Operazioni per l'elaborazione dei dati primitivi.
2. Operazioni di controllo della sequenza.
3. Operazioni e strutture dati per il controllo del trasferimento dati.
4. Operazioni e strutture dati per la gestione della memoria.

I dati primitivi del primo caso sono quei dati che sono direttamente rappresentabili dalla macchina: quindi avremo un insieme di operazioni definite (per esempio sui dati di tipo numerico avremo a disposizione la *somma, moltiplicazione, ecc.*) per l'elaborazione di questi.
L'interprete ha inoltre la possibilità di modificare l'indirizzo della prossima istruzione da eseguire. Può controllare il flusso di operazioni da compiere, e nel caso vi siano dei *salti condizionati* sarà opportuno disporre di operazioni per la manipolazione degli indirizzi di memoria per l'esecuzione della prossima istruzione.
Ovviamente l'interprete avrà a disposizione delle strutture dati atte al trasferimento dei dati (pile e stack) per il trasferimento dei dati dalla memoria all'interprete e viceversa e sarà fornito di alcuni meccanismi per la gestione della memoria.

Il ciclo d'esecuzione dell'interprete sarà schematizzato come segue.

1. Acquisizione della prossima istruzione da eseguire.
2. Decodifica.
3. Acquisizione degli operandi.
4. Esecuzione dell'operazione.
5. Arresto oppure ripetizione del punto 1.

Il linguaggio *L* compreso dalla macchina *ML* è detto linguaggio macchina di *ML*.

Una macchina astratta *ML* utilizzerà dei dispositivi per poter eseguire le istruzioni del linguaggio *L*. Esistono tre modi per realizzarla.

1. Realizzazione a livello Hardware.
2. Realizzazione a livello Software.
3. Realizzazione a livello firmware.

Con la realizzazione a livello hardware si avranno prestazioni maggiori rispetto agli altri casi. Tuttavia è da considerare che se il linguaggio è ad alto livello vi saranno notevoli problematiche in fase di realizzazione. Solitamente quando si parla di realizzazione a livello hardware si ha a che fare con linguaggi di basso livello e con macchine che prevedono scopi dedicati.

Con la simulazione software si ha l'idea di un linguaggio *L'* ed una macchina *M'L'.* Con i programmi scritti in *L'* e l'ausilio di *M'L'* si ha la possibilità di creare *ML* ma a discapito di una notevole lentezza d'esecuzione. Inoltre con l'introduzione di *M'L*' è stato aggiunto un ulteriore livello di astrazione.

L'ultima soluzione è quella firmware, ossia la soluzione intermedia alle precedenti, soluzione basata sull'ausilio dei microprogrammi (programmi scritti a bassissimo livello e quindi veloci).

Supponiamo di voler realizzare una macchina *ML* tramite una macchina ospite *MoLo*. Il processo di traduzione di *L* in *Lo* può avvenire in maniera implicita oppure esplicita (interpretazione interpretativa pura, interpretazione compilativa pura).
Nel primo caso si realizza un interprete scritto in *Lo* che interpreta tutte le istruzioni scritte in *L*. Ora si può prendere un programma scritto in *L* ed i suoi dati in input, darli tutti insieme all'interprete appena descritto da eseguire sulla macchina *MoLo*. [$P^L$(input) = $I^{lo}{}_L(P^L$,input) ]
Nel secondo caso la traduzione avviene tramite un compilatore. In questo contesto parliamo di linguaggio sorgente e linguaggio oggetto. Il $P^L$ dato come input al compilatore $C_{L,Lo}$ che darà come output $Pc^{Lo}$ ossia il programma compilato scritto in *Lo* il quale è eseguibile sulla macchina *MoLo*.
[$P^L$(input) = {$C_{L,LO}(P^L) = Pc^{LO}$(input)}]
*Differenze tra interprete e compilatore.*

L'interprete non genera codice e quindi necessita di poca memoria. All'interno di un programma possono essere decodificate più di una volta le istruzioni ripetute. Vi è maggiore flessibilità con l'interprete e si riesce ad interagire direttamente col programma di input. Ciò significa che è semplice implementare il debugging di programmi. In genere l'implementazione di un interprete richiede meno tempo della creazione di un compilatore.

La compilazione invece genera un nuovo programma e quindi occupa più spazio di memoria. Tralasciando il tempo necessario alla traduzione, il tempo di esecuzione sarà notevolmente minore del tempo di esecuzione dell'interprete. A differenza dell'interprete un'istruzione viene compilata una sola volta anche se questa all'interno di un programma si ripete più volte. E' notevolmente più complicato implementare strumenti di debugging inquanto non vi è una riconduzione al programma sorgente ma solo al programma compilato.

Nella realtà, capita spesso che entrambe le tecniche vengano utilizzate contemporaneamente. Un programma $P^L$ dato in pasto al compilatore $C_{L,\mathrm{LI}}$ da come risultato un $Pc^{LI}$ il quale può essere interpretato da $I^{LO}{}_{LI}$ per poter essere infine eseguito sulla macchina *MoLo*.

# *Capitolo 2 – Descrivere un linguaggio di programmazione.*

Un linguaggio di programmazione è un formalismo artificiale con il quale poter esprimere algoritmi. Questo linguaggio deve essere definito in base alla sua *grammatica*, *semantica* e *pragmatica.*

La *grammatica* è quella parte della descrizione di un linguaggio che risponde alla domanda "Quali frasi sono corrette?".
Per prima cosa nella *grammatica* si definisce l'alfabeto, tramite quest'ultimo si compongono delle sequenze di lettere, ossia le parole (o token). Una volta definito l'alfabeto e le parole, la semantica seleziona , a partire da tutte le possibili combinazioni di parole, un sottoinsieme di frasi del linguaggio stesso.

*1) Grammatica*

Supponiamo di avere a disposizione un alfabeto *A* definito come segue: *A={a,b}*.
Dato questo alfabeto ci poniamo il problema di ricavare tutte le parole palindrome. Una generica stringa palindroma *P* potrà essere una stringa vuota, una *a*, una *b* oppure una stringa palindroma preceduta e seguita da una *a* o una *b.*

*P* →
*P* → *a*
*P* → *b*
*P* → *aPa*
*P* → *bPb*

La *grammatica* appena descritta si chiama *grammatica libera da contesto*. Le grammatiche libere da contesto si possono definire come delle quadruple di non terminali, terminali, produzioni o regole ed infine simbolo iniziale: ciò da come risultato *G = (NT,T,P,S)*.

Descriviamo un esempio di operazioni aritmetiche di tipo somma e sottrazione.
*G = ([{E,I}, {a, b, +, -}, P, E)*
*P = {E → I; E → E + E; E → E – E; E → (E); I → a; I → b; I → Ia; I → Ib}*

Introduciamo ora il BNF, ossia un modo differente (ma standard e convenzionale) di definire le grammatiche. La grammatica appena descritta si può riscrivere come segue.

*<E> ::= <I> | <E + E> | <E – E> | <(E)>*
*<I> ::= <a> | <b> | <Ia> | <Ib>*

Ora i può dimostrare che l'espressione *a – b + (b – ab)* è un'espressione corretta secondo la grammatica appena descritta. Si parte ovviamente con *E (simbolo iniziale)*. Il simbolo ⇒ indica una derivazione.
⇒ *E +E (applico una produzione)*
⇒ *E – E + E*
⇒ *E – E + (E)*
⇒ *E – E + (E – E)*
⇒* *I – I + (I – I) (applico più di una regola)*
⇒* *a – b + (b – Ib)*
⇒ *a – b + (b – ab)*

Si nota che l'ordine con il quale si è proceduto non è obbligatorio e che ci si può ricondurre alla stringa sopracitata anche effettuando le derivazioni in ordine diverso.
Quest'ordine può essere rappresentato mediante l'uso degli alberi come segue.

Purtroppo questo tipo di rappresentazione ammette ambiguità: quando una stringa può essere espressa con due o più alberi di derivazione. E' necessario dunque, seppur aumentando la complessità, aggiungere tutte le informazioni necessarie alla grammatica affinché non si presentino ambiguità.

Si nota ora ciò che avviene in fase di compilazione.

1. *Analisi lessicale*: vengono letti sequenzialmente tutti i caratteri da sinistra verso destra, e vengono raggruppati tali caratteri in unità logicamente significative (*parole* o *token*).
2. *Analisi sintattica*: viene generato un albero di derivazione con i *token* precedentemente acquisiti. Questo procedimento è detto *parser*, e gli alberi generati avranno come foglie i *token* e leggendo l'albero da sinistra verso destra si avranno frasi di senso compiuto rispetto alla nostro grammatica.
3. *Analisi semantica*: se nella fase precedente viene garantita la correttezza sintattica in questo caso partendo dagli alberi di derivazione vengono controllate le dichiarazioni, e tipi di dato ed il numero di parametri delle funzioni. viene inoltre creata la tabella dei simboli e vengono ampliati gli alberi di derivazione.
4. *Generazione della forma intermedia*: Dagli alberi precedentemente ampliati viene generato un codice privo di ottimizzazione.
5. *Ottimizzazione del codice*:viene eliminato il codice inutile da codice precedentemente generato (fattorizzazione delle sottoespressioni, ottimizzazione dei cicli, eliminate chiamate a funzione con funzione stessa).
6. Generazione del codice: viene generato il codice oggetto a partire dal codice ottimizzato, successivamente possono avvenire ulteriori ottimizzazioni come la memorizzazione di alcune variabili in registri del processore.

*2) Semantica*

In questa fase si attribuisce un significato alle frasi ottenute precedentemente. Si analizzano gli stati e il modo con cui questi cambiano in relazione ai comandi eseguiti ossia tutte le varie computazioni. Gli esempi aiutano a capire meglio.

$\langle c,o\rangle \rightarrow t$ Eseguendo il comando $c$ lo stato $o$ muta in $t$.
$o[X \leftarrow 1]$ Lo stato $o$ rimane immutato tranne che per la variabile $X$ che assume il valore 1.

$$\frac{\langle c1,o1\rangle \rightarrow \langle c1',o1'\rangle \quad \langle c2,o2\rangle \rightarrow \langle c2',o2'\rangle}{\langle c,o\rangle \rightarrow \langle c',o'\rangle}$$

Significa che se la prima computazione e la seconda sono vere allora anche la terza lo è.

*3) La pragmatica*

Ai scopi del corso questa fase non è necessaria, tuttavia la pragmatica risponde alla domanda "Come uso questo costrutto?", "A cosa serve un certo comando?". Le specifiche della Grammatica e della semantica devono essere chiare ed esplicite, la pragmatica invece no.

## *Capitolo 3: Fondamenti*

Domandandosi su ciò che un programma può o non può fare si evince il problema della fermata, ossia calcolare in maniera statica se un determinato programma in fase di esecuzione possa entrare in ciclo: sapere ciò significherebbe conoscere a priori se il dato programma termina o meno.
Con una dimostrazione per assurdo si ottiene come risposta al precedente quesito che tale programma non esiste. Questo problema fa parte dei problemi d'indecidibilità dei programmi. Lo stesso esempio dimostra anche un problema di calcolabilità, ossia dimostra che esistono funzioni non calcolabili.

## *Capitolo 4: I nomi e l'ambiente*

Un nome in un programma è una sequenza di caratteri atta a distinguere e rappresentare un oggetto. Con l'introduzione del nome si fa riferimento ad una prima astrazione sui dati poiché assegnare un nome ad una variabile significa fare riferimento ad una specifica locazione di memoria.
Ovviamente nella fase di scrittura di un programma il programmatore non potrà denotare tutti gli oggetti come ad esempio i nomi dei tipi di variabile, i nomi delle operazioni aritmetiche (+, -, *, /).

La fase di associazione nome → variabile viene definita col termine inglese *binding*.
Lo spazio atto a contenere tutte le associazioni tra nomi e variabili in fase d'esecuzione, è detto ambiente. Si nota che un nome può denotare più oggetti differenti in regioni differenti del programma. Nel caso in cui diversi nomi denotino lo stesso oggetto si parla di *aliasing*.
Un blocco è una regione testuale del programma delimitata da un inizio ed una fine (di solito le parentesi graffe) con al suo interno delle dichiarazioni, nomi di variabili ecc.
I blocchi possono corrispondere ad una procedura, oppure essere a se stanti (in questo caso il blocco è detto in-line).

I blocchi devono essere annidati (dove permesso) in maniera regolare: all'apertura di un blocco deve corrispondere come prima chiusura di un blocco, la chiusura del blocco stesso. I vari blocchi possono "vedere" le variabili dichiarate esternamente e possono dichiarare delle nuove variabili locali. Tutti questi meccanismi sono denominati *regole di visibilità*.

L'entrata e l'uscita da un blocco apportano un cambio di stato nell'ambiente:

- Entrata
  - vengono create le associazioni tra nomi a gli oggetti dichiarati localmente .
  - vengono disattivate le associazioni per quei nomi che compaiono all'esterno di un blocco ma che vengono ridefiniti al suo interno.
- Uscita
  - vengono distrutte le associazioni create all'entrata del blocco.
  - vengono riattivate le associazioni già presenti all'esterno del blocco che erano state ridefinite all'interno del blocco.

In ogni ambiente possiamo quindi trovare associazioni sia attive che inattive: le operazioni che si possono svolgere sulle associazioni nell'ambiente sono il *naming,* il *referencing,* la disativazione e riattivazione e la distruzione.

Gli oggetti possono essere creati, acceduti, modificati e distrutti. La fase di creazione e di assegnazione può essere compiuta in un solo passo.

1) Scope Statico

```
{int x = 0;
void pippo(int n){
        x = n+1;        // Il valore di x lo si cerca nel blocco esterno.
}
pippo(3);
write(x);               // Stampa 4
        {int x = 0;
        pippo(3);
        write(x);       // Stampa 0
        }
write(x);               // Stampa 4
}
```

2) Scope Dinamico

```
{const x = 0;
void pippo(){           // Il valore di x va cercato nell'ambiente precedente in senso temporale.
        write(x);       // Stampa 1
}

void pluto(){
        const x = 1;
        {
                const x = 2;
        }
        pippo();
}
pluto();
}
```

## *Capitolo 5: La gestione della memoria*

La gestione della memoria può essere statica o dinamica. Nel caso in cui il linguaggio non supporti la ricorsione la gestione della memoria in modo statico è sufficiente. In linguaggi come il C dove è lecito allocare dinamicamente la memoria con operazioni tipo MALLOC e FREE si utilizzano strutture opportune dette HEAP, in tutti gli altri casi una PILA è sufficiente.

La gestione statica della memoria indica che vengono memorizzate in fase di esecuzione tutte le variabili locali, globali e le costanti: ovviamente non essendoci la “ricorsione” può essere memorizzato anche tutto il resto in modo statico.

La gestione dinamica della memoria mediante PILA avviene rispettando la politica LIFO. Ogni procedura attiva un RDA (record di attivazione) e memorizza le proprie variabili. La PILA detta "pila di sistema" con operazioni di PUSH inserisce un nuovo RDA mentre con operazioni tipo POP elimina un RDA e dealloca lo spazio riservatogli.

Un RDA è composto come segue.

- Variabili locali
  - Sono le variabili dichiarate all'interno del blocco stesso, il compilatore in base al loro tipo alloca una quantità di spazio di memoria. In alcuni linguaggi con gli array dinamici non è possibile conoscere a priori la loro capacità.
- Risultati intermedi
  - In alcuni casi è utile memorizzare dei risultati intermedi ancora prima di conoscere il risultato effettivo dell'operazione (es. nell' RDA potrebbe esserci un'associazione tipo x+y → valore).
- Puntatore catena dinamica
  - E' un puntatore all'ultimo RDA inserito nella PILA. L'insieme di questi puntatori prende il nome di CATENA DINAMICA.
- Puntatore catena statica
  - Necessario a gestire le regole di scope statico.
- Indirizzo di ritorno
  - Contiene l'indirizzo della prima istruzione da eseguire all'uscita del blocco.
- Indirizzo del risultato
  - Contiene l'indirizzo in cui è memorizzato l'eventuale parametro di ritorno.
- Parametri
  - Sezione in cui vengono memorizzati i valori passati alla funzione / procedura.

La differenza tra una procedura ed una funzione sta nel fatto che la funzione prevede un valore di ritorno, mentre la procedura no. I blocchi possono essere in-line, ossia inseriti di seguito ad un blocco con un annidamento oppure chiamati appunto tramite procedure o funzioni. Nel caso di procedure ovviamente non l'indirizzo del risultato non ha ragione d'esser presente. Per generalizzare utilizzo il termine procedura.

Nella pila di sistema che contiene i vari RDA vi è un Frame Pointer ossia il puntatore all'ultimo RDA inserito. Un altro puntatore è lo Stack Pointer il quale indica la prima posizione di memoria libera presenta nella pila.

La procedura che "generano" un altro procedura è detta chiamante, la procedura creata è detta chiamata. Il compilatore inserisce opportune righe di codice quando incontra un "creazione" di procedura: al chiamante viene aggiunta un parte di codice detta "sequenza di chiamata" mentre al chiamato vengono aggiunte due porzioni di codice dette prologo ed epilogo.

Ciò che avviene nella PILA successivamente alla sequenza di chiamata è:

- Modifica del Program Counter.
- Allocazione di spazio di memoria per il nuovo RDA.
- Modifica del Frame Pointer e Stack Pointer.
- Passaggio dei parametri (tra l' RDA del chiamante ed il nuovo RDA del chiamato).
- Salvataggio dei registri.
- Esecuzione d'inizializzazione (presente solo in alcuni linguaggi, utile ad inizializzare gli elementi del RDA).

Il caso analogo al precedente, ossia ciò che avviene dopo l'epilogo è:

- Ripristino del Program Counter.
- Restituzione dei valori.
- Ripristino dei registri.
- Esecuzione codice di finalizzazione (presente solo in alcuni linguaggi).
- Deallocazione dello spazio di memoria dalla PILA.

Con la gestione dinamica mediante HEAP si gestiscono sia i blocchi di dimensione variabile che fissa. Si fa riferimento alla spazio di memoria non allocato con il termine LL (lista libera).

- Blocchi a dimensione fissa
  - La LL viene suddivisa in blocchi di dimensione fissa. Vi è un puntatore al primo blocco libero presente nella LL e tutti i blocchi sono tra loro concatenati tramite un puntatore. Quando un blocco torna ad essere libero, questo viene posizionato in testa alla LL.
- Blocchi a dimensione variabile
  - Si utilizzano per gestire ad esempio array dinamici la cui dimensione non è nota a priori run-time. Si utilizzano nelle due tecniche seguenti (LL unica e LL multiple).

La frammentazione può essere interna oppure esterna. Per richiesta si intende la dimensione del blocco che si deve allocare.

- Interna
  - Quando viene allocato un blocco di dimensione maggiore alla richiesta. Lo spazio superfluo non utilizzato può essere allocato solo quando l'intero blocco viene deallocato.
- Esterna
  - Quando non vi è nessun blocco che preso singolarmente abbia dimensione maggiore o uguale alla richiesta, ma la somma di più blocchi liberi soddisfa la richiesta.

Per ovviare al problema della frammentazione vi sono diverse tecniche, tra cui la ricompattazione della memoria, ossia l'unione dei blocchi non allocati. Per ricompattare la memoria si devono spostare i blocchi allocati posizionando i modo contiguo i blocchi liberi. Questa procedura non è sempre ammissibile poiché si modificano gli indirizzi in memoria delle variabili ed inoltre è un'operazione molto onerosa.

La lista libera unica considera lo spazio di memoria come un'unica lista non suddivisa in blocchi.

- Allocazione.
  - Si alloca memoria a partire dall'inizio della lista fino al riempimento totale. Una volta deallocato uno spazio di memoria è possibile allocare nuovamente. Per forza di cose si genera una frammentazione.
- Ricerca blocco.
  - Supponendo una richiesta di dimensione N, viene cercata all'interno della lista un blocco di dimensione $K \geq N$. La ricerca può essere FIRST FIT, ossia si alloca il primo blocco con $K \geq N$, oppure BEST FIT cercando il blocco con dimensione più simili a N. Il primo metodo privilegia le performance mentre il secondo salvaguardia la memoria.
- Deallocazione.
  - Si riassegna il blocco alla LL controllando che vi siano blocchi liberi adiacenti: nel caso vi siano si compattano i blocchi. Questo metodo è detto compattazione parziale.

Le liste liste libere multiple riducono i costi di allocazione utilizzando due liste con dimensioni diverse dei blocchi. Le dimensioni dei blocchi possono essere statiche oppure dinamiche: in quest'ultimo caso si utilizzano due tecniche (buddy system e Fibonacci).

- Buddy System.
  - I blocchi sono di dimensione pari alle potenze di 2. Se la richiesta è di N viene cercato un blocco di dimensione $2^K \geq N$. Se tale blocco è libero, viene allocato. Se tale blocco non risulta disponibile si cerca il blocco $2^{K+1} \geq N$: di questo blocco si alloca solo metà dello spazio, la metà rimanente si restituisce alla LL. Quando si dealloca un blocco che precedentemente era stato diviso per l'allocazione, questo lo si ricompatto alla metà mancante (nel caso questa sia ancora non allocata).
- Fibonacci.
  - Stesso procedimento del Buddy System ma essendo la serie di Fibonacci “più lenta” rispetto alle potenze di 2, si evita maggiormente il problema della frammentazione interna.

Nella struttura dell' RDA precedentemente si è accennato al puntatore a catena statica. Per inserire quest'informazione nell' RDA si procede in due modi.

- Il blocco chiamato è esterno al blocco chiamante.
  - Si prede la profondità del blocco chiamante e si sottrae la profondità del blocco chiamato. La profondità delle procedura si calcola in fase di compilazione poiché il codice del programma è statico. Il puntatore dell' RDA “risale” di N posizioni la catena statica ottenute dal precedente calcolo.
- Il blocco chiamato è interno al blocco chiamante.
  - In questo caso il puntatore a catena statica prende come indirizzo l' RDA del blocco chiamante.

Nel caso in cui vi sia una diversità di profondità tra procedure notevole (in realtà questo numero non è quasi mai superiore a 3) , vi saranno rallentamenti nel calcolo del puntatore a catena statica. Per ovviare a questo problema si fa uso dei display.

Il Display è in pratica un array di N elementi tanti quanti sono i livelli di annidamento. E' una soluzione più costosa della catena statica poiché introduce appunto questo vettore in più. Grazie a questo meccanismo si riducono gli accessi in memoria ad una costante, ossia 2. In ogni “cella” del display vi è un riferimento all' RDA attivo per quel livello di profondità, quando si esce da questo RDA bisogna riattivare il suo RDA chiamante (pertanto ciò indica che bisogna memorizzare ad ogni entrata anche il riferimento all' RDA chiamante).

Nel caso dello scope dinamico è facile pensare che scorrendo la pila di sistema a ritroso si possa trovare le associazioni che non compaiono localmente nell' RDA attivo. Alternativamente a questa soluzione (valida ma poco conveniente) si può utilizzare la A-list (lista delle associazioni) nella quale sono memorizzati tutti i nomi di variabile:

- entrata in un nuovo blocco
  - vengono memorizzati nella lista le nuove associazioni.
- uscita da un blocco
  - vengono eliminate dalla lista le associazioni correnti.
- ricerca di un' associazione
  - si scorre a ritroso la lista alla prima occorrenza e si fa riferimento alla prima occorrenza del nome cercato trovato.

- il primo inconveniente sta nel fatto che il riferimento avviene in base al nome di ogni singola variabile e non in base ad uno specifico RDA.
- il secondo inconveniente risiede nell'inefficienza di questo tipo di ricerca. Se si deve ad esempio cercare una variabile globale memorizzata all'inizio dell'esecuzione, bisognerà scorrere l'intera lista.

Per ovviare a queste problematiche si introducono le CRT (central referencing table) a discapito di un rallentamento in fase di entrata e uscita da un blocco. Le CRT includono tutti i riferimenti presenti nel programma opportunamente aggiornati in base al blocco attivo a run-time. Ad ogni associazione si indica tramite un flag il fatto che sia attiva o meno ed un relativo puntatore al valore. Inoltre vi può essere una pila nascosta che comprende tutte le associazioni non attive. Ovviamente bisogna tener traccia delle associazioni ogni qual volta queste vengano aggiornate in base all'entrata di un nuovo blocco cosicché all'uscita del tale blocco il valore possa essere ripristinato. In fase di ricerca di un'associazione basta accedere alla CRT e prelevare il puntatore connesso all'associazione cercata.

## *Capitolo 6: Strutturare il Controllo*

In questo capitolo si affronta il problema del controllo di sequenza. Nei linguaggi di basso livello è sufficiente aggiornare il registro PC (Program Counter) mentre nei linguaggi ad alto livello le cose si complicano. Ovviamente nel caso di un operazione aritmetica bisogna specificare l'ordine d'esecuzione delle varie operazioni che compongono un' espressione.
Le espressioni sono i componenti essenziali di ogni linguaggio. Un' espressione è un'entità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita. L'espressione è composta da un operatore o da un'entità singola e da una serie di argomenti. Esistono tre diverse notazioni per esprimere un'espressione.

- Notazione Infissa.
- Notazione Prefissa.
- Notazione Postfissa.

La notazione infissa è la formula comune (3 + 5) * 2. Questa notazione necessita di parentesi o di regole di precedenza che indichino l'ordine con il quale eseguire le operazioni: in mancanza di queste si generano ambiguità sul risultato.

La notazione prefissa è la formula * ( + (3 5) 2). La valutazione di tale notazione risulta essere più facile della notazione infissa: si fa uso di una pila nella quale memorizzare i simboli e di un contatore che indica il numero degli operandi per ogni operando. L'algoritmo è il seguente:

1. Lettura simbolo S. PUSH(S) su PILA.
2. Se S è un operatore C = N dove N = numero di argomenti dell'operatore S.
3. Se S è un operando C--.
4. Se C ≠ 0 torna al punto 1.
5. Se C = 0
   a. Applica l'operatore l'ultimo operatore agli operandi inseriti successivamente e ottiene R come risultato. Sostituisce R agli operandi e all'operatore nella PILA.
   b. Se non vi sono altri simboli tipo operatore vai al punto 6.
   c. C = N – M con N = numero d'argomenti e M = numero di operatori. Vai al punto 4.
6. Se la sequenza dell'espressione che rimane da leggere non è vuota vai al punto 1.

Il problema del precedente algoritmo sta nel fatto che bisogna conoscere a priori il numero di operandi per ogni operatore. Inoltre bisogna effettuare un controllo per determinare se sulla pila vi sono effettivamente tutti gli operandi necessari per l'operatore in cima alla pila.

Nel caso della notazione postfissa non serve controllare che sulla pila vi siano tutti gli operandi dell'operatore in cima alla pila poiché la lettura da sinistra verso destra prevede che vi siano prima gli operandi e poi l'operatore come nella formula ((3 5) + 2) *.  L'algoritmo di valutazione è il seguente:

1. Lettura simbolo S. PUSH(S) su PILA.
2. Se S è un operatore applica agli operandi precedenti l'operatore S ottenendo il risultato R. Sostituisce R con gli operandi ed operatore nella PILA.
3. Se la sequenza dell'espressione che rimane non è vuota vai al punto 1.
4. Se S è un operando vai al punto 1.

Anche in quest'algoritmo si presenta il problema di conoscere a priori il numero di operandi per ogni operatore.

Come detto nel capitolo 2 (relativo alle grammatiche), gli alberi di derivazione possono essere usati per rappresentare le espressioni, che in questo caso chiameremo alberi sintattici. Un albero sintattico prevede le seguenti 3 regole.

- Ogni nodo che non è una foglia è un operatore.
- Ogni sotto-albero del nodo N costituisce un operando per il nodo N.
- Ogni foglia è una variabile / costante / oppure un operando elementare.

Con l'utilizzo degli alberi non vi sono problemi di precedenza tra gli operatori. In base al modo con il quale si “visita” l'albero si ottiene la forma infissa, post-fissa e prefissa.
Sorge il problema di quale espressione di pari livello eseguire per prima: se per esempio ci troviamo di fronte ad una stringa del genere < A x B x C > si dovrà eseguire prima < A x B > o < B x C > ?

- Effetti collaterali
  - La valutazione di alcune espressioni prima di altre può causare l'alterazione del risultato. In alcuni casi si evita che possano presentarsi chiamate a funzioni che causino effetti collaterali ma nella maggior parte dei linguaggi di programmazione ciò è ammesso. In JAVA per esempio si valuta sempre con l'ordine da sinistra verso destra.
- Aritmetica finita
  - Si possono presentare problemi di overflow nel caso si eseguano alcune espressioni prima di altre. Per esempio la stringa A + B – C causa un overflow, ma eseguire A – C + B non causa overflow.
- Operandi non definiti
  - Esistono due strategie di valutazione: eager e lazy. La strategia eager è la più logica, ossia si valutano prima tutti i valori degli operandi e poi si passano all'operatore che ne valuta il risultato. La strategia lazy invece prevede che tutti gli operandi vengano passati all'operatore e al momento della valutazione decide quali sono effettivamente necessari. Un'espressione del tipo < a == 0 ? b : b/a > è più consigliato valutarla con il metodo lazy evitando a priori che si possa verificare una divisione per 0, ossia un errore.

- Valutazione con corto circuito
  - Nel caso di operazioni logiche tipo < a == 0 || b/a > 2 > non si valutano entrambi gli operandi per verificare che l'espressione restituisca un valore vero o falso: infatti nel caso in cui il primo operando restituisca un valore vero, indipendentemente dal secondo operando si può concludere che il valore dell'espressione sarà vero.
- Ottimizzazioni
  - Nel caso in cui si presentino una serie di espressioni come < a = v[i]; b = a x a + c + d; > alcuni linguaggi valuteranno prima la seconda espressione (c + d) poiché il valore di a potrebbe non essere ancora disponibile (deve essere effettuato un accesso in memoria). I compilatori possono cambiare l'ordine degli operandi per ragioni di efficienza lasciando il codice in maniera semanticamente equivalente.

Un comando è un entità sintattica la cui valutazione non necessariamente restituisce un valore ma può vere un effetto collaterale. Nello specifico si può dire che un comando ha un effetto collaterale quando influenza il risultato di una computazione senza restituire alcun risultato.
Un comando di differisce da un'espressione poiché il risultato della valutazione di un'espressione è un valore mentre quello di un comando è un nuovo stato (dovuto ad un nuovo assegnamento).

Una variabile “modificabile” in matematica è un'incognita che può assumere un qualsiasi valore appartenente ad un definito insieme. Nello specifico è una sorta di contenitore (con riferimento a memoria fisica) che può assumere un nome e contenere dati di tipo omogeneo (caratteri, interi, ecc). Questo valore può cambiare nel tempo tramite le assegnazioni.
Alcuni linguaggi imperativi considerano le variabili come riferimenti ad un valore, altri le considerato come degli identificatori che denotano un valore.

Il comando di base che permette di modificare il valore ad una variabile è *l'assegnamento* (di conseguenza modifica anche lo stato).
La stringa x = 2 assegna alla variabile denotata dal nome di x il valore numerico 2.
La stringa x = x +1 assegna nella locazione di memoria denotata dalla prima x il valore della variabile denotata dal nome x aggiungendo un'unità numerica. In questo ultimo caso si evince la differenza tra la prima x (l-valore) e la seconda x (r-valore) poiché svolgono ruoli differenti. In generale gli l-valori denotano le locazioni di memoria nelle quali memorizzare i risultati.

In alcuni linguaggi (come ad esempio il C), con l'assegnamento, oltre a produrre un effetto collaterale, si ritorna il valore assegnato quindi è corretta una stringa come x = y = 2 (assegna sia ad x che a y il valore y).

Gli incrementi:

- x++
  - restituisce il valore di x, successivamente ne incrementa di un'unità il valore.
- x--
  - restituisce il valore di x, successivamente ne decrementa di un'unità il valore.
- ++x
  - incrementa di un'unità il valore di x e successivamente ne restituisce il valore.
- --x
  - decrementa di un'unità il valore di x e successivamente ne restituisce il valore.
- x += y
  - incrementa di y il valore di x e successivamente ne restituisce il valore.
- x -= y
  - decrementa di y il valore di x e successivamente ne restituisce il valore.

Oltre al comando d'assegnamento vi sono altri comandi, i quali si dividono principalmente in tre categorie: comandi per il controllo di sequenza, comandi condizionali e i comandi iterativi.

I *comandi per il controllo di sequenza* sono quei comandi tipo il ";" che delimitano l'inizio e la fine di un comando: per esempio < C1 ; C2 > indica che sarà eseguito prima il comando C1 poi il comando C2. In questa categoria vi sono anche i *comandi composti*, ossia quelli che precedentemente abbiamo definito blocchi, denotati da "{" "}" in C e "begin" e "end" in Algol.
Il GOTO è un comando che deriva dalle istruzioni Assembly il cui uso rende i codice poco leggibile, difficile da modificare e può creare situazioni complesse (come il salto all'interno di un blocco), pertanto il suo uso è fortemente sconsigliato: inoltre è stato dimostrato che il GOTO non cambia l'espressività di un programma quindi nei moderni linguaggi di programmazione (come JAVA) il GOTO è completamente scomparso.
Comandi appartenenti a questa categoria sono anche i comandi RETURN e BREAK i quali rispettivamente terminano l'esecuzione di un'iterazione e di una funzione. Altri comandi di questo tipo sono i comandi che gestiscono le eccezioni.

I *comandi condizionali* sono detti anche di selezione ed esprimono due o più alternative sul proseguo delle istruzioni di un programma. Tra questi analizziamo i comandi IF ed CASE.

- IF b-exp then C1 else C2
  - il comando analizza un'espressione booleana ed in base al suo risultato esegue o il comando C1 o il comando C2. Il ramo else può anche essere omesso. L'annidamento di più comandi IF può creare ambiguità facilmente risolvibili. In alcuni linguaggi è presente il suo terminatore ENDIF.
- CASE
  - il comando CASE si comporta in maniera analoga al comando IF annidato. Tuttavia l'uso del CASE rende più leggibile il codice e aumenta l'efficienza in fase di compilazione. Si può utilizzare una tabella di salto cosicché una volta valutata l'espressione "condizionale" si possa usare questo valore come indice di salto ed eseguire il comando opportuno. Lo svantaggio delle tabelle di salto (sono locazioni di memoria contigue) è lo "spreco" di memoria. Al termine di ogni CASE è opportuno inserire un BREAK come separatore dei comandi altrimenti si eseguono più comandi per una sola valutazione dell'espressione di condizione.

I comandi visti si qui permettono d'esprimere computazioni di lunghezze statiche.
Con i *comandi iterativi* alcune porzioni di codice possono essere rieseguite più volte.

- WHILE b-exp do C1
  - esegue il comando C1 fintanto che l'espressione booleana è vera.
- FOR i=1 to 10 by 1 do C1 (Algol) oppure FOR exp1 exp2 exp3 do C1 (JAVA)
  - il comando FOR ha subito diversi cambiamenti nel corso degli anni e diversi linguaggi ne hanno fatto un uso differente. Nel caso di Algol il comando FOR è un'iterazione determinata poiché è possibile conoscere a priori il numero di volte che il comando C1 verrà eseguito: il caso contrario avviene per il C o per JAVA dove il FOR è un'iterazione indeterminata. Il fatto che vengano utilizzati sia iterazioni determinate che indeterminate (ossia sia cicli WHILE che FOR) è per una questione di leggibilità. Nel caso specifico di C o JAVA possiamo notare come il FOR si comporti come abbreviazione del WHILE.
- FOR-EACH e : a (e è un singolo dato, a è un'array di dati tipo il dato e)
  - Le iterazioni sono spesso utilizzate su dati di tipo array (o collezioni di dati dello stesso tipo) e pertanto si può utilizzare questo comando per ottenere un codice più leggibile e più elegante.

Per programmazione strutturata s'intende una serie di convenzioni da rispettare: come si è visto per il comando GOTO che in alcuni linguaggi è ancora presente ma fortemente sconsigliato.

- Progettazione strutturata (da un concetto astratto ad una soluzione via via più concreta).
- Modularizzazione del codice.
- Utilizzo di nomi significativi.
- Utilizzo frequente di commenti.
- Uso di tipi dato strutturati.
- Uso di costrutti strutturati per il controllo (linearità nell'esecuzione dei comandi, assenza di salti in un blocco ed in un altro come per il comando GOTO).

Il concetto di ricorsione è molto simili per certi versi all'iterazione precedentemente vista ma dal punto di vista logico viene da pensare che ad ogni chiamata ricorsiva venga creato un nuovo RDA sulla pila di sistema. In realtà è sufficiente un solo RDA che viene sovrascritto ad ogni iterazione e la funzione ricorsiva deve essere strutturata in maniera tale che dopo la chiamata ricorsiva non vi sia più nessuna istruzione da eseguire. Ogni funzione ricorsiva può essere scritta con ricorsione in coda facendo eseguire altrimenti il codice dopo la chiamata ricorsiva ad una dedita funzione detta di "continuazione".

## *Capitolo 7: Astrarre sul Controllo*

Il concetto di astrazione sul controllo permette di gestire la complessità di un problema (ovviamente nel campo della programmazione): a questo si aggiunge l'astrazione sui dati che vedremo in seguito.

I programmi sono suddivisi in sotto-programmi e funzioni. Le funzioni sono identificate da un nome, hanno un ambiente proprio e scambiano informazioni con le altre funzioni mediante i parametri. I parametri possono essere formali (quelli che compaiono nella fase dichiarativa della funzione) e attuali (quelli che compaiono in fase di chiamata della funzione). In alcuni linguaggi si distingue il termine funzione da procedura per il fatto che la prima restituisca un valore mentre la seconda comunica col chiamante tramite l'ambiente non locale. Un comando spesso usato per "scambiare" un parametro tra due funzioni è RETURN il quale come già detto termina anche l'esecuzione di chi lo invoca. Per ambiente non locale s'intende tutto ciò che è visibile ad una funzione ma non appartiene alla funzione stessa.

I modi con i quali si "collegano" i parametri formali con quelli attuali sono detti modalità di passaggio dei parametri: queste modalità sono principalmente tre.

- Ingresso chiamante → chiamato
  - Passaggio x Valore
    - Viene copiato per intero il valore del parametro attuale nel parametro formale, al termine della procedura chiamata il parametro formale viene distrutto e non vi è nessun collegamento tra parametro formale e attuale.
  - Passaggio x Costante
    - Il parametro formale non è modificabile nel corpo della procedura chiamata, coincide con il Passaggio x Valore e la sua implementazione è a scelta della macchina astratta.

- Uscita chiamante ← chiamato
  - Passaggio x Risultato
    - Avviene un collegamento tra parametro formale e parametro attuale.
- Ingresso e Uscita chiamante ⇄ chiamato
  - Passaggio x Riferimento
    - Avviene un'associazione tra parametro attuale e parametro formale (aliasing). Le modifiche effettuate sul parametro formale si ripercuotono anche nel parametro attuale.
  - Passaggio x Valore-Risultato
    - Molto simile al Passaggio x Riferimento ma non prevede l'aliasing.
  - Passaggio x Nome
    - Viene sostituito il parametro formale con il parametro attuale per tutte le occorrenze presenti nel corpo della funzione chiamata.

Un linguaggio si può dire di ordine superiore se ammette la presenza di funzioni di ordine superiore: queste funzioni sono dette tali se possono prendere come parametro un'atra funzione oppure la possono restituire. Solitamente i linguaggi che prevedono funzioni che restituiscono altre funzioni sono detti *funzionali.*

Con le funzioni di ordine superiore è necessario definire il concetto di deep-binding e shallow-binding, ossia il riferimento all'ambiente non locale per una funzione passata come parametro.
Il deep-binding utilizza come ambiente non locale l'ambiente attivo al momento della creazione del legame tra parametro formale e parametro attuale (ovviamente riferito alla funzione). Lo shallow-binding invece fa riferimento all'ambiente attivo durante l'invocazione della funzione passata come parametro.

Le eccezioni, in alcuni linguaggi, possono essere controllate definendo un gestore ed incapsulando la porzione di codice nella quale si possa verificare appunto un'eccezione. Nel caso in cui tale eccezione si verifichi, il controllo passa al gestore associato.

## *Capitolo 8: Strutturare i dati*

Un linguaggio di programmazione offre costrutti e meccanismi per strutturare i dati. Ogni linguaggio di programmazione ha un proprio sistema di tipi.

- Insieme dei tipi predefiniti.
- Meccanismi per definire nuovi tipi.
- Meccanismi per il controllo dei tipi (equivalenza, compatibilità, influenza).
- specifiche dei vincoli.

I tipi di un linguaggio di programmazione possono essere classificati in base a come i valori possono essere manipolati

- denotabili (possono essere associati ad un nome).
- esprimibili (possono essere il risultato di un'espressione complessa).
- memorizzabili (possono essere memorizzati in una variabile).

Se i controlli dei vincoli di tipizzazione avvengono in fase compilativa si ha la tipizzazione statica, nel caso avvenga in fase interpretativa si ha la tipizzazione dinamica.

Riassunto dei tipi di dati.

- Tipi scalari.
  - Tipi predefiniti.
    - Booleani, caratteri, interi, a virgola fissa, complessi, reali e VOID.
  - Enumerazioni (serie di valori) e intervalli.
  - Tipi ordinali (hanno la nozione di precedente e successivo).
- Tipi composti.
  - Record, array, insiemi, puntatori e tipi ricorsivi.

Nel caso dei puntatori si presenti un caso del genere:

```
int* p;
int* q = (int*)malloc(sizeof(int));
p = q;
free(q);
```

si nota che il puntatore p fa riferimento ad una porzione di memoria non allocata, ciò è un errore detto Dangling Reference.

I tipi ricorsivi sono tipicamente gli alberi e le liste.

Per comparare due tipi di dato si necessita di regole di equivalenza le quali possono essere di tipo opaco (equivalenza per nome) e di tipo trasparente (equivalenza strutturale). Nel primo caso è il nome stesso a definire un nuovo tipo, pertanto due tipi sono equivalenti solo se hanno lo stesso nome. Nel secondo caso invece si analizza la struttura del tipi e quindi:

```
struct A{
        int x;
        int y;
};

struct B {
        int x;
        int y;
};

struct C {
        int w;
        int z;
};
```

si ha che A e B sono equivalenti tra loro, ma A non è equivalente con C.

Il concetto di compatibilità determina se due tipi sono tra loro compatibili: per esempio è il controllo che avviene nel momento in cui si passa dal parametro attuale al parametro formale e nel caso di un' assegnamento.
Siano dati due tipi di dato A e B, la loro compatibilità è dimostrata da:

- A e B sono equivalenti.
- i valori di A sono un sottoinsieme dei valori di B.
- tutte le operazioni ammesse in B lo sono anche in A.

Diversi tipi di dati possono essere convertiti in altri tipi tramite la conversione implicita detta anche coercizione ed esplicita detta anche cast. Con la prima tecnica si ha che l'interprete forza la conversione, mentre nel secondo caso è il programmatore che specifica tale conversione. Ovviamente non tutte le conversioni sono ammissibili ma solo quelle che il linguaggio è in grado d'implementare.

Il concetto di polimorfismo indica che un oggetto può avere più di un tipo assegnato. Per esempio ordinare un array di caratteri o di interi potrebbe essere comodo usare una funzione come la seguente: SORT (TYPE A[]). Quest'ultima funzione non specifica il tipo di parametro formale, ma ciò avviene a tempo debito, ossia quando la funzione viene invocata.
Il polimorfismo può essere

- ad hoc (overloading).
    - Ad uno stesso nome possono corrispondere un certo numero di oggetti diversi (numero noto a priori). Per esempio nella somma l'operatore + può essere usato nel contesto degli interi (2 + 3) oppure nel contesto dei reali (2,5 + 3,2).
- universale (parametrico / inclusione).
    - La differenza dall'overloading sta nel fatto che ad un nome può corrispondere un numero infinito di tipi. Per esempio La funzione SWAP(TYPE X, TYPE Y) può operare su diversi tipi di dato: ovviamente quando questa funzione viene invocata in base a parametri formali la funzione viene istanziata diversamente. Questo tipo di polimorfismo può essere implicito o esplicito relativamente al fatto che vi sia una specifica notazione testuale nel programma o meno.

Nella fase di compilazione le funzione polimorfe vengono istanziate per ogni tipo di chiamata. Sempre nel caso della funzione SWAP ci si chiede però quale sia lo spazio di memoria da assegnare alla variabile indefinita: nel codice eseguibile saranno presenti più istanze della funzione stessa. Una variante sarebbe quella di istanziare una sola volta la funzione polimorfa e anziché allocare direttamente lo spazio attuo a contenere i vari tipi di parametri si ricorre ai puntatori.

Solitamente una funzione è scritta in modo tale che indichi i parametri d'ingresso e il tipo di dato che si ottiene dalla valutazione della stessa. Il concetto di “inferenza di tipo” (utilizzato in ML) fa si che partendo dal tipo di parametri in ingresso si possa rilevare il tipo di dato in uscita.

Come precedentemente accennato i puntatori possono creare problemi di dangling reference: i linguaggi che ammettono la manipolazione totale dei puntatori sono detti un-safe.
Per ovviare al problema dei dangling reference si utilizzano tre tecniche (tomb stone, lock & keys, garbage collector).

Le tomb stone prevedono l'introduzione di un puntatore in più per ogni riferimento e ciò comporta un costo in termini d'efficienza.

Esempio

| | | |
|---|---|---|
| p = malloc(); | p → \|TS\| → 1 | Entrambi i puntatori puntano ad una tomb stone la |
| q = malloc(); | q → \|TS\| → 2 | a sua volta punta al valore reale. |
| *p = 1; | p → \|TS\| → 1 | Nel caso di dangling reference il puntatore non punta |
| *q = 2; | q ↗ | ad una zona di memoria non allocato ma ad una tomb |
| q = p; | q → \|TS\| | stone. |
| free(p); | | |

Nel caso dei lock & keys si utilizzano anziché dei puntatori intermedi delle parole generate random o incrementali che fungono da lucchetti e chiavi. Il programma precedente viene schematizzato come segue.

| | |
|---|---|
| p (123) → 1 (123) | Viene generata la parola 123 per il puntatore p e 124 per q. |
| q (124) → 2 (124) | La chiave viene associata al dato ai quali si riferiscono i puntatori. |
| q (123) → 1 (123) | Nel caso di un nuovo riferimento la chiave e il lucchetto sono copiati. |
| q (123) → NULL (0) | La chiave e il lucchetto non combaciano poiché al momento della deallocazione il valore della parola viene posto a 0. |

Il garbage collector si concentra su due fasi principali: il riconoscimento degli oggetti attivi e il possibile riuso degli spazi di memoria non più utilizzati.

- Reference counter (contatore dei riferimenti)
  - Consiste nel tener traccia di quanti puntatori fanno riferimento ad ogni oggetto: quando questo valore è 0 questo oggetto non è più attivo. Questa tecnica è poco efficiente e nel caso di puntatori ricorsivi non la si può adoperare. (p → q; q → p;)
- Mark & Sweep.
  - Vengono marcati inizialmente tutti gli oggetti sull'heap come “non utilizzati”, poi con una visita “a ritroso” dell'heap si marcano tutti gli oggetti attivi. La funzione analoga ripercorre tutti gli oggetti lasciando immutati quelli marcati come attivi e restituendo alla lista libera i rimanenti. Rimane il problema della frammentazione.
- Il rovesciamento dei puntatori.
  - Per la visita “a ritroso” di un grafo come per la Sweep, si fa uso della tecnica del rovesciamento dei puntatori: con questa tecnica si evita il problema dello spreco di memoria poiché si usano solo due puntatori ausiliari.
- Mark & Contact.
  - La funzione di Sweep può essere incorporata con quella di compattazione della memoria. Nonostante il suo uso comporti un costo computazionale notevole, le locazioni di memoria attive e inattive risultano essere contigue e ciò agevola la ri-allocazione.
- Copia (stop & copy).
  - Si suppone di avere due “zone” di memoria distinte (fromspace e tospace): con questa tecnica si ha un costo computazionale ragionevole e si evitano i problemi inerenti alla frammentazione della memoria. Durante l'esecuzione solo una ”zona” risulta allocabile e quando lo spazio a disposizione si esaurisce avviene una copia da una zona all'altra. Vi è inoltre un root-set, ossia una pila di puntatori agli oggetti attivi, e grazie al suo impiego vengono copiati tutti gli oggetti nella nuovo “zona” in modo contiguo ovviamente. Quando la copia termina il ruolo delle due “zone” di memoria viene invertito.

# *Capitolo 9: Astrarre sui dati*

Concettualmente si può dire che una macchina fisica riconosce un solo tipo di dati, ossia le stringhe di bit. Il concetto di astrazione fa si che il programmatore possa adoperare diverse tipologie di dato (int, boolean, char, ecc.).
Il tipo di dato astratto, detto ADT, è caratterizzato da:

- nome del tipo.
- rappresentazione e implementazione del tipo.
- insieme dei nomi delle operazioni per manipolare tale tipo di dato.
- per ogni operazione un'implementazione come al secondo punto.
- capsula di protezione che separa il nome del tipo dalla sua implementazione.

Se l' ADT serve ad incapsulare un tipo di dato con le sue relative operazioni, un modulo serve per definire un tipo composto da più tipi, descrivendo le varie regole di visibilità grazie alle quali si realizza l'incapsulamento dell'informazione. Il modulo distingue ciò che è pubblico, cioè visibile all'esterno da ciò che è privato, ossia che rimane occultato all'esterno.

# *Capitolo 10: Il paradigma orientato agli oggetti*

Con la programmazione orientata agli oggetti l' ADT viene sostituito da un concetto più complesso che si riassume in quattro punti:

- permettere l'incapsulamento e l'occultamento dell'informazione.
- possibilità di ereditare implementazioni da costrutti analoghi.
- nozione di compatibilità in termini di operazioni ammissibili su diversi tipi.
- selezione di operazioni in funzione del tipo usato.

Il costrutto principale della programmazione orientata agli oggetti è appunto l'oggetto, il quale è composto da dati ed operazioni che tramite un'interfaccia è possibile manipolare (salvo regole di visibilità). I dati di un oggetto sono detti campi o variabili d' istanza. Ogni oggetto appartiene almeno ad una classe e la sua istanziazione si presenta in questa forma:

*Pippo pluto = new Pippo();*

dove con Pippo si indica una classe, con pluto il nome dell'oggetto, con new si alloca uno spazio di memoria per tale oggetto e Pippo() rappresenta il costruttore per tale oggetto.

I costruttori solitamente hanno la stessa nomenclatura della classe alla quale appartengono e possono essere presenti più volte all'interno del corpo con tipi e numero di parametri differenti (polimorfismo).

I metodi ed i campi di una classe possono essere *statici,* ossia non facenti riferimento ad uno specifico oggetto ma appartenenti direttamente come caratteristica di una classe.

Supponiamo di avere un oggetto di classe B che estende un oggetto di classe A: l'interfaccia di A sarà un sottoinsieme dell'interfaccia di B e B si dice sottoclasse di A ed in modo analogo possiamo definire A super-classe di B o ancora B estende A. Una sottoclasse oltre ad ereditare tutti i metodi ed i campi dalla super-classe può aggiungerne di nuovi e sovrascrivere quelli ereditati: ciò è detto overriding.

Le regole di visibilità sono quindi di tre tipi e si applicano sia ai campi che ai metodi:

- pubbliche (tutti possono vedere).
- private (solo la classe corrente può vedere).
- protette (solo le classi che estendono la classe corrente può vedere).

Alcuni linguaggi di programmazione ammettono che una classe possa estendere più di una classe. Ciò potrebbe causare problematiche inerenti ai conflitti dei nomi dei metodi. Le soluzioni a questo problema sono principalmente tre:

- evitare sintatticamente che ciò accada.
- ogni conflitto risolto dal programma con riferimento esplicito alla classe (distinzione testuale del nome del metodo come A::B o C::B o simili).
- decidere convenzionalmente come risolvere tale problema in base all'ordine di ereditarietà (per esempio class B extends A,C, prima si considera A poi C).

Ad ogni modo il rapporto beneficio-costi non pende in modo netto verso nessuna delle due tecniche di eredità singolo o multipla.

Col termine DISPATCH indichiamo la sezione dinamica dei metodi. Supponiamo di avere i seguenti casi:

```
class A{                        class B extends A{              B b = new B();
        protected int x;                private int y = 0;      A a = b;
        public void reset(){            public void reset(){    a.reset();
                x = 0;                          x = 0;          Questa chiamata a reset() è
        }                                       y++;            della classe B poiché i metodi
        public void inc(){              }                       operano in modo dinamico.
                x++;                    public int getY(){
        }                                       return y;
        public int getX(){              }
                return x;       }
        }
}
```

In modo contrario si comportano le variabili d'istanza, ossia in modo statico come nell'esempio seguente:

```
class A{                        class B extends A{              B b = new B();
        int x = 1;                      int x = 2;              b.stampa();     // 2
        public void stampa(){           public void stampa(){   print(b.x);     // 2
                print(x);                       print(x);       A a = (A)b;
        }                               }                       a.stampa();     // 2
}                               }                               print(a.x);     // 1
```

Che l'oggetto di tipo B stampi 2 è piuttosto logico, ma l'oggetto di A si comporta differentemente. Si nota l'introduzione di un cast poiché nella sintassi JAVA gli oggetti non sono copiati ma clonati e questo meccanismo fornisce un Object, ossia un generico oggetto che deve essere castato all'oggetto desiderato. L'oggetto di tipo A stampa per causa dei metodi dinamici il numero 2 la prima volta e poi stampa il numero 1 poiché le variabili d'istanza sono appunto statiche e non dinamiche come per i metodi.

In questo esempio è presente anche un shadowing, ossia un mascheramento: come si nota la classe B che estende la classe A utilizza una variabile di nome x già presente nel corpo della classe A, e questa ridefinizione è detta appunto shadowing.

La rappresentazione dei vari oggetti può avvenire trattandoli come se fossero dei RDA.
Si può quindi notare come dall'esempio seguente la semplicità di paghi in termini di inefficienza poiché bisogna scorrere linearmente tutta la gerarchia delle classi.

```
class A {               class B extends A {    B b = new B();
        int a;                  int b;
        void f();               void f();
        void g();               void h();
}                       }
```

Il Late Binding è quando un collegamento non avviene in fase di compilazione ma in fase d'esecuzione: è notevole inoltre il fatto che durante la stesura del codice il riferimento non è immediatamente definito. L'esempio spiega meglio questo concetto.

```
class A{                        class B extends A {             B b = new B();
        void f(){                       void g(){               b.f();   //stampa B.
                this.g();                       print("B");     La chiamata al metodo f() ereditato da
        }                               }                       A richiama a sua volta il metodo g(). Il
        g(){                    }                               riferimento è per oggetto e non per
                print("A");                                     classe quindi il metodo g() invocato è
        }                                                       quello della classe B.
}
```

Nel caso di ereditarietà singola si può utilizzare un altro approccio per la rappresentazione delle classi facendo uso di una struttura dati detta vtable. Nel caso in cui la classe B estenda la classe A, B copierà per intero la vtable di A e aggiungerà eventualmente o ridefinirà alcuni metodi.

L'approccio appena descritto può causare problematiche per via dell'ereditarietà se una classe può presentare malfunzionamenti in base ad una modifica della sua super-classe. Ciò è detto problema della classe base fragile. Questo problema può sorgere per più motivi (uno architetturale ed uno implementativo).
Per esempio una super-classe potrebbe essere ricompilata con l'aggiunta di un nuovo metodo, le sotto-classi non potranno fare riferimento correttamente ai metodi ereditati poiché l'offset è cambiato. I metodi devono essere selezionati quindi, in modo dinamico.
Per ovviare a questo problema si fa uso delle costant-pool ossia delle tabelle dei simboli nelle quali sono contenute tutte le informazioni inerenti alle variabili d'istanza e ai metodi usati nella classe. Nelle costant-pool vi sono gli indici di dove si trovano determinati metodi o variabili ma non l'indirizzo esatto, pertanto in fase d'esecuzione tale indirizzo deve essere risolto, accedendo alla locazione di memoria corretto e aggiungendo l'indice riportato: tale indirizzo viene poi memorizzato nella costant-pool per non effettuare nuovamente questa risoluzione d'indirizzo.

Quando viene invocato un metodo si distinguono principalmente quattro casi:

- metodo statico (riferimento alla classe e non all'istanza).
  - non vengono passati altri parametri oltre a quelli presenti nel metodo stesso.
- metodo selezionato dinamicamente (metodo virtuale).
  - viene passato il riferimento all'oggetto su cui è chiamato il metodo.
- metodo selezionato dinamicamente invocato mediante “this” (metodo speciale).
  - viene passato il riferimento per this.
- metodo proveniente da un'interfaccia (metodo non implementato).

Per la relazione dei sottotipi ogni istanza della classe A ha come tipo le super-classi di A. Per esempio il metodo *void pippo(A a)* accetta come parametro qualsiasi sottoclasse di A.
In tal caso abbiamo un polimorfismo di sottotipo, tutte le operazioni applicabili ad A sono applicabili anche ai sottotipi di A. Questo polimorfismo non coincide non è di tipo universale poiché gli unici parametri accettati come già detto sono i sottotipi di A.

```
A pippo(A a){          C c = new C();
        return a;      c = (C)pippo(c);      // Viene introdotto un cast altrimenti vi sarebbe un
}                                            // errore in quanto A != C.
```

I generici in JAVA si possono trovare per esempio nelle liste dinamiche.

```
ArrayList<String> lista = new ArrayList<String>();
lista.add(“ciao”);
String s = lista.get(0);                     // In questo caso non è necessario il cast poiché il tipo
                                             // è già definito in partenza.
```

Supponiamo di avere una classe C che estende una classe A dalla quale eredita il metodo *stampa()*.
Una definizione del genere *ArrayList<A> lista = new ArrayList<A>();* ammette che un oggetto di tipo C possa essere passato alla lista in questione, ma in realtà non è così.

```
C c = new C();
lista.add(c);                  // Ciò non è possibile poiché il tipo ArrayList<A> != ArrayList<C>
lista.add(c);                  // A questo inconveniente la sintassi JAVA corretta sarebbe la
for (A a : lista)              // la seguente.
        a.stampa();
```

Il problema precedente è risolto con l'introduzione del tipo generico T che estende la classe A. In tal modo, la lista che viene passata comprende tipi che sono estensioni di A, quindi accetta sia A che tutti i suoi sottotipi.

```
<T extends A> void stampaTutti(List<T> L){
        for (A a : L)
                a.stampa();
}
```

Con l'uso dei generici si usa la tecnica di cancellazione in fase di compilazione: prima avviene il controllo dei tipi successivamente il programma originale viene trasformato in modo tale che tutti i generici scompaiano (es. List<Integer> diventa List). La JVM non necessita di modifiche particolari per il supporto dei generici ed il codice generico può coesistere con quello non generico con sicurezza introducendo, eventualmente, dei cast in fase di compilazione.

Come precedentemente accennato *List<A>* e *List<B>* sono diversi nonostante B sia un sottotipo di A. Ciò è spiegato meglio con un esempio per assurdo.

```
List<int> listaInt = new List<int>;
List<Object> listaObj = listaInt;       // Supponiamo che ciò sia possibile!
listaObj.add(new String("ciao"));       // Ovviamente anche una stringa è un sottotipo di Object
int i = listaInt.get(0);                // Errore: ci si aspetta un intero e si ottiene una stringa
```

Ciò dimostra il perché il compilatore non accetta la seconda riga di codice per motivi di sicurezza sui tipi.

Nel caso degli array le cose non cambiano, ma anziché ottenere un errore in fase compilativa lo si ottiene a run-time. Gli array in JAVA preservano i sottotipi e per questo sono detti covarianti.

```
int[] i = new int[10];
Object[] o = i;                 // Ciò è semanticamente corretto e quindi accettato
o[0] = new String("ciao");      // Anche questo è corretto, ma a run-time genera errore
```

Sempre nell'argomento dei sottotipi possiamo dire che anche i metodi possono essere ridefiniti come nel seguente esempio. Ammettiamo che Q sia un sottotipo di P.

```
class A{                        class B extends A{
        P niente(P p){...}              Q niente(P p){...}      // Questa ridefinizione di metodo è lecita
}                               }
```