Linguaggi e stili computazionali

Antonio Natali

31 maggio 2004

Indice

Capitolo 1

Elaboratori e stile imperativo

1.1 Elaboratori come interpreti di linguaggi

1.1.1 Linguaggi

Alla base di ogni comunicazione vi è un linguaggio. Come il linguaggio ha svolto un ruolo fondamentale nello sviluppo della cultura e nella evoluzione della specie umana, così esso assume un ruolo altrettanto importante nello sviluppo dei sistemi di elaborazione e nelle comunicazioni uomo-macchina e macchina-macchina. Grazie alle proprie capacità espressive, il linguaggio costituisce un potente stimolo per suggerire concetti e modi di soluzione appropriati per problemi di un ampio spettro di domini applicativi. Al tempo stesso, però, esso può rappresentare un impedimento, quando le sue metafore non siano adeguate o divengano obsolete.

Un dizionario definisce il linguaggio come *l'insieme di parole e metodi di combinazione di parole usate e comprese da una comunità di persone.* Tuttavia questa definizione non è sufficientemente precisa per un linguaggio di programmazione, in quanto non consente di caratterizzare i linguaggi in modo da superare le ambiguità tipiche dei linguaggi naturali o in modo da comprendere cosa voglia dire denotare processi computazionali meccanizzabili e stabilire proprietà su cui poter effettuare ragionamenti. Occorre pertanto caratterizzare il linguaggio come un particolare sistema matematico che consenta di dare risposta a domande come:

1. Quali sono le frasi lecite di un linguaggio?

2. Data una frase, è possibile stabilire se essa appartiene a un linguaggio?

3. Come si stabilisce il significato di una frase?

4. Quali elementi linguistici primitivi occorrono per poter esprimere la soluzione a un qualunque problema (risolvibile)?

1.1.2 Sintassi e semantica

La descrizione dei linguaggi di programmazione viene effettuata introducendo notazioni formali diverse per la descrizione della struttura delle frasi (**sintassi**) e per la descrizione del significato di una frase (**semantica**).

La descrizione della sintassi

La struttura delle frasi lecite di un linguaggio può essere descritta attraverso un dispositivo formale denominato **grammatica**.

Grammatica
Una **Grammatica** è definita da una quadrupla di enti (`VN, VT, S, P`) ove:

- **VN**= insieme di *meta-simboli* detti anche simboli *non- terminali* o *variabili*. I meta-simboli rappresentano categorie sintattiche.
- **VT** = insieme dei *simboli terminali*.
  - Sono stringhe su un alfabeto A, definito come un insieme finito e non vuoto di simboli atomici.

    Il simbolo A^* denota la *chiusura* dell'alfabeto, cioè tutte le stringhe che si possono comporre con esso:

    $A^* = A^0 \cup A^1 \cup A^2 \cup ...$

    Il simbolo A^+ denota la *chiusura positiva* dell'alfabeto, che non comprende la stringa vuota (di solito denotata con ϵ); ergo,

    $A^+ = A^* - \epsilon$
  - Gli insiemi `VT` e `VN` devono essere *disgiunti*, cioè $VT \cap VN = \phi$.
  - L'insieme $V = VT \cup VN$ si dice *vocabolario* della grammatica.
- `S` = *scopo* della grammatica: è un simbolo particolare in `VN`, detto anche *start-symbol*.
- `P` = insieme finito di *produzioni*, ossia regole di riscrittura, della forma

  $a \rightarrow b$,

  con

  $a \in V^+$, $b \in V^*$

Una grammatica stabilisce le regole di un gioco, le cui mosse sono costituite dalle produzioni: tutte le partite hanno origine dallo scopo `S` e consistono nell'applicare le produzioni per generare le frasi del linguaggio che la grammatica descrive.

Iniziando dallo scopo `S` ed applicando una o più regole di produzione, si ottengono via via diverse *riscritture*, dette *forme di frase (sentential forms)*;
la sequenza di passi necessari per produrre una certa forma di frase
a partire dallo scopo `S` si chiama *sequenza di derivazione*.

Una forma di frase (sentential form) costituita *di soli simboli terminali* si dice *frase* del linguaggio.

Si dice Linguaggio *L(G)* generato dalla grammatica `G` l'insieme delle frasi formate da soli simboli terminali raggiungibili dallo scopo `S` di `G` tramite una sequenza di derivazione.

Un requisito essenziale è che la grammatica di un linguaggio sia definita in modo da poter risolvere il problema della *decidibilità* del linguaggio,

ossia il problema di decidere se una certa stringa sia o meno una frase che appartiene al linguaggio.

Sul piano pragmatico, un linguaggio artificiale è effettivamente utilizzabile solo nel caso in cui possa esista un *automa riconoscitore* computazionalmente efficiente. Questo problema, essenziale per la costruzione di traduttori ed interpreti, è oggi risolto attraverso l'uso di grammatiche generative libere da contesto.

La classificazione di Chomsky delle grammatiche

Le grammatiche libere da contesto sono anche dette grammatiche di tipo 2 nella classificazione proposta da *Noam Chomsky*, articolata su 4 livelli: 0,1,2,3. Si veda sezione **??**, pag. **??**

La descrizione della semantica

Pur se meno diffusi per via della loro relativa maggior complessità, sono disponibili anche vari metodi formali per la descrizione della *semantica* di un linguaggio, che stabilisce un secondo, più interessante, rapporto con il concetto di automa. Il rapporto tra la semantica di un linguaggio di programmazione e l'idea di *automa per la risoluzione di problemi* è a sua volta collegato al quesito fondazionale dell'informatica: *quali elementi occorre introdurre per denotare processi computazionali meccanizzabili e quali proprietà di conseguenza possono essere associate a tali processi in virtù dei sistemi di denotazione adottati?*

La risposta a questo quesito trova le sue radici nella logica matematica ed è un risultato di studi compiuti a partire dagli anni 1920-30. Per il moderno ingegnere del software questa parte della storia dell'informatica costituisce un ottimo esempio del rapporto tra analisi, progetto e realizzazione nel contesto di un'impresa che mirava a costruire una macchina capace di costituire un *elaboratore universale* di informazione.

Per affrontare il problema gli studiosi hanno inizialmente focalizzato l'attenzione su *cosa dovesse fare* una macchina del genere e solo in un secondo tempo su *come lo dovesse fare*.

1.1.3 Automi

Il concetto di **automa** (per elaborare informazione) viene introdotto come un *dispositivo concettuale* che stabilisce una precisa *relazione* tra un dato di ingresso e un dato di uscita, soddisfacendo i seguenti vincoli di realizzabilità fisica:

- se l'automa è fatto di parti, queste sono in *numero finito*;
- l'ingresso e l'uscita sono denotabili attraverso un *insieme finito* di simboli.

In questo modo non si considera alcun aspetto fisico o tecnologico specifico: l'automa può essere realizzato da un insieme di dispositivi elettronici digitali, come pure da dispositivi meccanici o biologici. L'obiettivo è appunto astrarre dai singoli, specifici casi concreti enucleando le caratteristiche essenziali.

Vi è un profondo rapporto tra il concetto di linguaggio di programmazione e il concetto di *automa risolutore di problemi*, legato alla necessità di usare linguaggi per cui esista un (efficiente) *automa riconoscitore* ossia, appunto, un procedimento *meccanizzabile* per distinguere l'insieme delle frasi lecite da quelle sintatticamente scorrette.

La storia dell'informatica ha quindi avuto inizio impostando un processo di progettazione indipendente dalle tecnologie, che ha prodotto come risultato la definizione di *modelli matematici* caratterizzati da precise proprietà. Questi modelli, detti *sistemi formali*, hanno definito di fatto il concetto stesso di *computabilità* e sono stati alla base dello sviluppo tecnologico che ha caratterizzato l'era dell'informazione.

1.1.4 Sistemi formali

Tra i più noti sistemi formali introdotti per caratterizzare il concetto di computabilità vi è una gerarchia di macchine astratte che parte dagli automi a stati finiti (**ASF**) e termina alla macchina di Turing (**TM**).
[1]

In questa gerarchia, ogni livello caratterizza la capacità di risolvere classi diverse di problemi, attribuendo alla TM la capacità più elevata.

Macchine astratte e classificazione di Chomsky

[1] Automi a capacità computazionale intermedia sono gli *automi a stati finiti con stack* e la *macchina di Turing a nastro limitato*.

Vi è una importante corrispondenza tra la classificazione delle macchine astratte e quella delle grammatiche di Chomsky: ogni tipo di linguaggio nella classificazione di Chomsky può venire posto in corrispondenza con una specifica classe di macchine astratte, ciascuna delle quali caratterizza la struttura logica di un automa che riconosce le frasi del linguaggio. In particolare:

- i linguaggi di tipo 3 sono riconosciuti da automi a stati finiti;
- i linguaggi di tipo 2 sono riconosciuti da automi a stati finiti dotati di una memoria esterna a pila (detti PDA = Push Down Automata);
- i linguaggi di tipo 1 sono riconosciuti da TM con nastro limitato;
- i linguaggi di tipo 0 non sono decidibili in generale, ma, se lo sono, sono riconosciuti da macchine di Turing (TM).

La gerarchia di macchine astratte si fonda sul concetto di un dispositivo di elaborazione che dispone di un proprio *stato interno* utilizzabile come memoria e di un insieme di mosse elementari con cui risolvere diverse classi di problemi. Una specifica macchina astratta costituisce il modello logico di un automa che risolve un dato problema, analogamente a quanto si fa oggi costruendo un diagramma di stato UML di un sistema.

Automi a stati finiti e macchina di Turing
Un ASF è definito da una quintupla: (`I, O, S, sfn, s`), ove:

```
I = alfabeto (insieme dei simboli) di ingresso
O = alfabeto (insieme dei simboli) di uscita
S = insieme degli stati
s = un particolare elemento di S (detto stato iniziale)
mfn: I x S -> O  (machine function)
sfn: I x S -> S  (state function)
```

L'uscita dipende sia dall'ingresso sia dallo stato interno.
La macchina di Turing (TM) è formalmente definita dalla quintupla (`A,S, sfn,mfn,dfn`) ove:

```
A = insieme finito dei simboli di ingresso e uscita
S = insieme finito di stati (uno dei quali  halt)
mfn: A x S -> A (machine function)
sfn: A x S -> S (state function)
dfn: A x S -> {L,R,N} (direction function)
```

Questo automa è caratterizzato dalla capacità di leggere un simbolo dal nastro, di emettere una uscita, scrivendo sul nastro il simbolo specificato dalla funzione mfn, di transitare in un nuovo stato interno come specificato dalla sfn e di spostare una ideale

testina di lettura-scrittura del nastro come specificato dalla dfn (R significa spostamento di una posizione a destra, L di una posizione a sinistra e N nessun spostamento). Quando raggiunge lo stato halt, la macchina si ferma.

Per risolvere un problema mediante una TM si può pensare di procedere in due fasi:

- definire una opportuna rappresentazione dei dati di ingresso sul nastro;
- definire la parte di controllo in modo da rendere disponibile su nastro, una volta che la TM entra nello stato halt, la rappresentazione della risposta.

Invece di costruire ex novo un automa diverso per ogni specifico problema, può risultare molto più conveniente, se possibile, costruire un **unico** automa capace da fungere da elaboratore universale di informazione. L'idea fu inizialmente formulata considerando *elaboratore universale di informazione* un qualunque ente capace di effettuare un insieme di mosse computazionalmente equivalenti a quelle di una TM ed organizzato in modo da ricevere dall'esterno una *descrizione* del proprio comportamento, con cui *simulare* il comportamento di una specifica TM risolvente il problema.

Ne consegue un modello logico articolato su due livelli orizzontali: il primo livello, invariante, realizza l'interprete di un linguaggio (il così detto *linguaggio macchina*); il secondo rappresenta un insieme di frasi scritte in linguaggio macchina: cambiando le frasi cambia il comportamento.

La concretizzazione del modello costituisce ancora una macchina astratta: la *Macchina di Turing Universale* (**UTM**) di cui l'elaboratore di Von Neumann può essere considerata una realizzazione concreta, in tecnologia elettronica. Storicamente, si fa discendere da questa la famiglia di linguaggi *imperativi*. [2]

L'avvento dell'elaboratore elettronico come oggi lo conosciamo non ha ovviamente concluso l'iter di questo progetto, che continua a svolgersi lungo le volute di un tipico processo a spirale. Infatti la progettazione degli elaboratori (intesi come *hardware* più *software di sistema*) è un processo iterativo che produce continui raffinamenti innovativi anno dopo anno, sotto la spinta di nuovi scenari e nuovi requisiti. Tra le modifiche più rilevanti degli ultimi anni vi è il fatto che il concetto di automa di elaborazione ha superato i limiti del singolo dispositivo, a favore della visione di una *rete* di elaboratori in cui il concetto stesso di elaborazione come pura trasformazione di ingressi in uscita è oggetto di discussione e revisione.

Interaction machines

[2] Il fatto che UML presenti costrutti per definire ASF quando ormai si dispone di una macchina universale è ovviamente connesso al fatto che molti problemi possono essere (e conviene siano) risolti prendendo come base una macchina più semplice.

Vi è chi sostiene che la dimensione della comunicazione (interazione con l'ambiente) proietta il concetto di automa di elaborazione in una dimensione non catturabile con i sistemi formali.

In *Foundations of Interactive Computing, Brown University Technical Report No. CS. 96-01. April 1996* Peter Wegner sostiene che gli oggetti sono la classe più usata di *interaction machines* e che la tecnologia object based è divenuta dominante perchè, in quanto interattiva, è intrinsecamente più espressiva di una specifica algoritmica.

Sul piano del modello teorico, un elaboratore di informazione nasce dunque come un puro e cieco *manipolatore di segni*. Anche se oggi ogni elaboratore concreto è capace di riconoscere e gestire diversi tipi di informazione, non vi è alcuna necessità che a questi segni l'elaboratore dia una interpretazione predefinita: l'interpretazione rimane tutta nella mente di *chi usa* l'elaboratore.

Questo concetto, che sembra molto radicale e per certi versi irrealistico, costituisce una della basi della *Computer Science* e traspare evidente nel contesto dei sistemi formali e in parte anche dai linguaggi scaturiti da formalismi alternativi alla macchina di Turing.

Tra questi formalismi alternativi, l'approccio funzionale dei formalismi di *Hilbert, Church, Kleene* è fondato sul concetto di funzione matematica e ha come obiettivo caratterizzare il concetto di *funzione computabile*. Questi formalismi sono alla base della famiglia deii linguaggi di programmazione *funzionali*.

I sistemi di produzione di *Thue, Post, Markov* partono invece dall'idea di automa come insieme di *regole di riscrittura* (dette anche produzioni o regole di inferenza) che trasformano frasi (insiemi di simboli) in altre frasi. Essi sono alla base della famiglia dei linguaggi di programmazione *logici*.

La tesi di Church-Turing

Secondo la *tesi di Church-Turing* non vi è alcun formalismo capace di risolvere una classe più ampia di problemi della macchina di Turing. Questa affermazione non può essere formalmente dimostrata o confutata. Tuttavia essa è sostenuta dal fatto che i vari formalismi introdotti allo scopo di caratterizzare il concetto di computabilità si sono rivelati tutti equivalenti a quello di Turing e quindi l'uno all'altro. I diversi formalismi si differenziano invece radicalmente per il modo con cui giungono ad esprimere la soluzione ad un problema.

La iniziale prevalenza dello stile imperativo sugli stili funzionale e logico è da ricondurre alla scarsa capacità di elaborazione e memoria dei primi elaboratori, che rendeva troppo forte la distanza tra il livello dei linguaggio macchina e il livello di astrazione connesso ai formalismi non imperativi. Oggi queste limitazioni sono ampiamente superate e ciò che va colmato è la

lacuna culturale che si è nel frattempo formata nelle menti dei progettisti e dei loro formatori.

1.1.5 Elaboratore di Von Neumann

L'elaboratore elettronico è una macchina (concreta) con cui possiamo simulare il comportamento di ogni altro ente che opera con processi meccanizzabili. Più propriamente, un elaboratore può simulare il comportamento di ogni sistema formale fino ad oggi introdotto per caratterizzare il concetto stesso di processo meccanizzabile. Infatti il linguaggio macchina di un elaboratore è definito in modo da poter esprimere la soluzione a qualunque problema che cada nella categoria dei problemi risolubili (o computabili).

Problemi non risolubili

La formalizzazione del concetto di computabilità ha permesso di chiarire che possono esistere problemi ben formulati per cui non è possibile individuare alcun algoritmo che li risolve. L'esempio canonico è quello ormai famoso dell'halt della macchina di Turing:

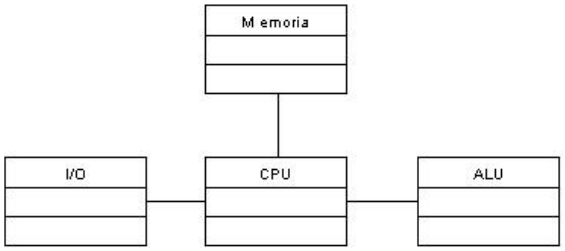
dato un qualunque programma P e una qualunque configurazione di ingresso D, stabilire se P con ingresso D termina o meno.

La non risolubilità di questo problema è relativa alla impossibilità di costruire un algoritmo valido per tutte le coppie (P,D). Ciò non esclude però che il problema possa essere risolto per coppie (P,D) specifiche o per classi particolari di programmi. Una delle tecniche più diffuse per stabilire che un problema non è risolubile è dimostrare che esso è riconducibile al problema dell'halt di Turing.

L'architettura di un moderno elaboratore elettronico si fonda ancora sui principi stabiliti negli anni quaranta da Von Neumann. Essa ha come elementi fondamentali *l'unità centrale di elaborazione(CPU)*, *l'unità Aritmetico-Logica (ALU)*, *la memoria* e *i dispositivi di I/O*.

In un processo di sviluppo basato su modelli, la fase di progettazione logica di un elaboratore potrebbe concretizzarsi nel seguente modello strutturale di primo livello:

Struttura dell'elaboratore



La memoria è una sequenza lineare di celle destinata a contenere la rappresentazione binaria di dati e di istruzioni.

Il punto chiave del progetto di un elaboratore consiste nel dotare la CPU della capacità di interpretare un insieme di istruzioni efficiente e computazionalmente completo.

La macchina di Minsky (MM)
L'insieme delle istruzioni necessarie a formare un sistema computazionalmente completo è sorprendentemente ridotto. L'esempio che segue, proposto da Minsky, è basato sulla ipotesi che l'unità di elaborazione possa fare riferimento ad un numero illimitato ma finito di celle di memoria, ognuna delle quali può contenere un numero intero arbitrariamente grande. La unità di elaborazione consiste in una sequenza di istruzioni, ciascuna associata in modo univoco ad un nome o etichetta (label), appartenenti al seguente insieme di istruzioni (instruction set):

- `ZERO cell` {poni 0 nella cella specificata dall'indirizzo cell}
- `INC cell` {incrementa di 1 il contenuto della cella specificata}
- `SUBJZ cell, label` {se il contenuto di cell vale 0 salta alla istruzione di etichetta label, altrimenti decrementa di 1 il contenuto di cell e prosegui in sequenza}
- `HALT` {ferma la macchina}

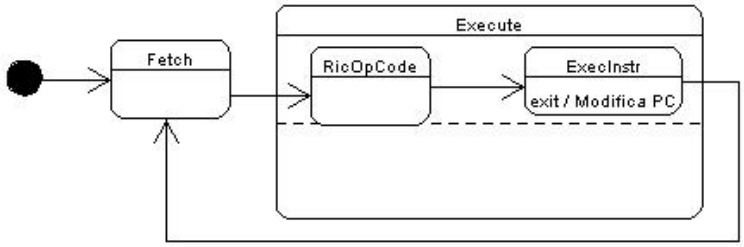
Minsky ha dimostrato che un automa di questo genere è Turing-equivalente. Ogni problema (computabile) può quindi essere risolto con questo ridottissimo insieme di istruzioni, definito in modo da riflettere lo stile imperativo tipico dei sistemi di Von Neumann. In particolare, si possono evidenziare le due categorie fondamentali di istruzioni di ogni elaboratore elettronico:

- istruzioni di manipolazione di dati (ZERO, INC, SUBJZ)
- istruzioni di controllo del flusso del progamma (SUBJZ)

Come in un elaboratore reale, per risolvere un problema con una MM occorre definire una sequenza di istruzioni, che la unità di programmazione eseguirà una dopo l'altra nell'ordine in cui sono scritte. L'unica eccezione a questa sequenzialità è data dalla istruzione di controllo SUBJZ, che può forzare l'esecuzione di una istruzione specifica non consecutiva alla precedente.

Il comportamento a tempo di esecuzione si articola nella esecuzione ripetuta di due fasi: reperimento della istruzione corrente dalla *sequenza* di istruzioni memorizzata in memoria (attività *fetch*) e riconoscimento-esecuzione della istruzione (attività *execute*). Queste due fasi sono rappresentabili attraverso un diagramma degli stati assumibili dalla CPU:

Funzionamento dell'automa hardware

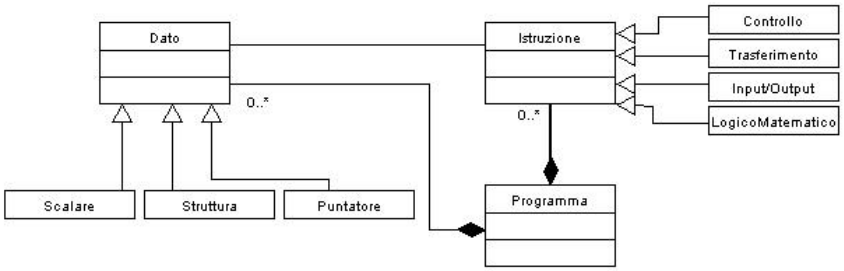


Il diagramma pone in evidenza che *concettualmente* la fase di *execute* termina con la modifica del **program counter** (PC) che viene incrementato della lunghezza della istruzione corrente, a meno che non sia oggetto esplicito di modifica da parte di istruzioni quali salto condizionato, incondizionato o ritorno.

Se l'hardware di un elaboratore elettronico potesse esprimere il proprio concetto di applicazione software, farebbe coincidere questo concetto con quello del programma descritto nella sua memoria e probabilmente direbbe che *un programma è una sequenza di istruzioni o di aggregati di istruzioni (procedure) che agiscono in memoria su dati semplici o aggregati (array) o su porte di I/O per produrre uno o più risultati.*

Direbbe anche di essere in grado di classificare le istruzioni in diverse categorie quali *trasformazioni*, *azioni* e *regole di controllo* e di saper distinguere tra diversi *tipi di dato*, quali caratteri, stringhe, numeri (interi e in virgola mobile) e valori logici, secondo il seguente modello:

Programma in linguaggio macchina



Infine, forse con un certo orgoglio, direbbe che il suo concetto di programma è tutto ciò che serve davvero in pratica. Gran parte dei costrutti tipici dei linguaggi di programmazione di alto livello oggi in uso sarebbero visti da tale hardware come ridondanza utile solo per gli esseri umani.

Questa visione riduzionista è tecnicamente corretta, proprio in quanto i concetti e le mosse conosciute da un elaboratore sono *computazionalmente*

complete, cioè in grado di esprimere la soluzione a qualunque problema risolvibile. Essa però non soddisfa le esigenze di chi deve pensare al modo migliore per realizzare la soluzione ad un dato problema o per organizzare in modo efficace la struttura di un sistema software di migliaia di linee di codice. Per questo fine occorre delineare uno spazio concettuale più vasto ed articolato rispetto a quello introdotto a livello hardware.

Va però osservato che questo nuovo spazio concettuale non sempre contiene in modo completo lo spazio concettuale dell'hardware.

Infatti la diversa velocità della CPU rispetto alle altre parti, rende necessario introdurre nell'hardware un meccanismo di ottimizzazione: l'**interruzione**. Questo meccanismo nasce per permettere alla CPU di lavorare senza dover attendere inoperosa il completamento di un comando di I/O e prevede che sia il dispositivo ad avvisare la CPU con un evento (detto *interrupt*) quando il comando è terminato. Alla ricezione dell'interrupt, la CPU sospende l'attività in corso, esegue una brevissima sequenza di istruzioni di risposta e quindi prosegue dal punto di interruzione. Lo stato computazionale della sequenza interrotta, rappresentato dal valore dei registri interni alla CPU, viene salvato in un'area di memoria gestita a pila (stack). In questo modo le interruzioni possono essere innestate, permettendo a dispositivi molto importanti o critici di essere gestiti prima di altri.

Progettare e costruire programmi corretti e consistenti in presenza di interruzioni è difficile. Per questo le interruzioni rappresentano il primo meccanismo scomparso dal linguaggio ad alto livello e completamente demandato ad un elaboratore virtuale intermedio, costituito dal *sistema operativo*.

Il sistema operativo pone a fattor comune funzionalità necessarie alla macchina virtuale di ogni linguaggio di programmazione, garantendo una gestione corretta delle interruzioni e una gestione consistente ed ottimale delle risorse. Mascherata dalla macchina virtuale del linguaggio e dal sistema operativo, l'idea di interruzione e la conseguente idea di molteplicità di flussi computazionali è stata fin dall'inizio esclusa dallo spazio concettuale dei linguaggi sequenziali.

Infatti la visibilità di questi ed altri meccanismi tipici dei linguaggi macchina risulta veramente indispensabile solo in settori applicativi alquanto circoscritti (ad esempio in alcune parti di applicazioni *real-time* stretto). La diffusione delle interfacce utente di tipo grafico e delle applicazioni di rete ha indotto ad estendere il modello sequenziale di computazione: molti moderni linguaggi permettono di esprimere e gestire *eventi* originati da dispositivi di I/O e di articolare la computazione in più flussi di controllo (*processi* o *thread*). Discuteremo questo aspetto in modo approfondito nella sezione *interazione*.

La conoscenza dei meccanismi hardware rimane comunque fondamentale

per comprendere appieno il comportamento a tempo di esecuzione di un programma scritto in un linguaggio di alto livello, che può essere considerato un **modello astratto** di un sistema software che trova la sua concretizzazione in un raffinamento costituito dal codice macchina. La corrispondenza (*mapping*) tra i due modelli è ottenuta in modo meccanico attraverso l'uso di un compilatore o di un interprete.

UML: Executable UML

Uno degli obiettivi della moderna ingegneria del software riguarda la possibilità di automatizzare il mapping tra la descrizione di un sistema espressa come un insieme di modelli UML e una implementazione concreta, attraverso un *model compiler*.

1.1.6 Elaboratori virtuali

Anche i primi, semplici esempi di programma riportati in tutti i testi di programmazione fanno intravedere elementi di uno spazio concettuale più ampio di quello dell'hardware. Ad esempio i testi di introduzione al C presentano nelle prime pagine codice simile a quello che segue:

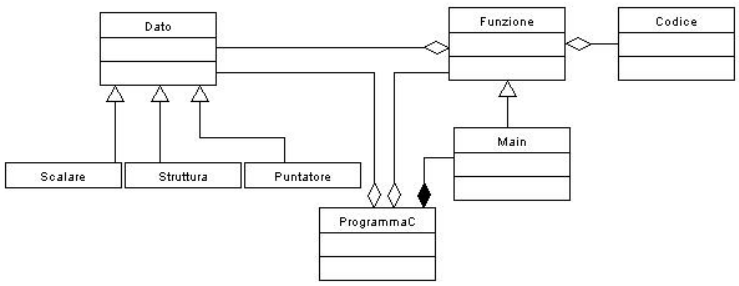
```
#include <stdio.h>
void main( int argc, char** argv ){
  printf( "Hello world\n" );
}
```

Già queste poche righe fanno intuire che è la *procedura* e non la singola istruzione a svolgere un ruolo fondamentale nella organizzazione del programma, sia per l'obbligatorietà del *main*, sia in relazione all'uso della procedura di libreria *printf* per effettuare la prima azione utile.

Programmi C

Lo spazio concettuale che il C propone ai suoi utenti si fonda sull'idea di programma come un aggregato di dati e funzioni, tra cui ne deve comparire una di nome `main`, che costituisce il punto di ingresso. Il diagramma UML che segue fornisce un modello di primo livello della struttura logica di un programma C.

Programma C



Una funzione è un aggregato di dati e codice. Le funzioni e i dati sono considerati appartenere a categorie computazionali diverse.

Programmi Java

Anche i testi di introduzione a Java iniziano spesso con un esempio simile, costringendo il lettore ad affrontare subito nuovi costrutti e nuovi concetti:

```
package Prova;
class program0{
public static void main( String args[] ){
  System.out.println( "Hello world" );
}
}//program0
```

Il programma è espresso da una procedura sostanzialmente simile a quella del C, contenuta in una classe di nome `program0` a sua volta contenuta in un package di nome `Prova`. Si intuisce dunque come il mattone fondamentale della costruzione di programmi Java non sia più nemmeno la procedura, ma sia invece legato a enti sconosciuti al mondo del C come la *classe* e il *package*.

Come leggere il programma Java

Il programma Java fa già uso di classi, oggetti e package predefiniti.

Il **package** è un costrutto introdotto da Java per raggruppare interfacce e classi semanticamente correlate tra loro. Un package in sè non ha semantica, anche se spesso l'insieme di interfacce e classi che include sono interpretate come gli elementi costitutivi di uno specifico *sottosistema*.

In particolare sono oggetti il dispositivo standard di uscita, che in Java è denotato da `System.out`, la stringa *Hello world* e l'array di stringhe `args[]`.

Tecnicamente, la frase `System.out` denota la variabile `out` della classe predefinita `System` (definita nel package, implicitamente referenziato, `java.lang`).

Questa variabile è inizializzata dall'interprete Java in modo da referenziare un oggetto che corrisponde allo standard output.

Lo standard output viene considerato da Java una specializzazione della classe `PrintStream` definita nel package `java.io`. Anche la stringa *Hello world* viene considerata un oggetto, istanza della classe predefinita `java.lang.String`. L'argomento (obbligatorio) del main, `String args[]`, è un oggetto-array di stringhe.

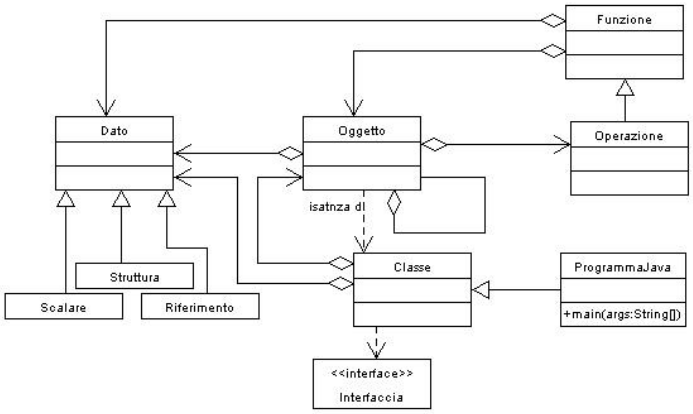
La frase `System.out.println(...)` potrebbe essere letta come una semplice notazione sintattica. Tuttavia, in accordo al modello di programmazione ad oggetti, essa deve venire interpretata come una *richiesta* all'oggetto referenziato dalla variabile predefinita `System.out` di eseguire la procedura (il metodo) di nome `println` con l'argomento specificato.

Diversamente dal C, Java è un linguaggio che pone al centro dell'attenzione l'idea di oggetto, anche se non in modo esclusivo. Infatti in Java il modello dei dati scalari del C rimane disponibile; per ciascun scalare C-like (ad esempio `int`) Java introduce però anche una *classe* (ad esempio `java.lang.Integer`) che rappresenta lo stesso tipo di informazione.

Una **classe** costituisce la descrizione della struttura e del comportamento di un insieme di oggetti costituito dalle istanze di quella classe. In Java esistono ancora costrutti sintatticamente e semanticamente analoghi alle funzioni. Una funzione però non può più esistere in modo a sè stante: deve fare sempre parte di una classe (*metodo di classe*) o di un oggetto (*metodo*).

Un programma Java viene concepito come un aggregato di dati ed oggetti definito all'interno di una classe che deve includere un metodo di classe di nome `main`. Il diagramma UML che segue fornisce un modello di primo livello della struttura logica di un programma Java.

Programma Java



Un oggetto è un aggregato di dati, operazioni ed oggetti e rappresenta l'istanza di una classe che concettualmente implementa un'interfaccia. Le operazioni sono assimilabili a funzioni il cui ambiente di visibilità è connesso ad uno specifico oggetto.

Non vi è una categoria di enti che possa costituire un denominatore comune tra oggetti, funzioni e dati.

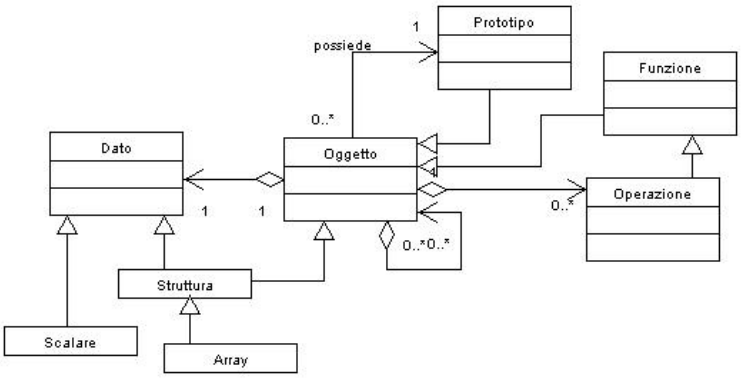
La presenza in Java del modello computazionale del C in relazione alle operazioni, e l'adozione da parte di Java di una sintassi identica a quella del C, fa sì che parti di codice C che utilizzino funzioni con dati scalari e classiche strutture di controllo `if`, `while`, `for` possano essere riusate in Java.

Programmi JavaScript

La diffusione delle applicazioni di rete ha promosso la diffusione di linguaggi che vivono in simbiosi con i browser, cioè con sistemi software nati per visualizzare pagine HTML e progressivamente evolutisi in una piattaforma estendibile di servizi da porre sul lato client.

JavaScript è un linguaggio ad oggetti con forti legami con i linguaggi funzionali. Al centro dell'attenzione è posta l'idea di *oggetto*, unitamente a un meccanismo di riusabilità a delegazione basato su *prototipi*. JavaScript considera oggetti anche i dati strutturati (*array*) e le *funzioni*.

Programma JavaScript



Essendo un linguaggio interpretato, un programma JavaScript è una *sequenza di espressioni* formate da dati ed oggetti. La valutazione di una frase JavaScript produce sempre un risultato con la possibilità di produrre effetti collaterali su oggetti.

Per motivi di sicurezza, una frase JavaScript può manipolare solo oggetti creati localmente alla computazione oppure oggetti che costituiscono componenti della pagina HTML in cui essa è inserita.

Document Object Model (DOM)

Nei browser più recenti, i vari elementi che compongono una pagina HTML sono definiti in accordo a un modello ad oggetti dei documenti, denominato **Document Object Model (DOM)**. Questo modello è oggi alla base della tecnologia nota come *HTML dinamico* (**DHTML**), che permette di manipolare tutti gli elementi di una pagina Web sul lato client senza coinvolgere il server.

Agli occhi di un utente, una pagina HTML può venire rappresentata da una gerarchia di oggetti organizzata come segue:

```
window
  self, window,parent, top
  navigator
plugins (array)
navigator
mimeTypes (array)
  frames (array)
  location
  history
  document
forms (array)
   elements (array) di ...
Button
Checkbox
FileUpload
Hidden
Password
Radio
Reset
Select
Sumbit
Text
Textarea
  anchors(array)
  links(array)
  images(array)
  applets(array)
  embeds(array)
```

Attraverso JavaScript non sono quindi accessibili oggetti per comunicazioni remote, nè risorse esterne al browser, quali file e dispositivi di I/O.

I linguaggi di programmazione ad alto livello, anche se di una stessa famiglia, presentano dunque notevoli differenze sia sintattiche sia semantiche.

La sintassi di un linguaggio mira a migliorare la leggibilità e la modificabilità del software in quanto prodotto, mentre la semantica realizza una sorta di *elaboratore virtuale* che estende l'hardware sia sul piano dell'organizzazione sia sul piano del funzionamento. La semantica di un linguaggio svolge dunque un ruolo fondamentale sia in relazione al comportamento a tempo di esecuzione del software, sia in relazione al modo di affrontare e organizzare il progetto delle applicazioni e il processo di produzione.

Un linguaggio stabilisce spesso una *ripartizione di compiti* tra l'applicazione e la propria macchina virtuale, con la conseguenza che lo sviluppatore

può evitare di includere spinose tematiche nel contesto di progetto. Il caso più noto è la attribuzione alla macchina virtuale di compiti di recupero della memoria attribuita a strutture dati dinamiche non più utilizzate (*garbage collector*). Altri importanti compiti svolti da una moderna macchina virtuale riguardano la *sicurezza*.

I casi di macchine virtuali oggi più note sono senza dubbio la *Java Virtual Machine* (**JVM**) e il *Common Language Runtime* (**CLR**).

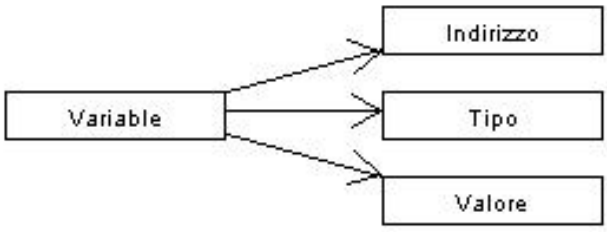
1.1.7 Il retaggio dello hardware

Molte delle caratteristiche sintattiche e semantiche dei linguaggi della famiglia imperativa risentono di una forte influenza dei meccanismi dell'hardware. Non bisogna infatti dimenticare che un linguaggio di programmazione è uno strumento per dare istruzioni di alto livello ad un elaboratore senza penalizzare (troppo) l'efficienza. Inoltre bisogna ricordare che i linguaggi di una stessa famiglia evolvono l'uno dall'altro, senza forti discontinuità, essendo il mercato il fattore che decide e seleziona le specie più forti.

Tra i meccanismi hardware che si sono riversati direttamente all'interno della famiglia dei linguaggi imperativi, emergendovi come la punta di un iceberg, sono fondamentali il concetto di *variabile* e di *assegnamento*.

A livello di linguaggio macchina, le celle di memoria fungono da contenitori di informazione codificata in codice binario. Conseguentemente i linguaggi imperativi concepiscono una *variabile* come astrazione di un gruppo di celle di memoria, e la associano concettualmente ad una terna di attributi: il *tipo* di dati che che la variabile può denotare, il *valore* da essa denotato e l'*indirizzo* della prima cella di memoria del blocco di celle ad essa associato.

Modello logico di variabile nei linguaggi imperativi



La associazione variabile-tipo permette di rilevare staticamente (a tempo di compilazione) incongruenze sull'uso delle variabili ed è uno dei punti di forza dei linguaggi ad alto livello.

è noto che in C o in Java una frase di definizione quale `int x = 2;` implica:

- la introduzione di un simbolo `x` come nome di una variabile;
- la dichiarazione che tale simbolo sarà usato solo per denotare valori interi;
- l'allocazione di un'area di memoria (in byte) di ampiezza sufficiente a contenere la rappresentazione binaria di un numero intero;
- la configurazione di questa area di memoria con la sequenza di bit che denota la rappresentazione binaria del numero intero *due*.

In JavaScript invece non è necessario dichiarare una variabile prima del suo utilizzo, nè dichiarare il tipo di dati che essa può denotare. JavaScript infatti è un linguaggio *debolmente tipizzato* e come tale non prevede un controllo dei tipi statico: perciò non obbliga a dichiarare *a priori* il tipo di valori denotabili da una variabile. L'allocazione di una variabile prevede sempre lo spazio necessario ad un riferimento.

JavaScript: Variabili

Questa parte è stata riscritta, sia per correggere l'uso dei termini definire e dichiarare, sia per eliminare ogni riferimento alla parola chiave var, che avrebbe dato undefined in fase di prova. La vecchia versione è commentata nel codice XML.

Le variabili vengono definite implicitamente al loro primo utilizzo, hanno la semantica delle variabili imperative e possono quindi essere legate (ossia associate a un valore) tramite assegnamento. Ad esempio:

```
x = 3+2*5-1; // x assume il valore 12
```

Anche la frase seguente, che era in contraddizione con quanto detto sopra (dato che abbiamo appena detto che non esiste un concetto di dichiarazione ma solo di definizione, contestuale al primo utilizzo) è stata riscritta ex novo.

Esiste anche un modo alternativo per definire una variabile, premettendo ad essa la parola chiave `var`: in tal caso però la variabile viene considerata **locale**, cioè visibile solo nel blocco in cui la definizione compare, anzichè globalmente. Ove esista anche una variabile `x` nell'ambiente globale, essa risulterà nascosta dalla definizione locale. Si ritornerà meglio su questo argomento in seguito.

Assegnamento

Il comando di assegnamento nasce come astrazione del meccanismo macchina per modificare il contenuto di una cella di memoria; per questa ragione

la semantica dell'assegnamento è talvolta fonte di confusione concettuale ed è spesso sorgente di indesiderati effetti collaterali.

Si consideri ad esempio l'istruzione `x = x+1;` (si supponga `x` una variabile di tipo *intero*). Per descriverne il significato è necessario interpretare le due occorrenze di `x` in due modi diversi: la `x` a sinistra del segno `=` significa *il contenitore associato a* x, mentre la `x` alla destra del segno `=` significa *il valore intero rappresentato da* x. Solo così la frase data acquista un senso: se la si interpretasse come equazione matematica, infatti, essa risulterebbe sempre falsa, essendo notoriamente impossibile soddisfare l'uguaglianza `x = x+1`.

L-vale e R-value

Per chiarire di quale attributo si parla quando si scrive un simbolo di variabile, si introduce il termine **l-value** per denotare l'associazione simbolo-indirizzo e il termine **r-value** per denotare il contenuto della cella, ispirandosi proprio all'istruzione di assegnamento (`l` sta per *left* e `r` per *right*).

L'istruzione `x = x+1;` mostra un'altra peculiarità dei linguaggi imperativi: ad una stessa variabile possono essere associati valori diversi in tempi diversi. Benchè a prima vista questo sembri esattamente ciò che ci si aspetta da una variabile, occorre chiarire che una variabile è tale perchè *non si può dire a priori quale valore denoti*, non perchè possa cambiare *dinamicamente* il suo valore.

Un altro elemento che può aumentare la confusione concettuale è che in molti linguaggi (tra cui C, Java e JavaScript) l'assegnamento è considerato un operatore associativo a destra, che restituisce l'r-value. è quindi possibile scrivere frasi del tipo `x = y = x-y+1`, da interpretarsi dicendo che dapprima il valore corrente di `x` e `y` è usato per calcolare l'espressione `x-y+1`, poi tale risultato è usato per modificare il valore associato alla variabile `y` e alla variabile `x`, in questo ordine (da destra verso sinistra).

Puntatori e riferimenti

Un ulteriore retaggio dell'hardware è legato alla possibilità di introdurre variabili che possono assumere come *valore un indirizzo* e di inserire tali variabili in espressioni aritmetiche, come se esse denotassero numeri interi. In questo modo il programmatore può accedere a tutta la memoria, con notevoli vantaggi sul piano delle prestazioni, ma con elevate probabilità di distruggere la coerenza e la consistenza interna del sistema.

La visione che emerge nel linguaggio ad alto livello è il ben noto concetto di **puntatore**, che i i linguaggi più recenti tendono ad eliminare a favore del concetto meno pericoloso di *riferimento*. Data una variabile che ha come valore un riferimento, il linguaggio permette di operare il solo *dereferenziamento*, cioè di accedere al contenuto delle celle di memoria referenziate.

L'eliminazione di goto e puntatori

L'incapsulamento della istruzione di salto incondizionato (`goto`) e l'eliminazione del concetto di puntatore sono tra i casi più noti del contributo della macchina virtuale di un linguaggio alla organizzazione dei programmi. Il `goto` è praticamente scomparso dai linguaggi di programmazione (e dai pensieri dei progettisti) dagli anni settanta, quando fu dimostrato che ogni algoritmo poteva essere espresso tramite la scelta condizionata `if-then-else` e il ciclo `while-do`.

Tradizionalmente ritenuti indispensabili per costruire strutture dati dinamiche, i puntatori sono stati più recentemente eliminati dai linguaggi più evoluti, pienamente sostituiti dal concetto di riferimento e dall'unica operazione di dereferenziamento. Il compito di gestire la parte di deallocazione delle strutture dinamiche è stato affidato a un componente della macchina virtuale: il *garbage collector*.

1.1.8 Simboli vocabolari e domini

introdurre una virgola nel titolo, dopo simboli (MAGARI)

L'introduzione del concetto di variabile permette di denotare le informazioni di interesse in forma simbolica, astraendo dal contesto fisico che permette a queste informazioni di esistere.

Anche chi è chiamato a operare a livello macchina non viene comunque mai costretto a lavorare a livello di codice binario, in quanto può far conto sulla sovrastruttura costituita dal **linguaggio assembler**. Il principale contributo dell'assembler consiste sia nel supportare nuovi concetti (ad esempio *espressioni*, *funzioni*, *record*) ed utili meccanismi per il riuso di codice (ad esempio le `macro`) sia, soprattutto, nel permettere di denotare istruzioni e dati in forma simbolica.

La possibilità di denotare informazione in forma simbolica è uno degli elementi fondamentali di ogni linguaggio ed è rilevante anche per la organizzazione strutturale del software in quanto *prodotto*. Il rapporto tra simbolo e struttura nasce in relazione ad alcuni problemi di carattere generale:

- Come si fa a *definire* il significato di un simbolo?
- Come si fa a *decidere* il significato di un simbolo?
- Quali sono i simboli *utilizzabili* (che hanno significato univoco) in un certo punto del testo?
- Quali sono i simboli definiti in una certa sezione del testo che sono *visibili* in altre parti?
- Come si può *verificare* se l'uso di un simbolo è coerente con il significato inteso?

Come definire il significato di un simbolo?

Ogni linguaggio introduce simboli predefiniti, il cui significato deve essere assiomaticamente noto. I problemi citati riguardano in modo specifico i nuovi simboli introdotti dall'utente, in quanto un linguaggio che non permettesse la definizione e l'uso di nuovi simboli sarebbe del tutto inutile. Nuovi simboli vengono introdotti di norma attraverso costrutti di *dichiarazione* o di *definizione*. Tecnicamente questi costrutti stabiliscono un legame (*binding*) simbolo-valore.

Come decidere il significato di un simbolo?

In un sistema software, un simbolo denota informazione in quanto esplicitamente o implicitamente correlato ad un significato. Il referente di un simbolo è un'entità nota all'eleboratore (*valore*) la cui semantica è connessa al dominio applicativo.

Simboli, valori e significati

Ad esempio il simbolo `pgreco` può essere usato per denotare il numero reale `3.14...` che nella mente di un progettista può rappresentare la metà della lunghezza della circonferenza di raggio unitario.

La definizione dei simboli significativi per un'applicazione avviene nella fase di analisi del problema, che nella moderna ingegneria del software promuove la definizione di un **vocabolario** caratteristico del dominio applicativo, ricavato da una attenta valutazione dei requisiti. La definizione del vocabolario di dominio è uno dei fattori chiave per migliorare la comunicazione tra i diversi attori del processo di costruzione, in particolare tra analisti e progettisti.

Dominio

Il termine **dominio** è usato per denotare un mondo autonomo - reale, immaginario o astratto - costituito da un insieme di **entità** astratte che operano in accordo a precise regole, politiche e vincoli.

Un dominio può fare assunzioni sulla esistenza di altri domini, caratterizzati da precise proprietà.

Un dominio viene rappresentato in UML da un **package**.

Dato un simbolo, la macchina decide il suo significato il base al binding corrente che lega quel simbolo. Ogni interpretazione semantica è esclusa.

Quali sono i simboli utilizzabili?

La risposta è quasi sempre legata alla struttura statica del testo e alla sua suddivisione in *blocchi*.

Quali sono i simboli visibili?

La risposta è legata alle regole di visibilità del linguaggio (*scope rules*) e

ai qualificatori di visibilità specificati in fase di definizione. Esempi ben noti di questi qualificatori sono `private`, `protected` e `public`, usati in C++, Java e C#.

Come si decide se un simbolo è usato in modo coerente?

Molti costrutti di *dichiarazione* (e di definizione) permettono di specificare in modo esplicito o implicito il tipo di informazione che un dato simbolo può denotare. Ad esempio, nel caso il simbolo corrisponda al nome di una variabile imperativa, la dichiarazione di tipo, come ad esempio nella frase C, Java `int x`, specifica la collezione di valori che la variabile può lecitamente assumere durante la esecuzione del programma. Questa specifica consente al compilatore di verificare se vi sono espressioni semanticamente scorrette effettuando un *controllo statico di correttezza* relativamente ai tipi di dato coinvolti nel calcolo.

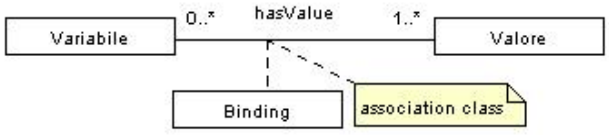
La sezione successiva è dedicata ad un approfndimento di questi punti.

1.2 Simboli, variabili e tipi

1.2.1 Binding e spazi di nomi

Tecnicamente, la conoscenza del valore denotato da un simbolo viene espressa da un legame (*binding*) che, per i simboli definiti dall'utente, deve essere esplicitamente stabilito.

Binding



NOTA: sei convinto della cardinalità 1..* sul lato destro? Stai dicendo che un valore è sempre associato ad almeno una variabile. Ma esistono anche le costanti... esistono espressioni del tipo 1+2, in cui nè 1 nè 2 sono associati a variabili. Mi sembra un poco restrittivo. Una variabile può essere associata ad uno o più valori (si pensi al caso degli array). Un valore può essere associato a 0 o più variabili. In questo secondo caso si parla di *aliasing*.

I binding e quindi i significati attribuibili ad un simbolo all'interno di un sistema software sono di solito soggetti a limitazioni temporali e spaziali.

La durata temporale di una entità del dominio applicativo e di un binding può essere infatti più breve del tempo di vita di un'applicazione. Dalla

diversità dei tempi di vita nasce la concreta possibilità che un simbolo utilizzabile non sia più connesso al suo referente, e che un referente non sia più referenziato da alcun simbolo. La prima situazione è sintomo di inconsistenza del sistema e va accuratamente evitata, La seconda è tollerabile e diventa fisiologica se la macchina virtuale del linguaggio è dotata di **garbage collector**, cioè di un dispositivo che distrugge automaticamente gli oggetti che si dimostrano non più referenziabili.

Tempo di vita

La durata di un ente del dominio si dice *indefinita* quando l'oggetto continua ad esistere fintanto che rimane aperta la possibilità di referenziarlo. Si parla invece di *durata dinamica* per quegli oggetti che vengono automaticamente distrutti quando temina il costrutto che li introduce (si pensi ad esempio al record di attivazione di una procedura).

Ogni linguaggio di programmazione presenta vari costrutti che implicano la definizione esplicita od implicita di un legame simbolo/valore. Tra questi, i più immediatamente riconoscibili sono i costrutti di *definizione* e di *dichiarazione*. La **dichiarazione** esprime solo proprietà associate al simbolo, mentre la **definizione** implica anche azioni concrete. Di solito vi possono essere più dichiarazioni di uno stesso simbolo, ma una sola definizione.

La validità di una dichiarazione o definizione è connessa alla suddivisione del testo di un programma in *blocchi* che possono essere anche innestati tra loro.

Il **blocco** costituisce una forma di delimitazione di contesto, che può essere associata a vari costrutti. Java ad esempio propone una gerarchia di delimitatori statici di contesto formata da *package*, *classi* e *metodi*.

Un linguaggio può permettere che simboli identici compaiano in diversi contesti lessicali. Se due costrutti che introducono simboli identici sono testualmente innestati, allora i riferimenti entro il costrutto più interno designano l'oggetto stabilito dal legame più interno: in altri termini, il legame più recente nasconde (*shadows*) il precedente, che viene ripristinato all'uscita dal costrutto interno.

Blocchi e regole di visibilità lessicali

In tutti i linguaggi di programmazione, un tipico esempio di blocco è costituito dal corpo di una funzione: gli argomenti della funzione sono considerati simboli definiti localmente e visibili dalle istruzioni che formano il corpo della funzione. è anche diffusa la regola secondo cui tutti gli identificatori dichiarati nel blocco in cui è inserita la definizione della funzione sono visibili pure dentro la funzione, a meno che non siano ridefiniti localmente.

Un legame simbolo/valore può essere stabilito in modo statico (lessicale) oppure dinamico. Il *legame lessicale* è connesso alla struttura testuale del pro-

gramma ed impone una *limitazione spaziale* sulla occorrenza dei riferimenti ma non impone alcuna limitazione temporale. Il *legame dinamico* avviene a tempo di esecuzione ed impone una *limitazione temporale* (la possibilità di riferimento cessa quando il costrutto che introduce il legame termina) ma nessuna limitazione spaziale.

Si parla di legame anticipato (**eager binding**) nel caso in cui il legame simbolo-valore sia stabilito il prima possibile, di solito a tempo di compilazione. Si parla invece di legame ritardato (**late binding**) quando esso è differito al momento immediatamente precedente l'uso del simbolo. Il meccanismo del late binding offerto dalle macchine virtuali dei linguaggi ad oggetti costituisce il supporto al polimorfismo connesso alla relazione di ereditarietà tra classi.

Un legame simbolo/valore può essere inoltre stabilito:

- in modo prefissato e permanente;
- in modo non prefissato, ma permanente dopo che è stato stabilito;
- in modo non prefissato e non permanente.

Un esempio di simboli a legame prefissato e permanente sono le *costanti*, mentre le *variabili* sono esempi di simboli a legame non prefissato e (nei linguaggi imperativi) modificabile (si veda la sezione 1.2.3, pag. 30).

1.2.2 Ambienti e spazi di nomi

Dato un testo, il significato dei simboli che vi compaiono è sempre legato ad un preciso contesto o ambiente di riferimento. Nei linguaggi di programmazione, si definisce ambiente (**environment**) la collezione dei legami validi in (un certo punto di) un programma, stabiliti da frasi di dichiarazione o di definizione valutate in precedenza.

In particolare si dice *ambiente lessicale* l'ambiente in cui compare la definizione testuale del simbolo, mentre si dice *ambiente dinamico* l'ambiente in essere al momento in cui un simbolo è effettivamente utilizzato.

JavaScript: eval

Come normalmente accade per i linguaggi interpretati, JavaScript offre una funzione `eval` che accetta in ingresso una stringa e la valuta come frase del linguaggio. Può essere un utile esercizio descrivere il comportamento di `eval` in relazione all'ambiente lessicale e dinamico tenendo conto dei risultati degli esperimenti che seguono: attenzione, eval(y+5) dà 7, non 5 come erroneamente indicato. Correggo.

```
eval("2+3");                // restituisce 5
eval("var x = 2; x+x");     // restituisce 4
var y = 2;
eval("y+5");                // restituisce 7
z=7;
w = function(z){
       return eval("z*3");
    }(1)                    // restituisce 3
```

Spazi di nomi Java

I package permettono di organizzare un programma Java in una struttura logica di tipo gerarchico. Ad esempio, una variabile `v` dichiarata pubblica nella classe `CL` del package `P` può essere referenziata mediante la *notazione puntata (dot notation)* `P.CL.v`. Per accedere alle informazioni contenute in un package evitando la prolissità della dot notation, si possono usare direttive del tipo Attenzione! Si usavano le #include in Java, ma non esistono. Si intendeva evidentemente usare la import (che non è una direttiva ma una vera e propria istruzione del linguaggio). Correggo la frase, lasciando commentata l'originale.

```
import P.CL;  // importa il nome CL del package P
import P.*;   // importa tutti i nomi pubblici del package P
```

La struttura logica dei package Java viene riflessa in una struttura a livello di file system, in cui ad ogni package corrisponde una diversa *directory*. Alla notazione `P.CL.v` corrisponde dunque il fatto che la frase di definizione della variabile `v` si trova in un file di nome `CL.java`, posizionato in una directory di nome `P`.

Spazi di nomi XML

Nei linguaggi per la descrizione dei documenti i concetti di blocco ed environment si concretizzano nel meccanismo dello *spazio di nomi* (**namespace**).

XML: Namespace

XML fa uso del concetto di *namespace* per denotare entità diverse aventi però lo stesso nome, distinguendole attraverso una notazione del tipo `prefix:name` ove *prefix* è un prefisso dichiarato attraverso attributi del tipo `xmlns:prefix=URI`.

`URI` sta per *Uniform Resource Identifier* e denota un locator o un nome (o anche entrambi).

Nell'esempio che segue, l'elemento `description` compare sia come parte dell'elemento `message` sia come parte dell'elemento `order`.

```
<?xml encoding="UTF-8"?>
<message xmlns:unibo="http://www.unibo.it">
<text>
  Please send immediately
</text>
<description>
  Order message
</description>

<unibo: order id="221">
  <item id="32"  quantity="16">
     <description>
           books
     </description>
  </item>
  <item id="172" quantity="31">
     <description>
           papers
     </description>
  </item>
</unibo: order>
</message>
```

La qualifica del namespace `unibo` permette di distinguere i due casi. L'elemento `message` è associato a un namespace di default che non richiede la specifica mediante prefisso.

Anche un attributo può essere associato a un namespace. Ciò è utile per estendere le informazioni correlate a un elemento senza dover modificare il tipo del documento.

1.2.3 Costanti e variabili

Una costante è un simbolo usato come sinonimo di un valore. Tutti i linguaggi introducono simboli a legame prefissato e permanente per designare valori quali numeri, caratteri, etc.

JavaScript: Costanti predefinite

JavaScript introduce come primitivi i numeri, le stringhe e i valori logici. All'interno del sistema non esistono numeri interi, ma solo reali, rappresentati in virgola mobile. I caratteri componenti le stringhe sono rappresentati secondo la codifica UNICODE. Inserire riferimento

Il tipo di un dato viene inferito dalla forma sintattica dei diversi letterali introdotti dal linguaggio: volendo, può essere esplicitato mediante la funzione predefinita `typeof`.

Ad esempio ho corretto qui sotto, distinguevi fra intero e reale, mentre hai appena detto che esiste un'unica categoria di numeri:

```
3         un valore numerico (intero)
3.2       un valore numerico (reale)
"3.2"     una stringa
'3'       una stringa (di un solo carattere)
true      un valore logico
false     un valore logico
```

Per la prova, conviene immettere frasi quali `typeof('3')`, `typeof(3.2)`, etc.; l'operatore `typeof` sarà meglio discusso più avanti.

Nelle applicazioni di rete occorre poter identificare in modo univoco risorse che possono essere allocate in un qualsiasi nodo. La *Internet Engineering Task Force* (**IETF**) ha introdotto il concetto di `URI` come una *stringa che identifica una risorsa fisica o logica*.

URI, URL, URN, UUID

La forma più consueta di `URI` è l'`URL` (*Uniform Resource Locator*) che permette di referenziare l'informazione contenuta nei file system di un qualsiasi elaboratore del pianeta connesso in Internet. Esso è composto da tre parti principali:

- un identificatore di servizio, per esempio `http://`, `ftp://`, `telnet://` che identifica il protocollo di collegamento tra due elaboratori;
- un dominio, per esempio `www.deis.unibo.it`, che identifica il particolare elaboratore (server) dove reperire l'informazione che interessa;
- un percorso, per esempio `www.deis.unibo.it/fondamenti`, che indica il percorso da seguire all'interno dell'elaboratore per individuare il file che contiene l'informazione. Questa parte è facoltativa: se non viene specificata, il server invia l'informazione contenuta in un file di default (homepage).

Sta diffondendosi anche l'uso di `URN` (*Uniform Resource Names*) che identificano le risorse tramite identificatori univoci (detti `UUID`, *Universal Unique Identifiers*). Nel caso di `URN`, il passaggio dal nome simbolico alla locazione fisica è demandato ad agenti o servizi appositi, con il vantaggio di poter spostare la risorsa senza cambiare le applicazioni.

Tutti i linguaggi introducono anche costrutti per definire nuove costanti. Il C ad esempio permette di introdurre sinonimi attraverso una direttiva al compilatore come:

```
#define CirconfUnitaria 6.28
```

Ogni apparizione, nel testo del programma, della stringa `CirconfUnitaria` viene sostituita, in una fase di pre-elaborazione alla compilazione, con la stringa `6.28`. Dunque, il compilatore C non vedrà mai il simbolo `CirconfUnitaria`, in quanto esso, dopo la pre-elaborazione, non farà più parte del testo del programma. Tale pre-elaborazione viene svolta da un componente noto come *preprocessore C*.

Manifestando una chiara influenza dello spazio concettuale del linguaggio macchina, alcuni linguaggi, come C++ e Java, permettono l'introduzione di una costante come una variabile a legame non prefissato e permanente. In particolare, Java permette di definire una costante premettendo la parola chiave `final` alla dichiarazione del tipo di una variabile:

```
final float  CirconfUnitaria = 6.28;
```

Questa impostazione è però fonte di confusione ed errori, in quanto consente di parlare - impropriamente - di allocazione di una costante e di indirizzo di una costante.

Le variabili a legame permanente sono tipiche dei linguaggi funzionali e logici, mentre le variabili a legame non permanente sono tipiche dei linguaggi imperativi: per un linguaggio imperativo infatti, una variabile è un simbolo che può essere associato, in tempi diversi, a valori diversi.

Nei linguaggi funzionali (puri) e nei linguaggi dichiarativi, la variabile tende ad essere vista invece come sinonimo di un valore. In questi linguaggi, una volta che si sia stabilito, ad esempio, che il simbolo `x` denota il numero `3.22`, non è più possibile modificare questa associazione.

Variabili locali, variabili lessicali e variabili libere

In C e Java, come in molti altri linguaggi, è permessa la definizione di nuove variabili entro un blocco, come ad esempio il corpo di una funzione. Ciò permette di introdurre e gestire informazione in accordo ai principi di incapsulamento e di localizzazione.

Le variabili la cui definizione compare testualmente nell'ambito del blocco costituito dal corpo della funzione, inclusi gli argomenti, vengono denominate *variabili lessicali*. Le variabili diverse dagli argomenti che non sono definiti entro il blocco della funzione vengono denominate *variabili libere*. Perchè il programma sia sintatticamente corretto una variabile libera deve essere definita nel blocco esterno a quello in cui compare la funzione.

JavaScript: Variabili lessicali e libere

Il seguente codice JavaScript ha una forma sintattica e una semantica dei blocchi molto simile a quelle di C e Java. Per la sintassi delle funzioni si veda la sezione 3.1.3, pag. 57 . manca il riferimento!

La funzione di nome `max` costituisce un blocco: i simboli `x` e `y` sono due simboli (di variabile) lessicali. L'invocazione `max(2,3)` dà come risposta `3`.

```
max = function(x, y){ return x > y ? x : y; }
```

Nel caso che segue, invece, il corpo della funzione presenta un blocco innestato in cui viene ridefinito il simbolo `y`. L'invocazione `max1(2,3)` dà perciò come risposta `5`:

```
max1 = function(x, y){
          if (x  > y )  return x;
          else { y = 5;  return y; };
       }
```

Infine, nel caso sottostante, all'interno del corpo della funzione di nome `max3`, i simboli `x`, `y` e `z` sono simboli (di variabili) lessicali, mentre il simbolo `k` costituisce un simbolo di *variabile libera*, che deve essere definito all'esterno della funzione:

```
max3 = function(x, y){
           var z = 3;
           return x  > y ? k : z;
        }
```

1.2.4 Tipi e sistema dei tipi

Il concetto di tipo di dato viene introdotto nei linguaggi di programmazione per raggiungere due obiettivi:

1. esprimere in modo sintetico un insieme di valori, la loro rappresentazione in memoria e un insieme di operazioni ammissibili;
2. permettere di effettuare controlli statici (al momento della compilazione) sulla correttezza del programma.

Ad esempio, la parola chiave `integer` del Pascal oppure `int` del C e Java denota il tipo dei numeri interi – o meglio, una loro approssimazione. Una variabile dichiarata di tipo `int` può assumere solo valori interi, per la manipolazione dei quali il linguaggio offre un insieme di operazioni denotate da operatori infissi quali `+`,`-`,`*`,`/`, etc. Una variabile dichiarata di tipo `int` non può partecipare ad espressioni che richiedano valori di tipo incompatibile con gli interi.

Rilevare errori in fase di traduzione, cioè in una fase di *analisi* che avvenga prima della esecuzione del programma, è molto importante in quanto non sempre una espressione scorretta viene raggiunta a tempo di esecuzione, dipendentemente dal flusso che il programma segue a partire dai particolari dati iniziali forniti: pertanto, in mancanza di una tale rilevazione, un errore potrebbe rimanere latente nel programma, manifestandosi anche dopo anni, con effetti disastrosi (come realmente avvenuto).

Ad esempio, una generica espressione `X+Y*5.32`, dove `X` e `Y` sono costanti o variabili, è semanticamente corretta se i tipi degli operandi che vi compiano sono compatibili, cioè numeri reali o interi.

Il termine *tipo* è sovraccarico di significato (*overloaded*), in quanto con esso ci si riferisce sia alle proprietà sintattiche delle espressioni e al sistema di ragionamento sui tipi (*type checking*) sia alle proprietà semantiche dei valori e alle regole per calcolare con i valori.

Type system

L'insieme di regole per associare un tipo alle espressioni di un linguaggio e per stabilire la compatibilità tra i diversi tipi di dato prende il nome di **type system**. Un sistema di tipi si dice *forte* se accetta solo espressioni ben formate in tipo, o **debole** in caso contrario. Uno dei problemi che i linguaggi devono affrontare è che un *type system* forte potrebbe rigettare troppi programmi.

Un esempio di sistema di tipi inutilmente forte sarebbe quello che rigettasse espressioni miste di interi e reali, come ad esempio `2 + 3.5`.

In generale, ogni linguaggio include regole per la conversione automatica di tipo in espressioni composte da operandi di tipo diverso. Nel caso precedente il valore numerico intero `2` è automaticamente convertito nel numero reale `2.0`. Il C (ma non Java) effettua anche una conversione tra caratteri e interi, in espressioni quali `2 + a`, che dà come risultato il carattere `'c'`.

JavaScript: Conversione di tipo

Alcuni operatori effettuano conversioni automatiche di tipo quando uno degli argomenti non è del tipo previsto. Ad esempio, un operatore relazionale tra un numero e una stringa provoca la conversione della stringa in numero, mentre l'operatore di concatenazione `+` provoca la conversione dell'operando non stringa in stringa:

```
1 + 3 + "2"          produce la stringa "42"
1 + 3 + " gatti"     produce la stringa "4 gatti"
"gatti " + 1 + 3     produce la stringa "gatti 13"
"0" + 1 + 1 <= 2     produce il boolean false (011 <= 2)
"0" + (1 + 1) <= 2   produce il boolean true (02 <= 2)
```

Alcuni linguaggi permettono che un oggetto computazionale possa appartenere a più di un tipo e offrono operatori per determinare dinamicamente il tipo di un oggetto. è il caso di JavaScript.

JavaScript: Typeof
L'operatore `typeof`, applicato ad un oggetto `X`, restituisce una stringa che denota un tipo tra quelli cui `X` appartiene. Ad esempio:

```
typeof(3)            // number
typeof(3.2)          // number
typeof("3.2")        // string
typeof '3'           // string
typeof(true)         // boolean
typeof false         // boolean
typeof typeof false  // string
```

Come si vede, l'operatore *typeof* può essere usato sia come operatore unario prefisso sia come simbolo di funzione.

Applicato a enti che non siano valoro di tipi primitivi l'operatore restituisce sempre `object`:

```
typeof new Date      // object
```

In *Common Lisp*, il dato restituito dallo stesso operatore non è una stringa ma appartiene al tipo `Standard Type Specifier Symbol`.

Conversioni esplicite di tipo

I linguaggi danno spesso la possibilità di effettuare conversioni esplicite di tipo. La notazione usata in C e Java, nota come *cast*, assume la forma `(<nomeTipo>) <valoreDaConvertire>`

Ad esempio, la frase `(int)'0'` converte il carattere `'0'` nel numero intero che corrisponde alla sua codifica interna in codice ASCII o UNICODE. Dualmente, la frase `(char)22` converte il numero `22` nel carattere avente `22` come codifica interna ASCII o UNICODE.

L'uso di conversioni di tipo è talvolta indispensabile, ma deve essere sempre attentamente valutato: molto spesso è il segnale di una cattiva progettazione, soprattutto in linguaggi orientati agli oggetti. Java infatti è molto più restrittivo del C a questo riguardo ed in particolare impedisce la conversione del tipo `boolean` in intero.

Linguaggi polimorfici

Quando un linguaggio consente che un valore o una variabile possa denotare istanze di un solo tipo si parla di *linguaggio monomorfo*. I linguaggi *polimorfici* sono viceversa caratterizzati dalla proprietà di permettere che un valore o una variabile possa denotare istanze di molti diversi tipi, purchè questi siano collegati tra loro da relazioni tipo-sottotipo.

1.2.5 Definizione di nuovi tipi

Per quanto numerosi ed articolati, i tipi di dato predefiniti da un linguaggio non possono essere sufficienti per soddisfare le esigenze pratiche di progettazione e sviluppo di sistemi software: vi sarà sempre la necessità di introdurre e gestire nuove astrazioni, in relazione alle esigenze applicative.

Pensiamo ad un'applicazione che debba automatizzare la prenotazione di posti (in aereo, treno, etc.). In questa applicazione occorre modellare concetti quali *posto*, *cliente*, *destinazione*, etc., che nulla hanno a che fare con numeri o caratteri.

Occorrono dunque anche meccanismi per organizzare i dati in strutture articolate e complesse, che possano costituire un appropriato modello del mondo reale e colmare la distanza (*gap semantico*) tra le categorie di informazione di uno specifico dominio applicativo e le categorie di dati che l'elaboratore è in grado di comprendere direttamente.

Per colmare questo divario i linguaggi hanno via via proposto vari costrutti, tra cui *espressioni di tipo* (basate su *costruttori di strutture di dati*), *moduli*, *classi*.

Espressione di tipo

Una espressione di tipo (*type expression*) descrive la struttura concreta di una nuova categoria di dati attraverso l'uso di costrutti di aggregazione, detti *costruttori di tipo*.

Tra i costrutti più diffusi di questa categoria vi sono l'`array` e il `record` (`struct` in C). Attraverso questi costrutti il progettista può definire strutture di dati che costituiscano la rappresentazione concreta delle astrazioni che ha nella sua mente.

Nell'applicazione di prenotazione di posti all'astrazione cliente può corrispondere una struttura di dati così organizzata:

- una stringa (array di caratteri) che ne rappresenti il nome;
- un intero (naturale) che ne rappresenti l'età;
- una terna di interi (record) che rappresenti il numero di telefono;
- un array di interi che rappresenti i posti riservati.

Definire nuovi tipi partendo dalla rappresentazione concreta dell'informazione pone forti vincoli sulla modularità, modificabilità e riusabilità del software. Per superare questi limiti sono stati proposti diversi modi per disaccoppiare la specifica logica di un tipo dalla sua rappresentazione concreta.

Storicamente, si è cercato dapprima di porre una barriera di astrazione intorno alla rappresentazione concreta di una struttura di dati attraverso il costrutto *modulo*. Definito come un puro contenitore di informazione, il modulo non risulta però capace di catturare appieno il concetto di astrazione di dato e viene ben presto superato a questo fine dal concetto di oggetto e dal costrutto `class` del modello di programmazione ad oggetti.

1.2.6 Ragionare sui tipi

Progettare nello spazio concettuale dei tipi di dato significa effettuare ragionamenti in cui si utilizzano spesso, oltre alla classificazione, altre due forme di astrazione: la *aggregazione* e la *generalizzazione*.

La *aggregazione* è la relazione che permette di giungere alla definizione di un ente (un concetto, un tipo) attraverso altri enti che ne formano le parti costituenti. Ad esempio un tipo *automobile* si può ottenere come aggregazione di un *telaio*, un *motore*, una *carrozzeria*, etc. Tra un'aggregazione e i suoi aggregati si stabilisce una relazione *componente-di (part-of)*.

La *generalizzazione* è il risultato di un processo di astrazione con cui si giunge a porre a fattor comune alcune caratteristiche di tipi o concetti. Ad esempio, la classe *persona* si può ottenere per generalizzazione dalle classi *dirigente* ed *impiegato*. Ma dalle stesse due classi si può ottenere per generalizzazione anche la classe *stipendiato*: quale concetto far emergere diviene perciò una delle dimensioni della progettazione.

Sottotipi e regole di compatibilità e conformità

Se `T1` è sottotipo di un tipo `T`, valgono le seguenti regole:

- a `T1` corrisponde un sottoinsieme dell'insieme di oggetti definito da `T`;
- *regola di compatibilità:*

  una variabile dichiarata assumere valori di tipo `T` può anche denotare un oggetto di tipo `T1`, ma non viceversa. Infatti, mentre è vero che ogni oggetto di tipo `T1` è sempre anche di tipo `T`, non è mai vero il contrario;
- *regola di conformità:*

  una qualunque istanza di `T1` può essere usata in qualsiasi contesto in cui è ammesso l'uso di un oggetto di tipo `T`.

Nel modello a oggetti, la regola di conformità è alla base del progetto e della realizzazione di componenti software polimorfici, in connessione al meccanismo di ereditarietà.

1.3 Caso di studio: una prima soluzione in C

1.3.1 Soluzione come programma C

Le prime pagine del volume *The C Programming Language* di *B.W.Kernighan* e *D.M.Ritchie* forniscono una soluzione molto sintetica ed efficace al problema del conteggio delle parole:

```
#define YES 1
#define NO  0
#define EOF -1

main( ){ /*Count words in input*/
 int c, nw, inword;
 nw = 0;
 inword = NO;
   while( (c=getchar()) != EOF ) {
     if(c==' ' || c=='\n' || c=='\t' )
       inword = NO;
     else if( inword == NO ){
       inword = YES;
       ++nw;
     }
   }
   printf("%d1n", nw);
}
```

Il codice utilizza simboli di costante per aumentare il livello di astrazione della descrizione senza penalizzare l'efficienza. Le variabili sono tutte lessicalmente legate e locali al blocco relativo al `main`. L'environment globale contiene (concettualmente) solo le costanti.

La variabile `c`, che denota il carattere corrente, è definita di tipo `int` in quanto usata per memorizzare il risultato della operazione di nome `getchar`, che può anche non essere un carattere: in caso il file di input sia vuoto o se ne sia terminata la scansione; infatti in tal caso `getchar` restituisce un valore negativo, denotato dalla costante `EOF` . L'operazione `getchar` non fa parte del C, ma costituisce una operazione di libreria standard universalmente disponibile.

La soluzione viene espressa come trasposizione ad alto livello di un insieme di istruzioni di modifica dei dati in memoria, salto condizionato e salto incondizionato che sono comuni a tutti gli elaboratori reali: in particolare il salto incondizionato è incapsulato entro il costrutto `while`.

La soluzione è dichiaratamente ritagliata per il dispositivo di ingresso.

Capitolo 2

La descrizione dei linguaggi

2.1 Descrizione della sintassi

2.1.1 Definizione formale di linguaggio

Introduciamo alcune definizioni:

Alfabeto: insieme finito e non vuoto di simboli atomici.

```
Esempio di alfabeto: A = { a,b }
```

Stringa: sequenza di simboli, elemento del prodotto cartesiano A^n .

```
Esempio di stringhe su A: a ab aba bb
```

Lunghezza $|S|$ di una stringa `S`. Numero dei simboli che compongono la stringa.

ϵ denota la **stringa vuota** , cioè la stringa di lunghezza zero. Si noti che $A^0 = \{\epsilon\}$

Linguaggio: un insieme `L` di stringhe.

Esempi di linguaggio:

$L1 = \{aa, aaa\}$

$L2 = \{aba, aab\}$

$L3 = \{ab, ba, aabb, abab, aaabbb, aabbab, ...\}$

$L4 = \{a^n | nprimo\}$

$L5 = \{a^n b^n | n > 0\}$

Frase (o **sentence**): una stringa appartenente a un linguaggio.

Cardinalità di un linguaggio `L`: numero delle frasi di un linguaggio `L`.

Un linguaggio si dice (in)finito quando ha cardinalità (in)finita.

Ad esempio, `L5` è' un linguaggio su `A` a cardinalità infinita.

Chiusura A^* di un alfabeto A o **linguaggio universale su A**. Insieme infinito di tutte le stringhe composte con i simboli di A ($A^* = A^0 U A^1 U A^2 \ldots$). Esempio di chiusura: $A^* = \{\epsilon, a, b, aa, ab, ba, bb, aab, aba, ...\}$

Chiusura positiva A^+ di un alfabeto A : Insieme di tutte le stringhe non nulle su A .

$A^+ = A^* - \{\epsilon\}$

2.1.2 Produzioni grammaticali

Per specificare un linguaggio finito basta elencarne le frasi. Nel caso di un linguaggio infinito invece, occorre stabilire come una notazione finita (come ad esempio quella usata per `L4` e `L5`) possa senza ambiguità descrivere un insieme infinito di elementi. Per questo scopo si utilizzano sistemi formali costituiti da regole (dette *regole di produzione* o *regole di riscrittura*) che esprimono una collezione di possibili trasformazioni di stringhe in stringhe.

Una *produzione* è una espressione della forma

$\alpha -> \beta$

ove α e β denotano stringhe. La regola esprime una trasformazione lecita (una mossa elementare) che consente di scrivere (in una frase data) la stringa β al posto della stringa α .

Una **Grammatica** è definita da una quadrupla di enti (`VN, VT, S, P`) ove:

- `VN` = insieme di *meta-simboli*, detti anche simboli *non- terminali* o *variabili*, che rappresentano categorie sintattiche.
- `VT` = insieme dei *simboli terminali*, definite come stringhe su un alfabeto A .
- `S` = *scopo* della grammatica: è un simbolo particolare in `VN`, detto anche *start-symbol*.
- `P` = insieme finito di *produzioni*

L'insieme $V = VT \cup VN$ si dice *vocabolario* della grammatica.

Gli insiemi VT e VN devono essere *disgiunti*, cioè $VT \cap VN = \phi$.

2.1.3 Derivazioni

Si dice **forma di frase** (**sentential form**): una qualunque stringa (comprensiva di simboli terminali e non) derivabile dallo scopo di una grammatica `G`.

Si dice **sequenza di derivazione** la sequenza di passi necessari per produrre una forma di frase `s` a partire dallo scopo `S` di una grammatica `G` mediante applicazione di una o più regole di produzione. Si usano le seguenti notazioni:

$S =>$ `s` : `s` deriva da `S` in una sola applicazione di produzioni (un passo)

$S =^{+}>$ `s` : `s` deriva da `S` in una o più applicazioni di produzioni

$S =^{*}>$ `s` : `s` deriva da `S` applicando zero o più applicazioni di produzioni

Si dice **frase** una forma di frase costituita da soli simboli terminali.

Si dice **Linguaggio L(G)** generato dalla grammatica `G` l'insieme delle frasi `s` tali che:

$s \in VT^*$ e $S =^{*}> s$

Esempio di linguaggio generato.

Il linguaggio $L5 = \{a^n b^n | n > 0\}$ può essere descritto dalla grammatica `G5=(VN, VT, S, P)` con `VT={a,b}`, `VN={F}`, `S=F` e le regole di produzione:

```
F ::= a F b
F ::= a b
```

La prima regola stabilsce che il meta-simbolo `F` (scopo della grammatica) può essere riscritto come la frase `aFb`. La applicazione di questa produzione consente la effettuazione di un ulteriore passo generativo, a causa della presenza di `F` nella nuova frase. La seconda regola stabilisce una alternativa di riscrittura di `F`, scegliendo la quale non si provoca (nè si consente) altra generazione. Ogni stringa prodotta applicando le regole precedenti che non contenga il meta-simbolo `F` è una frase del linguaggio generato dalla grammatica.

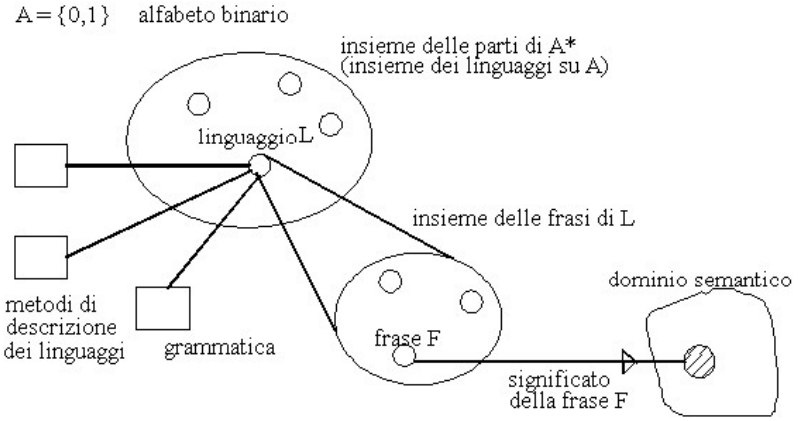
Le regole di produzione sono frasi di tipo dichiarativo che esprimono che `L5` è costitituito da frasi composte da un ugual numero (al minimo 1) di caratteri `a` e `b`, in cui tutti i caratteri `a` precedono i caratteri `b`, fornendo contemporaneamente un modo per generare queste frasi. La sequenza:

```
F => aFb => aaFbb => aaabbb
```

costituisce la sequenza di derivazione della frase `aaabbb` di `L5`.

2.1.4 Interpreti e compilatori

La figura riassume in modo schematico il modo con si caratterizza il concetto di linguaggio.



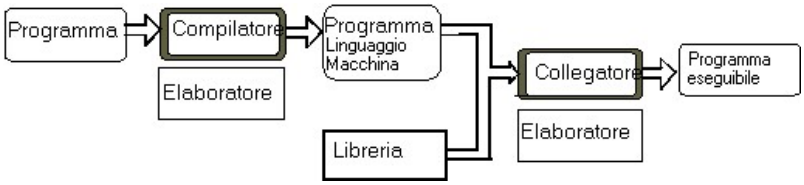
Interpreti

Un interprete per il linguaggio `L` è un programma che accetta come ingresso le frasi di `L`, eseguendole una alla volta. L'uscita di un interprete è quindi la valutazione di una frase di `L`.

Compilatori

Il compilatore per un linguaggio `L` è un programma che accetta come ingresso un programma scritto in `L`. L'uscita del compilatore è una riscrittura dell'intero programma in un altro linguaggio (di solito nel linguaggio macchina di uno specifico elaboratore) se questo risulta sintatticamente e semanticamente corretto.

Normalmente un linguaggio di alto livello consente di esprimere una collezione di operazioni già rese disponibili dall'elaboratore o dal sistema operativo. Il complesso di queste operazioni viene di solito incluso in una libreria di programmi specifica per il particolare elaboratore usato, che il linguaggio in qualche modo eredita attraverso il suo collegamento col programma tradotto.



2.1.5 Analisi sintattica e semantica

L'analisi lessicale

L'analisi lessicale consiste nella individuazione delle parole individuali (dette in gerco tecnico **token**) che compongono una frase.

I token che possono comparire nelle frasi di un linguaggio vengono stabiliti dalle regole grammaticali che descrivono il linguaggio.

Un analizzatore lessicale è un componente che, data una frase espressa da una sequenza di caratteri, restituisce la sequenza ordinata dei nomi, parole-chiave, simboli di punteggiatura, etc. che compaiono in quella frase.

L'analisi sintattica

L'analisi sintattica consiste nella verifica che una frase può essere costruita in base alle regole grammaticali che descrivono il linguaggio. Un analizzatore sintattico è un componente che, data la sequenza di token prodotti dall'analizzatore lessicale, produce di solito una rappresentazione interna della frase, in forma di albero.

L'analisi semantica

L'analisi semantica consiste nel calcolo del significato del linguaggio. Un analizzatore semantico è un componente che, data la rappresentazione intermedia prodotta dall'analizzatore sintattico, controlla la corenza logica interna del programma (ad esempio se le variabili sono usate dopo essere state definite, se sono rispettate le regole di compatibilità in tipo, etc). Nel caso di un compilatore, l'analizzatore semantico può trasformare la rappresentazione delle frasi in modo più adatto per la generazione di codice.

2.2 Tipi di grammatiche

2.2.1 La classificazione di Chomsky

Nel formalismo delle produzioni grammaticali, il parametro più importante è costituito dalla forma delle produzioni. Se infatti si impongono delle restrizioni sulla forma delle regole di riscrittura, allora certe mosse non divengono più possibili. Ne consegue un sistema formale meno potente ma non per questo meno interessante ed importante dal punto di vista informatico. Noahm Chomsky ha introdotto un insieme ormai classico di possibili restrizioni sulle produzioni, che è alla base della seguente classificazione delle grammatiche:

GRAMMATICHE DI TIPO 0.

Nessuna restrizione sulle produzioni.

GRAMMATICHE DI TIPO 1 o dipendenti dal contesto.

Le produzioni hanno la forma:

$xAy ::= x\alpha y \;\; x, y \in V^*, A \in VN, a \in V^+ (a \neq \epsilon)$

La variabile A può essere sostituita con la stringa non vuota solo quando appare nel contesto di `x y`.

Un altro modo per caratterizzare le produzioni delle grammatiche dipendenti dal contesto è il seguente:

$\alpha ::= \beta \ \ \alpha, \alpha \in \ \ V^+ |\alpha| <= |\beta|$

Ogni produzione non diminuisce mai la lunghezza della forma di frase corrente, mentre le grammatiche di tipo 0 possono includere produzioni che accorciano la forma di frase

GRAMMATICHE DI TIPO 2 o libere da contesto o algebriche

. Le produzioni hanno la forma:

$A ::= \alpha \ \ A \in VN, a \in V^+ (a \neq \epsilon)$

La variabile A può essere sempre sostituita con la stringa α . Le produzioni di tipo 2 sembrano meno restrittive di quelle di tipo 1 e possono indurre alla idea che i sistemi generativi di tipo 1 siano più vincolati. In realtà è facile convincersi che le produzioni di tipo 2 sono incluse in quelle di tipo 1 e che queste consentono maggiore capacità espressiva.

Se ogni produzione ha la forma:

$A ::= uBv \ \ con A, B \in VN, u, v \in VT$

$A ::= u$

allora `G` si dice **lineare** e `L(G)` linguaggio lineare.

GRAMMATICHE DI TIPO 3 o regolari.

Le produzioni hanno la forma:

$A ::= a$

$A ::= aB \ \ A, B \in VN, a \in VT$

oppure la forma:

$A ::= a$

$A ::= Ba \ \ A, B \in VN, a \in VT$

Nel primo caso la grammatica si dice lineare a destra, mentre nel secondo caso lineare a sinistra.

2.2.2 Relazioni tra i tipi di grammatiche

I diversi tipi di grammatiche sono in relazione gerarchica tra loro. In particoare ogni grammatica regolare (*regular*) è anche libera da contesto (*context-free*), ogni grammatica context-free è anche dipendente da contesto (*context-sensitive*) e ogni grammatica context-sensitive è anche di tipo 0.

Un linguaggio `L(G)` si dice contextsensitive, context-free e regular quando può essere generato rispettivamente da una grammatica G context-sensitive, context-free o regular. Si dice lineare quando può essere generato da una grammatica lineare.

Un linguaggio (ad esempio regolare) può essere generato da più di un tipo di grammatica (anche di tipo 2,1,0). Il fatto che una grammatica `G` generi un linguaggio, non significa che questo sia necessariamente dello stesso tipo della grammatica.

Due grammatiche che generano lo stesso linguaggio possono essere definite equivalenti. Va però subito detto che una grammatica può essere preferibile ad un'altra dal punto di vista della analisi sintattica e del processo di compilazione. Il concetto di equivalenza tra grammatiche va dunque considerato con attenzione.

2.2.3 La stringa vuota

La stringa vuota non fa parte dei linguaggi generati da grammatiche di tipo 1, 2 o 3. Le definizioni precedenti possono però essere estese ammettendo produzioni della forma:

$S ::= \epsilon$

a patto che `S` sia lo scopo della grammatica e che `S` non compaia nella parte destra di alcuna produzione.

In questo modo la produzione precedente può essere usata solo al primo passo di una derivazione e le stringhe non possono più accorciarsi. Si può dimostrare (si veda [Hopcroft-Ullman]) che se `L` è un linguaggio di tipo 0,1,2 o 3, allora anche $L \cup \{\}$ e $L - \{\}$ sono dello stesso tipo.

Esempio

Le produzioni:

$S1 ::= \epsilon S1 ::= aSbS ::= aSbS ::= ab$

definiscono il linguaggio context-free $L5 \cup \{\}$

cioè $\{a^n b^n | n \geq 0\}$

Sempre in [Hopcroft-Ullman] si può trovare la dimostrazione al seguente teorema.

Teorema.

Se `G =<VN,VT,S,P>` è una grammatica context-free in cui ogni produzione è della forma:

$A ::= \alpha$, con $\alpha \in V^*$

(α può essere ϵ)

allora esiste una grammatica context-free `G1` che genera lo stesso linguaggio `L(G)` in cui ogni produzione è della forma

$A ::= \alpha$, con $\alpha \in V^+$

oppure

$S ::= \epsilon$

ed `S` non compare nella parte destra di alcuna produzione.

Il teorema assicura che l'unica differenza di una grammatica context-free con

$\epsilon - rules$

rispetto ad una senza tali produzioni, è che la prima può generare un linguaggio che include la stringa vuota.

2.3 Il problema del riconoscimento

2.3.1 Alberi di derivazione per grammatiche context free

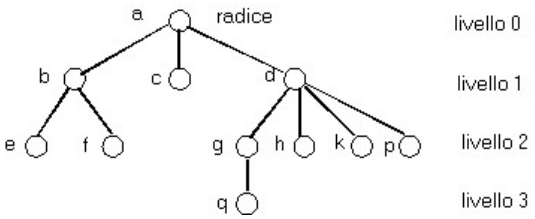
Per descrivere una derivazione è molto utile usare una forma particolare di grafo aciclico, detta *albero*.

Sia `A` un insieme finito e `T` una relazione su `A`. `T` si dice *albero* se esiste un unico elemento `r0` (detto *radice*) per cui vi è un unico cammino (collezione di nodi connessi) in `T` da `r0` ad ogni altro elemento in `A` e non esiste alcun arco entrante in `r0`. Dato un albero `T` di radice `r0`, si dice:

- insieme dei **discendenti** diretti (figli) di un nodo `m` di `T`: l'insieme dei nodi connessi dagli archi che si diramano da `m`;
- **discendente** di un nodo `m`: un nodo `n` tale che esiste una sequenza di nodi $n_1, n_2, \ldots n_k$ per cui

  $n_1 = m, n_k = n$ e

  n_{i+1} è un discendente diretto di n_i per ($1 < i < n$);
- **livello** di un nodo `m`: lunghezza del cammino (numero di nodi) dalla radice `r0` al nodo `m`;
- **grado** di un nodo `m`: il numero dei figli di `m`
- **foglia**: un nodo senza figli (cioè di grado `0`)



I nodi `e`,`f` sono figli di `b` e discendenti di `a`. Il grado del nodo `a` è `3`, quello del nodo `d` è `4`. I nodi `e,f,c,q,h,k,p` sono foglie Tra i figli di ogni nodo esiste una relazione d'ordine che distingue tra il primo, secondo, etc. figlio (di norma disegnati da sinistra a destra).

Se `G=(VN, VT, S, P)` è una grammatica context-free, un albero è un albero di derivazione per `G` se:

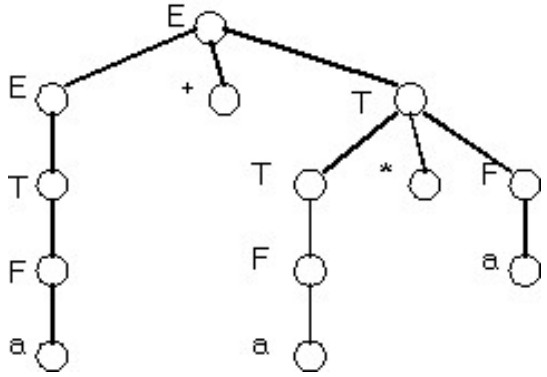
- ciascun nodo è associato ad un simbolo di `V`

- la radice è lo scopo `S` della grammatica
- se i nodi $A_1, A_2, ...A_k$ sono i figli ordinati di un nodo di etichetta `A`, allora $AA_1 ::= A_2, ...A_k$ è una produzione in `P`.

Esempio Si consideri ad la grammatica `G = (VN,VT,S,P)` con:

```
VN = {E,T,F} , VT = {a,+,*,(,)} , S = E,
P = {
E ::= E + T | T
T ::= T * F | F
F ::= ( E ) |
a }
```

L' albero di derivazione della frase `a+a*a` è il seguente:



L'albero di derivazione di una frase esprime la molteplicità delle singole sequenze di derivazione ottenibili attraverso la applicazione delle produzioni.

Derivazione canoniche

Supponendo di adottare un metodo di riscrittura che sostituisce sempre la variabile più a sinistra, (la così detta derivazione canonica sinistra), nel caso dell'esempio si avrebbe:

```
E => E+T => T + T => F + T => a + T => a + T * F => a + F * F =>
a+a * F => a + a * a
```

Sostituendo invece sempre la variabile più a destra (derivazione canonica destra), si otterrebbe la sequenza:

```
E => E+T => E+T*F => E+T*a => E+F*a => E+a*a => T+a*a =>
F+a*a => a + a * a
```

In [Hopcroft, Ullman] si può trovare la dimostrazione che $S =^*> s$ se e solo se esiste un albero di derivazione per la frase `s`.

2.3.2 Decidibilita dei linguaggi

Ogni frase di un linguaggio context-free può essere generata mediante una derivazione canonica (sinistra) (per la dimostrazione si veda [Hopcroft, Ullman]).

I linguaggi dipendenti dal contesto (e quindi anche quelli liberi da contesto e regolari) sono decidibili. I linguaggi di tipo 0 invece non è sempre detto siano decidibili.

La decidbilità, cioè il fatto che per linguaggi di tipo 1 (e quindi 2 e 3) esista un un procedimento meccanizzabile (algoritmo) capace di stabilire in un tempo finito se una qualunque frase costituita da simboli dell'alfabeto appartiene o meno al linguaggio è evidentemente un requisito essenziale per ogni linguaggio di programmazione.

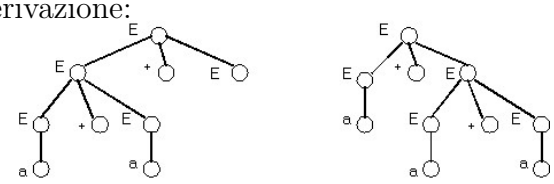
In particolare si comprende perchè la sintassi dei linguaggi di programmazione viene generalmente descritta mediante grammatiche libere da contesto, o meglio da classi speciali di grammatiche context-free. Infatti le produzioni di questo tipo di grammatiche hanno una struttura tale da assicurare la definizione di riconoscitori e traduttori molto più efficienti di quanto non sia possibile nel caso di linguaggi generati da grammatiche dipendenti dal contesto. Le grammatiche regolari sono molto usate per decrivere alcune sottoparti di un linguaggio, come ad esempio identificatori e numeri.

2.3.3 Ambiguita di grammatiche libere da contesto

Una grammatica context-free `G` si dice *ambigua* se esiste una frase `L(G)` che ammette due o più derivazioni canoniche sinistre distinte, o per cui vi sono almeno due alberi sintattici. Supponiamo ad esempio di riscrivere le produzioni precedenti come segue:

```
E ::= E + E
E ::= E * E
E ::= ( E )
E ::= a
```

Se consideriamo la stringa `a + a + a`, possiamo costruire due alberi di derivazione:



Ovviamente l'ambiguità di una grammatica non è una caratteristica desiderabile e va eviata.

2.3.4 La notazione BNF

La notazione BNF (*Bakus-Naur Form*) prende il nome da coloro che la proposero per descrivere la sintassi del linguaggio *Algol60*, che può essere considerato il progenitore di tutta la famiglia dei linguaggi imperativi di tipo *Pascal*. Rispetto alle notazione precedenti si introducono le seguenti nuove convenzioni:

- i simboli non terminali sono sempre racchiusi tra le parentesi `< >`
- per semplificare la scrittura di più produzioni per uno stesso non-terminale si introduce il meta-simbolo `|` per indicare una alternativa.

Ad esempio, le produzioni:

```
<letter> ::= A
<letter> ::= B
<letter> ::= C
```

si possono compattare in quella che segue:

```
<letter> ::= A | B | C
```

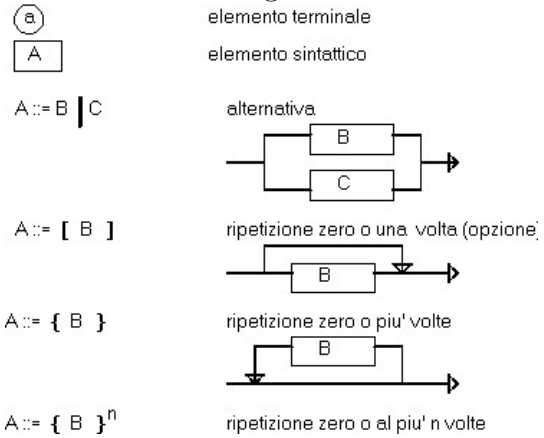
Frasi ricorsive come:

```
<alfa> ::= <letter> | <alfa> <letter>
```

possono essere riscritte includendo tra { } le forme ripetibili zero o più volte:

```
<alfa> ::= <letter> { <letter> }
```

Per rendere più facilmente leggibili le regole di produzione, si fa inoltre uso di una rappresentazione grafica delle produzioni, detta diagramma sintattico, come riassunto nella figura:

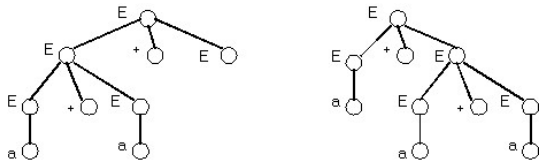


Complessivamente, queste notazioni costituiscono un meta-linguaggio con cui descrivere le regole di produzione.

Esempio (identificatori).

E' molto comune che i linguaggi di alto livello introducano identificatori per denotare i vari oggetti del proprio dominio, nel modo che segue:

```
<letter> ::= A | B | ... | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<ident> ::= <letter> | <ident> <letter> | <ident> <digit>
```



Questa sintassi ammette identificatori di qualunque lunghezza. Per stabilire un limite massimo di 10 simboli significativi (come fanno molti compilatori ed interpreti) si può scrivere:

```
<ident> ::= <letter> | <letter> { <idrest> }9
<idrest> ::= <letter> | <digit>
```

Capitolo 3

Lo stile funzionale

3.1 Funzioni e procedure

3.1.1 Progettare con le operazioni del linguaggio

Impostare la risoluzione di un problema in termini di istruzioni ad un elaboratore assomiglia molto allo svolgimento di un gioco in cui le mosse possibili sono connesse alle istruzioni e ai meccanismi di combinazione del linguaggio di programmazione.

Anche se un progettista software ha la possibilità di decidere di introdurre nuove mosse e regole rispetto a quelle di base, molto spesso accade che il gioco si svolge nella mente delle persone avendo come riferimento le sole mosse elementari della macchina virtuale del linguaggio.

Tra gli elementi di base universalmente disponibili vi sono le strutture di controllo con cui esprimere cicli (`for`, `while`) e la capacità di denotare valori, variabili e oggetti predefiniti.

L'uso di oggetti JavaScript costituisce una buona palestra per comprendere la natura e la differenza tra le *variabili*, i *valori* e gli *oggetti* e l'influenza che l'uso di questi diversi elementi, unitamente alle convenzionali strutture di controllo, può avere sul modo di progettare e realizzare processi computazionali.

Consideriamo ad esempio il seguente problema:

Problema: VocaliMaiuscole

Data una stringa, fare in modo che tutte le vocali che vi compaiano siano lettere maiuscole

Supponiamo di denotare con la variabile `s` la stringa che rappresenta il dato del problema e di voler collaudare la soluzione rispetto alla stringa `s=abe`. In tal caso la risposta attesa è `Abe`.

Un progetto basato sulle mosse elementari disponibili

Come linguaggio di codifica scegliamo JavaScript, che costituisce una buona palestra con cui comprendere la natura e la differenza tra valori, variabili ed oggetti.

Nell'impostare la soluzione è essenziale introdurre variabili ed altri simboli in modo motivato, strettamente correlato al *ragionamento di progetto* e con il pieno controllo sul significato di quanto stiamo facendo e sul comportamento a tempo di esecuzione del sistema. Sottoponiamo quindi all'interprete JavaScript la frase

```
s = "abe";
```

dando ad essa la seguente interpretazione::

`s = 'abè;`: è intenzione del progettista denotare con il simbolo `s` il dato di ingresso del problema. Tecnicamente, `s` è il nome di una variabile destinata a rappresentare un qualsiasi valore JavaScript. La variabile viene legata in fase di definizione ad un valore di tipo stringa che ha come primo carattere `'a`', come secondo carattere ... etc.

In JavaScript le stringhe sono *tipi primitivi* che presentano all'utente un'interfaccia d'uso nello stile a oggetti. In particolare, JavaScript associa alle stringhe le seguenti proprietà ed operazioni:

- `s.length` restituisce il numero dei caratteri che compongono la stringa;
- `s.charAt(i)` restituisce il carattere di posizione `i`, con la convenzione che le posizioni siano numerate a partire da zero;
- `s.charCodeAt(i)` restituisce il valore UNICODE del carattere di posizione `i` (numero naturale).

NON VERO in virtù del comportamento della macchina virtuale del linguaggio, la variabile imperativa `s` è collegata (bound) a un *riferimento* all'oggetto stringa 'abè.

L'effetto della esecuzione della frase precedente è di estendere l'environment corrente con il simbolo `s`, come si constata sottoponendo all'interprete la frase `s`: ciò produce infatti come risposta la frase `'abè`. NON VERO L'interprete effettua dunque un *dereferenziamento automatico* di `s`.

Poichè JavaScript, come Java, considera le stringhe tipi strutturati *non modificabili*, la soluzione dovrà forzatamente seguire uno stile computazionale costruttivo-funzionale.

In altre parole, il linguaggio non permette di impostare la soluzione con l'idea di *modificare* la stringa, ma induce a costruirne un'altra, che rappresenta la risposta voluta.

Le mosse elementari disponibili per risolvere il problema sono le normali strutture di controllo e le operazioni rese disponibili sugli oggetti stringa.

Inoltre, poichè in JavaScript non esiste il tipo carattere, per rappresentare un singolo carattere occorre usare una stringa di lunghezza unitaria.

Il metodo `toUpperCase` applicato a una stringa restituisce la stringa in cui tutte le lettere sono in forma di maiuscola.

JavaScript: Esempi di uso di stringhe

```
s = "abe";
s.charAt(0);                      //vale "a"
"abe".charAt(0).toUpperCase();    //vale "A"
s.charAt(0).length;               //vale 1
"abe".length;                     //vale 3
for( i=0, s1=""; i < s.length; i++ )
      s1 = s1+s.charAt(i)+s.charAt(i).toUpperCase();
//s1 vale "aAbBeE"
```

è molto frequente imbattersi in programmi e sistemi software che introducono variabili inutili e in modo manifestamente non correlato ad alcun convincente ragionamento di progetto. Ad esempio, è pratica comune descrivere il significato di una frase (C, Java) quale `int x=2;` dicendo semplicemente: *il programma introduce l'intero x inizializzato a 2.* Se però ci si sforza di cercare un'interpretazione che ponga in luce il punto di vista del progettista, allora può essere più facile capire lo scopo per cui la variabile viene introdotta. Meglio quindi partire da una impostazione descrittiva del tipo:

`int x = 2:` è intenzione del progettista denotare con il simbolo `x` una variabile destinata a rappresentare *numeri interi* e di inizializzare detta variabile assegnando ad essa il valore `2`.

Analogamente, nel descrivere la frase JavaScript `var s = abc;` si dovrebbe dire:

`var s = abc;`: è intenzione del progettista denotare con il simbolo `s` una variabile destinata a rappresentare un qualunque valore JavaScritpt. La variabile è inizialmente legata ad un valore di tipo stringa che ha come primo carattere `'a'`, come secondo carattere `'b'`, etc,.

La non modificabilità delle stringhe induce a risolvere molti problemi adottando uno stile computazionale costruttivo-funzionale. Ad esempio, il problema:

Problema: *data una stringa, fare in modo che tutte le vocali che vi compaiano siano lettere maiuscole* ;

non può essere affrontato con l'idea di modificare la stringa, ma solo con l'idea di costruirne un'altra.

Sulla base delle mosse disponibili, il ragionamento di progetto potrebbe essere così descritto:

```
sia s la variabile che denota la stringa data;
sia s1 la variabile che denota il risultato,inizializzata a "";
per ogni carattere ch in s, esegui {
 se ch  non costituisce una vocale, allora
    concatena ch a s1;
 se invece ch  costituisce una vocale, allora
    concatena a s1 la maiuscola che corrisponde a ch
}
```

In termini di codice, questo ragionamento può essere espresso e realizzato da un ciclo:

```
s  = "abe"; // s denota la stringa di ingresso
s1 ="";     // s1 denota la stringa risultato
for( i=0, s1=""; i < s.length; i++ )
  switch( s.charCodeAt(i) ){
    case 97:
    case 101:
    case 105:
    case 111:
    case 117: s1 = s1+s.charAt(i).toUpperCase(); break;
    default : s1 = s1+s.charAt(i);
 } //s1 vale "AbE"
```

Questa realizzazione presenta due caratteristiche:

- sul piano progettuale, l'operatore `+` di concatenazione tra stringhe è usato come un *costruttore* della nuova stringa risultato;
- sul piano pratico, essa proietta su JavaScript le caratteristiche del linguaggio C in merito al costrutto *switch*, che vincola le etichette `case` ad essere numeri naturali.

JavaScript, però, a differenza del C, consente l'uso di stringhe come valori per le etichette `case`. Quindi il codice può essere riscritto in forma più leggibile come segue:

```
s = "abe"; //s denota la stringa di ingresso
s1; //s1 denota la stringa risultato
 for( i=0, s1=""; i < s.length; i++ )
  switch( s.charAt(i) ){
    case "a":
    case "e":
```

```
    case "i":
    case "o":
    case "u": s1 = s1+s.charAt(i).toUpperCase(); break;
    default : s1 = s1+s.charAt(i);
 } //s1 vale "abE"
```

JavaScript: Stringhe e valori non naturali come etichette case

Essendo JavaScript debolmente tipizzato, i valori per le etichette `case` possono essere valori anche di tipo diverso tra loro, da valutarsi in fase di compilazione. Ad esempio:

```
switch( obj ){
  case 0:
  case 10*2 :
  case "ettore":
  case "hello" + 2:
  default: "scelta finale";
}
```

Le varie versioni di codice basate su variabili e su mosse elementari del linguaggio non cambiano in modo significativo l'approccio mentale alla risoluzione del problema. Una linea di progettazione sostanzialmente diversa consiste invece nel porsi subito come obiettivo quello di elevare il livello di astrazione e di costruire un ente (componente) che possa risultare riutilizzabile sia all'interno dello stesso sistema software sia, in un secondo momento, per la risoluzione di altri problemi.

Per raggiungere questo scopo occorre abbandonare l'idea che risolvere un problema significhi eseguire una sequenza di comandi che agiscono su dati denotati da variabili, in favore del progetto e della realizzazione di *operazioni logiche* connesse al dominio del problema.

Questo significa fare ricorso a costrutti linguistici relativi alle funzioni (o alle procedure) e sfruttare appieno, in modo motivato e consapevole, metodologie di progetto e sviluppo bottom-up e/o top-down.

3.1.2 Il ruolo di funzioni e procedure

Da sempre lo spazio concettuale relativo alla computazione ha incluso due enti che risultano ancora oggi fondamentali per elevare il livello di astrazione nel progetto e nella costruzione del software: la procedura e la funzione.

- La **procedura** permette l'aggregazione di una sequenza di operazioni elementari in una unica macro-operazione dotata di nome e di argomenti. In quanto costrutto sintattico, la procedura costituisce un *blocco* e introduce regole di visibilità che nascondono ad un osservatore esterno la sua struttura interna. Semanticamente essa realizza i suoi effetti modificando parametri di uscita ricevuti dal chiamante e/o un ambiente esterno che può consistere in un insieme di variabili (non locali), dispositivi di I/O, files, etc.

- La **funzione** può essere vista come una procedura con restrizioni di tipo semantico: essa non effettua alcuna azione visibile sul mondo circostante, non modifica gli argomenti ricevuti dal chiamante e realizza i suoi effetti restituendo un risultato.

D'ora in avanti useremo il termine *funzione* per riferirci in modo generale a funzioni e procedure. Quando si tratterà di distinguere tra funzioni e procedure lo faremo esplicitamente.

Le funzioni nascono a livello hardware come meccanismo di salto incondizionato con salvataggio dell'indirizzo di ritorno (`call`), ripristinabile attraverso un apposita istruzione (`return`). La naturale esigenza di poter eseguire una `call` all'interno di un'altra `call` e la conseguente necessità di garantire politiche di ritorno `LIFO` (*Last In First Out*) ha indotto ad avvalersi di dispositivi di memorizzazione organizzati a pila (`stack`), assenti dai primi elaboratori, che sono stati poi realizzati direttamente in hardware, proprio per dare efficiente supporto alle funzioni.

Oggi infrastrutture più evolute permettono l'uso delle funzioni anche oltre lo spazio di indirizzamento di un singolo elaboratore. Java ad esempio fornisce in proposito il meccanismo **RMI** (*Remote Method Invocation*).

Le funzioni costituiscono una prima forma di componente software riusabile all'interno di una infrastruttura logicamente costituita dal programma principale. L'arma più potente che le funzioni offrono al costruttore di software è la possibilità di definire nuove operazioni *esplicitamente legate al dominio del problema*, che possono essere *combinate* in modi diversi per costruire nuove operazioni, in una successione di livelli senza fine.

Tornando al problema *VocaliMaiuscole*, si potrebbe introdurre una funzione facendo diventare il codice precedente il corpo di una funzione e avviluppandogli attorno un involucro costituito dalla intestazione della funzione (discuteremo la sintassi JavaScript nella sezione che segue):

```
function upCase( s ){
 // s denota la stringa da convertire
 var s1; // s1 denota la stringa risultato
```

```
 for( i=0, s1=""; i < s.length; i++ )
  switch( s.charAt(i) ){
    case "a":
    case "e":
    case "i":
    case "o":
    case "u": s1 = s1+s.charAt(i).toUpperCase(); break;
    default : s1 = s1+s.charAt(i);
 }
 return s1;
}
```

Questo modo di procedere, tuttavia, non coglierebbe l'opportunità di sfruttare funzioni come uno strumento di progetto prima ancora che di realizzazione. Prima di procedere in questa direzione, occorre approfondire il concetto di funzione in JavaScript, sia per apprendere una visione delle funzioni del tutto nuova rispetto ai linguaggi tradizionali, sia per continuare a svolgere con profitto la sperimentazione on-line.

3.1.3 Funzioni JavaScript come chiusure

Poichè le funzioni sono astrazioni di sequenze di istruzioni, molti linguaggi tendono a considerare questi costrutti come appartenenti ad una categoria diversa dai dati: di conseguenza, una funzione non può ricevere o restituire informazione costituita da altre funzioni.

In realtà questo vincolo si dimostra talmente stringente che alcuni linguaggi tentano di rilassarlo, proponendo meccanismi semanticamente non ben definiti. Il C permette ad esempio di scrivere:

```
int f( int v,  int property(int) ){
  if( property(v) ) return v*v;
  return v;
}//f
```

Questa funzione restituisce in generale lo stesso numero ricevuto in ingresso, tranne nel caso in cui esso possegga la proprietà verificata dalla funzione `property` ricevuta come argomento: in tal caso restituisce il quadrato del numero.

L'argomento `property` è una funzione-decisore, che compare in un contesto in cui ci si aspetta un dato.

Tecnicamente il parametro formale `property` è un *puntatore* al codice della funzione, ottenuto specificando il nome di una funzione (senza le parentesi, che denotano l'operatore di chiamata!) nel corrispondente argomento in ingresso. Ad esempio:

```
int pari( int x ){ return x%2 == 0;}
int dispari( int x ){ return x%2 != 0;}
int y = f( 2, pari );    // restituisce 4
int z = f( 2, dispari ); // restituisce 2
```

Si noti l'assenza, dopo il nome delle funzioni da trasferire come argomento, delle parentesi di chiamata `()`.

Mentre il trasferimento di una funzione a una funzione è in qualche modo possibile, non è previsto dal C che una funzione possa restituire una funzione. D'altra parte, quest'ultimo concetto sembra quanto meno astruso, visto che non è chiaro con quali ingredienti e in quale modo si potrebbe costruire una funzione per fornirla poi come risultato di un'elaborazione.

La risposta a questa domanda la fornisce JavaScript, che rivela in questo la forte influenza ricevuta dai linguaggi funzionali.

Funzioni in JavaScript

JavaScript introduce una istruzione di **definizione** di funzioni che assume la forma desumibile dall'esempio che segue:

```
f = function( x, y ){ return x+y;}
```

La frase va interpretata come segue:

`f = function( x, y ){ return x+y;}`:

è intenzione del progettista denotare con il simbolo `f` una variabile destinata a rappresentare un qualunque oggetto JavaScritpt. La variabile è inizialmente legata a (un riferimento a) un oggetto di tipo funzione.

La valutazione di espressioni che includono il letterale `function` restituisce un ente (oggetto) computazionale di tipo funzione, privo di nome, i cui argomenti sono `x` ed `y`, e il cui corpo è `{return x+y;}`. Infatti:

```
f;          // restituisce la rappresentazione esterna della funzione
typeof(f);  // restituisce function
```

La **chiamata** avviene attraverso l'operatore `()` posto dopo un oggetto funzione. Ad esempio:

```
(function( x, y ){ return x+y;}(3,2)); //valutata, restituisce 5
f(3,2); //valutata, restituisce 5
```

Si noti, nel primo caso, la necessità di racchiudere tutta la frase tra parentesi, per esprimere il fatto che si desidera valutare l'applicazione dell'oggetto funzione alla coppia di argomenti (3,2). Qualora tali parentesi fossero omesse, come in:

```
function( x, y ){ return x+y;}(3,2); //non  come prima!
```

si otterrebbe una valutazione completamente diversa, poichè verrebbero valutati, nell'ordine, l'oggetto funzione, il primo elemento della coppia (3,2) e il secondo elemento di tale coppia, visualizzando solo l'ultimo risultato.

Chiusure

Il risultato di espressioni `function` è semanticamente una **chiusura**, cioè un oggetto computazionale che associa un'espressione testuale ad un ambiente di definizione dei simboli. Si parla di *chiusura lessicale* quando l'ambiente che si considera è quello in cui compare la *definizione* dell'espressione. Si parla di *chiusura dinamica* quando l'ambiente che si considera è quello in cui compare l'uso (la valutazione) dell'espressione.

Interpretare una funzione come una *chiusura lessicale*, significa che si crea un legame tra l'espressione che denota il corpo della funzione e l'ambiente presente nel momento in cui la funzione *è definita*. Consideriamo il seguente esempio:

JavaScript: Definizioni annidate di funzioni

```
g = function( x, y ){ return function(){return x+y;}}
v = g(2,3); //Restituisce una chiusura lessicale;
x = 10;
y = 20;
v(); //Restituisce 5;
```

La chiamata `g(2,3)` restituisce la chiusura lessicale in cui l'ambiente (che denoteremo con *env*) è costituito dall'insieme dei legami *env*=`{(x,2),(y,3)}`. La chiamata `v()` invoca la chiusura denotata dalla variabile `v`: in risposta, la macchina virtuale Javascript esegue il corpo `return x+y` nell'ambiente *env*, ignorando i valori di `x` e `y` definiti nell'ambiente globale.

ho eliminato l'espressione di prima classe perchè è da sempre fuorviante: non aggiunge significato e fa solo nascere dubbi in chi legge (esistono funzioni

di seconda classe? da dove esce questa classificazione? cosa si intende per 'classè? ha a che fare con la 'classè dei linguaggi a oggetti?...)

In quanto oggetti, le funzioni JavaScript possono essere inserite in strutture dati, essere trasferite come argomenti ad altre funzioni e restituite come valori, aprendo nuove tecniche di progetto che discuteremo in una sezione apposita.

Per il momento ci atterremmo alle proprietà delle funzioni che sono comuni a tutti i linguaggi di programmazione.

L'istruzione function

Come zucchero sintattico JavaScript introduce anche la possibilità di scrivere:

```
function f( x, y ){ return x+y; };
```

Questa frase corrisponde alla definizione di una funzione di nome `f` con argomenti `x`,`y` e corpo `{return x+y;}`.

Volendo usare questa forma **nell'ambiente di prova**, si deve tenere presente che essa *non provoca effetti collaterali nell'ambiente globale*: perciò, occorre che l'uso segua subito la definizione, come nell'esempio che segue:

```
function f( x, y ){ return x+y; }; f(2,3);
```

3.1.4 Progettare in termini di operazioni astratte

Impostare la soluzione di un problema in termini di funzioni permette di focalizzare l'attenzione sulle *operazioni logiche del dominio*, piuttosto che sulle mosse elementari rese disponibili dalla macchina o dal linguaggio. Ad esempio la decisione sulla natura di un carattere (di essere una vocale o meno) può essere incapsulata entro una funzione:

```
vocale = function( ch ){
 return (ch=="a" || ch=="e" || ch=="i" || ch=="o" || ch=="u");
}
```

Di conseguenza il codice della funzione risolvente il problema *Vocali-Maiuscole* diventa molto più leggibile:

```
upCase = function( s ){
//s denota la stringa da convertire
var s1; //denota la stringa risultato
 for( i=0, s1=""; i < s.length; i++ ){
  if( vocale( s.charAt(i) ) ) s1 = s1+s.charAt(i).toUpperCase();
  else s1 = s1+s.charAt(i);
```

```
 }
 return s1;
}
```

Inoltre, nel contesto di JavaScript, è immediato rendere parametrica la funzione risolvente rispetto al tipo di carattere:

```
upCase = function( s, property ){
//s denota la stringa da convertire
var s1; //denota la stringa risultato
 for( i=0, s1=""; i < s.length; i++ ){
  if( property( s.charAt(i) ) ) s1 = s1+s.charAt(i).toUpperCase();
  else s1 = s1+s.charAt(i);
 }
 return s1;
};
```

Come risultato, l'invocazione della funzione `upCase` con decisore `vocale` assume la forma seguente:

```
upCase("abcde",vocale); //restituisce "AbcdE"
```

Il contributo più importante connesso all'uso delle funzioni è però legato alla possibilità di impostare il ragionamento risolutivo in modo radicalmente diverso, spostando l'attenzione dal funzionamento interno alla interfaccia esterna che la funzione deve presentare ai suoi clienti. Questa interfaccia è costituita dalla *intestazione* della funzione risolvente.

Un progetto basato su ricorsione

Astraendo per il momento dal linguaggio di codifica, definiamo l'intestazione della funzione risolvente nel modo che segue:

```
 String upCase( String s, int cont )
```

denotando con `s` la stringa di ingresso e con `cont` il numero di caratteri già esaminati.

La specifica dell'intestazione ci permette di ragionare sulla soluzione senza ancora avere deciso come sarà il funzionamento interno e induce a pianificare situazioni interessanti per il collaudo del prodotto finale.

Ad esempio:

la frase `upCase('abcdè, 0)` dovrà denotare la stringa `'Abcdè`;

la frase `upCase('abcdè, 3)` dovrà denotare la stringa `'dè`.

Si intuisce che la variabile di ingresso `cont` rende il componente più flessibile, dando al chiamante la possibilità di decidere il punto iniziale dell'elaborazione della stringa.

Una volta disponibile l'interfaccia d'uso della funzione che risolve il problema, si apre la possibilità di utilizzare potenti tecniche di ragionamento ricorsivo, in cui il progettista riusa, per risolvere il problema, la funzione stessa che sta costruendo.

In JavaScript, seguendo tecniche di progettazione che discuteremo nella sezione **??**, pag. **??**

ne potrebbe scaturire il codice che segue:

sembra un pò buttato lì... Non c'è una sola parola di commento. Si postula che la ricorsione sia già nota come tecnica?

```
upCase = function(  s,  cont ){
//s denota la stringa di ingresso
//cont denota il numero dei caratteri gi esaminati
if (s.length == cont) return "";
if( vocale( s.charAt(cont) ) )
 return s.charAt(cont).toUpperCase() + upCase(s,++cont);
else  return s.charAt(cont) + upCase(s,++cont);
}
```

Un progetto iterativo

Un progettista abituato ad uno stile funzionale o un programmatore Prolog avrebbero molto probabilmente impostato la soluzione partendo da una intestazione della funzione risolvente della forma:

```
String upCase( String s, int cont, String  sOut)
```

ove `s` denota la stringa di ingresso, `cont` il numero di caratteri esaminati e `sOut` la stringa di uscita relativa alla parte già elaborata (i primi `cont` caratteri) della stringa di ingresso (dal carattere di indice `0` al carattere di indice `cont-1`). In tal caso:

la frase `upCase('abcdè, 0, '')` deve denotare la stringa `'Abcdè`;
la frase `upCase('abcdè, 0, 'X')` deve denotare la stringa `'XAbcdè`;
la frase `upCase('abcdè, 3, 'Abc')` deve denotare la stringa `'Abcdè`
la frase `upCase('abcdè, 4, 'à)` deve denotare la stringa `'Aè`.

In JavaScript questo nuovo componente potrebbe essere costruito come segue:

```
upCase = function(  s,  cont,  sOut ){
//s: la stringa di ingresso
//cont: numero dei caratteri gi esaminati
//sOut: stringa di uscita per sottostringa s da 1 a cont
```

```
if (s.length == cont) return sOut;
if( vocale( s.charAt(cont) ) )
  return  upCase(s, cont+1, sOut+s.charAt(cont).toUpperCase());
else return upCase(s, cont+1, sOut+s.charAt(cont));
}
```

Di fronte a queste diverse soluzioni sorgono immediatamente alcune domande:

- Qual è il ragionamento che porta alle diverse soluzioni?
- Cosa accade a tempo di esecuzione?
- Qual è il costo computazionale delle diverse soluzioni? In particolare, qual'è la più efficiente?
- Perchè l'ultima soluzione è stata definita come una soluzione iterativa quando la funzione invoca palesemente sè stessa, secondo la tipica struttura ricorsiva?

Queste domande sono talmente rilevanti da richiedere di approfondire il tema in modo specifico.

3.1.5 Contratti, funzionamento e struttura

Strutturalmente una funzione è costituita da due parti: una intestazione (*signature*) e un corpo (*body*).

Signature di una funzione
Per *signature* di una funzione (o di una procedura) si intende l'insieme formato dal nome e dalla lista (numero e tipo) degli argomenti. Di norma il tipo del valore restituito non viene considerato parte della signature.

La signature di una funzione rappresenta la specifica parziale di un contratto tra il cliente (chiamante) e il servitore (funzione) che permette di catturare importanti vincoli connessi ad un uso corretto del servitore o alla realizzazione corretta della funzione stessa.

La specifica è solo parziale in quanto molte possibili violazioni, come l'insorgere di eventi anomali (esaurimento della memoria, risorse non disponibili, etc) o il mancato soddisfacimento di protocolli d'uso può venire rilevato solo a tempo di esecuzione: di solito queste violazioni sono gestite in modo da provocare la terminazione immediata del programma. Linguaggi come C++ e Java permettono che una funzione possa terminare lanciando una

eccezione. Esamineremo in una delle prossime sezioni le eccezioni come meccanismo e come concetto in relazione alla struttura e al processo di sviluppo del software.

Le funzioni costituiscono il primo esempio di ente computazionale parametrico, modulare e riusabile, di rilevante importanza anche per le moderne esigenze di produzione del software. Infatti:

- la definizione di una funzione corrisponde alla definizione di un *servitore sequenziale e passivo*, che può introdurre e gestire informazione in modo non accessibile ai clienti;
- le funzioni costituiscono componenti software riusabili, che possono essere costruiti per composizione di altre funzioni;
- una funzione presenta una chiara distinzione tra interfaccia e struttura interna;
- soluzioni espresse in termini puramente funzionali non possono provocare erori dovuti ad effetti collaterali in quanto godono della proprietà nota come **trasparenza referenziale**.

Trasparenza referenziale

La trasparenza referenziale è una proprietà legata all'uso di simboli per denotare informazione e consiste nel fatto che un dato simbolo denota sempre lo stesso valore ogni volta in cui compare in data espressione. Si consideri ad esempio l'espressione:

`f(x) + g(f(x), q(x + f(y)))`

Facendo appello alla comune cultura matematica, è ragionevole aspettarsi che:

- `f,g,q` siano simboli che denotano funzioni;
- il simbolo `x` denoti un valore (di tipo numerico o di altro tipo);
- qualunque sia il valore denotato da `x, x` denoti sempre lo stesso valore in ogni punto in cui compare nell'espressione;
- l'espressione `f(x)` denoti sempre lo stesso valore in ogni punto in cui compare nell'espressione.

Pur non sapendo nulla di cosa fanno le funzioni di nome `f, g, q`, ci aspettiamo che l'espressione precedente sia del tutto equivalente a quella che segue:

`a + g(a, q(x + b))`, essendo `a=f(x), b=f(y)`

Analogamente, ci aspettiamo che **non** sia corretto dire che l'espressione precedente equivalga alla seguente:

`a + g(a, q(x + a))`, essendo `a=f(x)`

Infatti, in generale, ci aspettiamo che il valore `f` in corrispondenza al valore denotato dal simbolo `y` sia diverso dal valore di `f` in corrispondenza al valore denotato dal simbolo `x`.

L'interazione tra un cliente e il servitore rappresentato dalla funzione comporta:

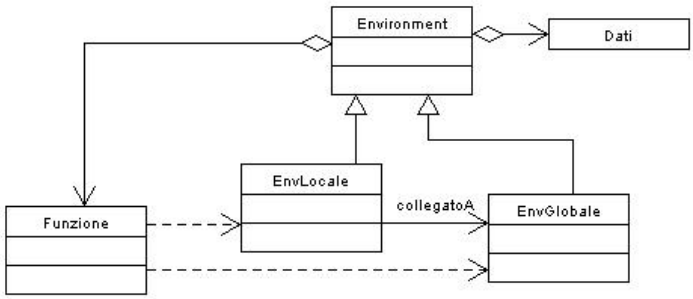
- trasferimento del controllo dal cliente al servitore;
- trasferimento esplicito di informazione (per copia o per riferimento);
- trasferimento implicito di informazione (attraverso un ambiente condiviso).

Il trasferimento di controllo avviene attraverso istruzioni di invocazione della funzione. Il trasferimento esplicito di informazione avviene attraverso gli argomenti definiti nella intestazione della funzione, che fungono da *parametri formali*, cui vengono messi in corrispondenza *valori attuali* al momento della chiamata.

I parametri formali possono ricevere dati (nel qual caso il trasferimento avviene *per copia*) o indirizzi di memoria in cui sono contenuti i dati (nel qual caso il trasferimento avviene *per riferimento*).

Parlare di funzioni (e procedure) significa introdurre implicitamente nel discorso due livelli di organizzazione: l'organizzazione lessicale e l'organizzazione a tempo di esecuzione. In particolare si dice *ambiente lessicale* l'ambiente in cui compare la definizione testuale di una funzione, mentre si dice *ambiente dinamico* l'ambiente in essere al momento di una particolare attivazione della funzione.

Funzioni: organizzazione dell'ambiente dinamico



Nel modello l'entità *Funzione* rappresenta un'istanza attiva.

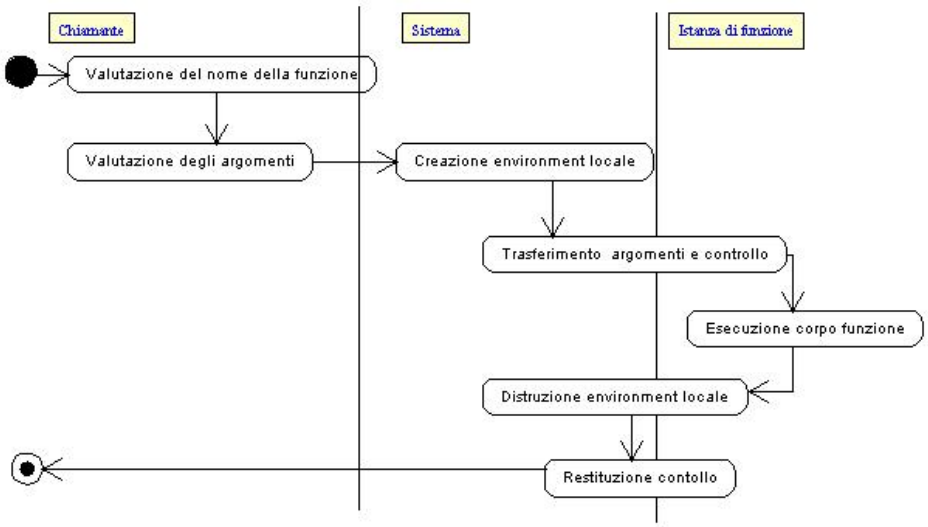
3.1.6 Il modello di valutazione applicativo

Il comportamento a tempo di esecuzione di una funzione segue il **modello applicativo**, che prevede *concettualmente* le seguenti attività:

1. Valutazione, nell'environment lessicale, del simbolo che denota il nome della funzione.
2. Valutazione, nell'environment lessicale, delle espressioni che denotano gli argomenti.
3. Commutazione all'environment di definizione della funzione.
4. Trasferimento degli argomenti e del controllo alla funzione.
5. Esecuzione del corpo della funzione.
6. Restituzione al chiamante sia del controllo sia del risultato, con ripristino dell'environment esistente al momento della chiamata.

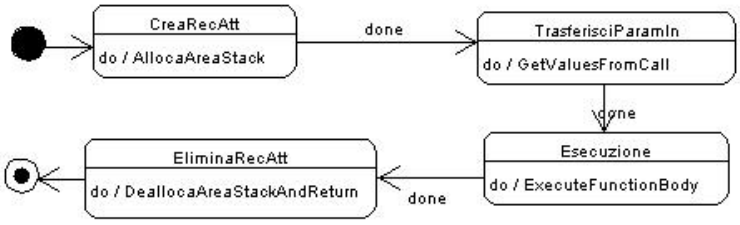
Queste attività si possono rappresentare con il seguente *activity diagram* UML, in cui le *swimlines* relative a `chiamante`, `sistema` e `istanza di funzione` servono per chiarire il contesto logico di riferimento, anche se alcuni compiti sono distribuiti tra le parti.

Modello applicativo



Lo *state diagram* UML che segue pone in luce alcuni stati significativi della macchina virtuale del linguaggio in relazione alle azioni precedentemente illustrate.

Funzioni: comportamento del sistema



Lo stato *creaRecAtt* costruisce sullo stack un **record di attivazione**, cioè una struttura di dati che corrisponde all'environment locale. Il collegamento con l'environment globale può essere esplicito o implicito.

Il diagramma non include stati connessi al trasferimento di parametri in uscita in quanto i valori di ritorno sono mappati in un registro macchina. Una funzione può anche agire direttamente nell'area di memoria del chiamante attraverso i puntatori o riferimenti ricevuti come argomenti di ingresso.

La funzione in esecuzione in un certo istante ha sempre il proprio record di attivazione (RA) in cima allo stack. Funzioni definite ma non invocate non occupano memoria nello stack, ma solo nell'area codice.

Record di attivazione e accesso all'ambiente
Il record di attivazione (**RA**) di una funzione:

- forma l'environment di lavoro della funzione come estensione dell'environment di definizione.

  Nel caso il linguaggio, come accade per JavaScript, permetta definizioni innestate di funzioni, l'environment si può articolare in più aree dati (dette anche **frame**) collegate tra loro attraverso una catena di puntatori, detta *catena statica*: ognuno di questi puntatori è memorizzato in un campo del frame detto *static link*. Quando, viceversa, l'environment di definizione di una funzione è costituito dall'insieme dei simboli globali definiti testualmente prima di essa (ossia, non sono consentite definizioni innestate di funzioni), come in C, lo static link non è necessario.

- tiene traccia del punto del codice chiamante cui restituire il controllo al termine della esecuzione della funzione. A questo fine utilizza un campo detto *return address*. Ponendo nel Program Counter il contenuto di questo campo, la macchina virtuale può restituire il controllo, al termine della esecuzione di una funzione o procedura, alla istruzione successiva del cliente che ne aveva provocato l'attivazione.

- contiene l'allocazione dei parametri e le variabili locali relativi alla particolare attivazione della funzione.

Le *variabili globali* sono di norma referenziate attraverso indirizzi assoluti, determinati prima della esecuzione del programma. In sistemi a memoria segmentata, esse sono allocate nel segmento-dati (*data-segment*).

Le variabili locali ad una procedura o funzione sono referenziate mediante un indirizzamento a `base + offset`, ove il valore della base è l'indirizzo (noto solo a tempo di esecuzione) del RA, mentre l'offset è noto a tempo di compilazione.

Nei linguaggi in cui sono possibili definizione innestate di funzioni, il riferimento alle variabili non locali è più complesso: una tecnica consiste nel risalire la catena statica di un opportuno numero di livelli. è importante comunque sottolineare come non sia necessaria alcuna ricerca dinamica, in quanto il compilatore può stabilire staticamente di quanti livelli occorre risalire per trovare la cella di memoria relativa ad una specifica variabile.

In base al modello applicativo, il processo di valutazione della espressione

```
2+3.0*sqr(1.0)-x
```

si articola come segue:

1. suddivisione della frase nei suoi enti costitutivi, individuati nelle seguenti parti:

   `2` simbolo che denota una costante costituita dal numero intero *due*

   `+` simbolo che denota un operatore binario

   `3.0` simbolo che denota una costante costituita dal numero reale *tre*

   `sqr` simbolo che denota il nome di una funzione

   `(` simbolo che denota una parentesi (aperta)

   `1.0` simbolo che denota una costante costituita dal numero reale *uno*

   `)` simbolo che denota una parentesi (chiusa)

   `-` simbolo che denota un operatore binario

   `x` simbolo che denota il nome di una variabile.

   Questa scomposizione viene effettuata da un *analizzatore lessicale* che opera sulla base delle descrizione della struttura sintattica del linguaggio definita da una grammatica *context-free*.

2. analisi del significato dei simboli `+`,`*`,`-`,`sqr` e `x`. Pur sapendo che `+`,`*`,`-` sono operatori binari, l'elaboratore potrebbe legare ciascuno di essi a una versione diversa del codice, in funzione del tipo degli argomenti (ad esempio la funzione di somma tra due interi potrebbe essere diversa dalla funzione di somma tra due reali).

3. valutazione della frase seguendo il modello applicativo:

   1) valutazione di `1.0` con risultato `r1` (numero reale)

   2) applicazione di `sqr` a `r1` con risultato `r2` (numero reale)

   3) valutazione di `3.0` con risultato `r3` (numero reale)

   4) valutazione di `2` con risultato `r4` (numero intero)

   5) conversione di `r4` in reale con risultato `r5` (numero reale)

   6) applicazione di `+` (somma tra reali) a `r3` e `r5` con risultato `r6` (numero reale)

7) valutazione di `x` con risultato `r7`

9) applicazione di `-` (differenza tra reali) a `r6` e `r7` con risultato `r8`

Naturalmente, affinchè la valutazione sia completata con successo, il simbolo `x` deve essere definito nell'environment corrente.

3.1.7 Trasferimento degli argomenti

Il modello applicativo è ormai adottato da tutti i linguaggi di programmazione, anche se sono stati studiati e sperimentati altri meccanismi, soprattutto in relazione al trasferimento dei parametri. I meccanismi fondamentali di trasferimento nel modello applicativo sono due:

- il trasferimento **per valore** (*Call-by-value*): prevede la valutazione della espressione che corrisponde ad un parametro attuale e l'associazione (binding) di una *copia* del valore che ne risulta al corrispondente parametro formale;

- il trasferimento **per riferimento** (*Call-by-reference*): prevede che al parametro formale venga sia associato (bound) un *riferimento* al valore associato al corrispondente parametro attuale. Se al parametro attuale non è associabile alcun riferimento (ad esempio nel caso sia una costante) si ha un inconsistenza e si genera un errore, di solito rilevato a tempo di compilazione.

Esempi di passaggio dei parametri
Consideriamo una funzione `succ` che dato un intero, ne restituisce il successore:

```
 int succ(int n){return n+1;}
int v = 2+3;
int next = succ(v);
```

Nella *call-by-value*, il parametro formale `n` viene legato al valore 5 (*r-value* di x).
Nella *call-by-reference*, il parametro formale `n` viene legato allo *l-value* di x. Per restituire `6`, la funzione, nel valutare l'espressione `n+1`, dovrebbe effettuare un dereferenziamento automatico di `n`. In C sarebbe necessario usare l'operatore `*`.

Il trasferimento per riferimento dà ad una funzione-servitore la possibilità di modificare direttamente l'ambiente del cliente. Il trasferimento per valore è intrinsecamente più sicuro, ma poco efficiente nel caso di dati voluminosi.

Un terzo modo per trasferire parametri a funzioni è costituito dal trasferimento per **valore-risultato** (*Call-by-value-result*), che si basa sulla trasmissione per valore *al momento della chiamata*; al termine della esecuzione della funzione, i valori correnti dei parametri formali vengono assegnati alle variabili che denotano i corrispondenti parametri attuali.

Questo meccanismo, ormai scomparso dai linguaggi di programmazione, sta tornando in auge per le comunicazioni con servizi di rete, ad esempio nei *Web Services*.

Trasferimento per valore-risultato

Nella *Call-by-value-result*, anche detto meccanismo *copy-in/copy-out*, il caso dell'esempio precedente si articolerebbe come segue:

- *Fase di chiamata*: l'r-value di `v` viene trasferito a `n`, mentre lo l-value di `v` viene contemporaneamente memorizzato all'interno della funzione;
- *Fase di ritorno al chiamante*: il valore corrente di `n` viene copiato in `v`, sfruttando all'uopo lo l-value memorizzato in precedenza.

Questo modello permette un maggiore disaccoppiamento rispetto al passaggio per copia, in quanto il risultato calcolato viene copiato in `v`, sfruttandone lo l-value, solo *al termine* dell'elaborazione.

Ciò comporta, in particolare, che tutti i valori intermedi eventualmente assunti **non** siano riflessi in `v`, che passerà quindi direttamente dal valore iniziale al valore finale.

Questo non è vero nel trasfierimento per indirizzo, in cui ogni singola azione del chiamato avviene sulla variabile del chiamante.

Se il codice della funzione fosse:

```
int succ(int n){n=n+1; return n;}
```

nella *Call-by-reference* e nella *Call-by-value-result* il valore della variabile `v` del chiamante verrebbe modificato a `6` (come pure, ovviamente, la variabile `next` del chiamante).

Al contrario, nella call-by-value il valore della variabile `v` del chiamante rimarrebbe `5`, poichè le modifiche al parametro `n` del chiamato sarebbero soltanto locali; in tal caso il risultato 6 verrebbe attribuito esclusivamente alla variabile `next` del chiamante.

Il meccanismo Call-by-name

Altri meccanismi di trasferimento derivano da modelli di valutazione diversi dal modello applicativo. Il modello di valutazione cosiddetto normale (**normal-order evaluation**) prevede che la determinazione degli argomenti di una funzione avvenga sostituendo ai parametri *sostituzioni testuali* delle espressioni che denotano i valori attuali:

nel corso della computazione queste espressioni vengono poi ridotte a valori atomici e primitivi solo *quando necessario*.

La sostituzione testuale di un argomento del modello normale (detta anche **Call-by-name**) non implica la valutazione dell'argomento ma solo la trasmissione alla procedura di un opportuno oggetto computazionale.

Nella realizzazione della *call-by-name* dell'Algol60 questo oggetto è detto **thunk** e correla l'espressione che corrisponde all'argomento con il suo ambiente di valutazione, come una *chiusura* (si veda sezione 3.1.3, pag. 57).

Si consideri ad esempio la seguente applicazione:

```
ftry = function(a, b){return (a==0) ? 1 : b;}
x = 1;
y = 2;
c = ftry(x,y);
```

Semanticamente la *call-by-name* opera come se scrivessimo il codice che segue:

```
ftry = function(a, b){return (a()==0) ? 1 : b();}
x = 1;
y = 2;
c = ftry( function(){return x;}, function(){return y;} );
```

I parametri attuali sono chiusure a zero argomenti: la valutazione di un argomento avviene perciò solo se e quando esso viene *effettivamente usato* dalla procedura chiamata (valutazione ritardata o **lazy evaluation**). Ad esempio, nel caso dell'invocazione di `ftry(x,y)` con `x=0` e `y=2`, il valore dell'argomento `b` non verrebbe valutato.

In molte situazioni il modello normale e quello applicativo forniscono gli stessi risultati: è il caso dell'esempio precedente, `ftry(1,2)`. Vi sono però situazioni in cui la valutazione immediata dell'argomento tipica del modello applicativo (detta anche *eager evaluation*) porta alla impossibilità di continuare un'elaborazione che, invece, terminerebbe con successo nel caso ritardato. Consideriamo a tale proposito la chiamata:

```
x = 0;
c =ftry( x, 1/x );
```

Il modello normale restituisce il valore `1`, poichè il secondo argomento (la cui valutazione dà luogo a errore) non viene valutato. Viceversa, con il modello applicativo il corpo della funzione non viene nemmeno eseguito, poichè la valutazione di `1/x` porta ad un errore di `division by zero` a tempo di esecuzione.

Il modello normale permette quindi di esprimere un insieme di programmi che terminano con successo *più ampio* dell'insieme corrispondente del modello applicativo.

3.1.8 Eccezioni

La signature di un'operazione rappresenta la specifica solo parziale di un contratto: in generale infatti non è possibile catturare a livello di signature tutti i vincoli connessi all'uso corretto di una funzione da parte dei clienti o alla realizzazione corretta della funzione. Molte violazioni, come l'insorgere di eventi anomali (esaurimento della memoria, risorse non disponibili, etc) o il mancato soddisfacimento di regole d'uso, vengono catturate solo a tempo di esecuzione e sono normalmente gestite in modo da provocare la terminazione immediata del programma.

I meccanismi di gestione delle eccezioni (*exception handling*) sono stati introdotti nei linguaggi C++, Java, C# e JavaScript per permettere di continuare l'esecuzione nonostante il verificarsi di segnali di errore.

Sul piano concettuale le eccezioni ampliano lo spazio di progettazione in quanto introducono l'idea di *anomalia* e di *gestione degli eventi* che segnalano tali anomalie. Una eccezione è la manifestazione della violazione di un vincolo implicitamente od esplicitamente posto nel sistema o in una sua parte e non un convenzionale meccanismo di controllo. La gestione di una eccezione può avvenire in accordo a uno dei seguenti due modelli di comportamento:

- il *modello a resumption*, che prevede di procedere in modo simile alla ricezione di una *interruzione*: il flusso normale di controllo è sospeso per eseguire l'intervento di gestione dell'evento e poi riprende;
- il *modello a termination*, che prevede di interrompere il normale flusso di controllo e trasferirlo a un gestore di eccezioni (*exception handler*) senza più tornare indietro.

Dal punto di vista delle politiche di gestione si può:

- tentare di ricondurre il sistema ad uno stato precedente corretto per poi ripartire (*backward error recovery*) . Questo schema richiede opportuni supporti a run-time ed è poco diffuso nei linguaggi di programmazione, mentre viene adottato (spesso all'insaputa dell'utente) nei sistemi di gestione delle basi dati, per la parte di *supporto alle transazioni*.
- tentare di eseguire un'operazione di correzione dello stato erroneo per poi proseguire (*forward error recovery*) .

Tutti i linguaggi sopra citati seguono il modello a terminazione ipotizzando che l'*exception handler* imposti una politica di *forward error recovery*. L'operazione che ha causato (o subito) l'eccezione termina sempre, cedendo alla fine il controllo al cliente che può catturarla nel caso in cui abbia impostato la chiamata nel modo che segue:

```
 try {
   //azione che pu sollevare eccezioni;
}catch( tipoEccezione1 e){ //tipoEccezione1 specializza Exception
    //gestione della eccezione
}catch( tipoEccezione2 e) { //tipoEccezione2 specializza Exception
   //gestione della eccezione
}
```

In questo e negli esempi successivi faremo riferimento al linguaggio Java, tenendo conto che C++ e C# sono nella sostanza identici. JavaScript ha invece introdotto le eccezioni solo nelle ultime versioni e presenta alcune inevitabili differenze rispetto agli altri linguaggi a causa della mancanza di dichiarazioni di tipo.

In Java le eccezioni sono oggetti, istanze di classi appartenenti a una gerarchia di *exception classes* che ha come radice la classe `java.lang.Throwable` (vedi **Material/Tools/jdk1.2.1-docs/api/java/lang/Throwable.html**) .

Per definire una nuova eccezione, di solito si introduce però una classe con la seguente intestazione:

```
public class ExceptionType extends Exception
```

essendo `java.lang.Exception` (vedi **Material/Tools/jdk1.2.1-docs/api/java/lang/Exception.html**) una specializzazione di `Throwable`.

Java distingue tra eccezioni *checked* e *unchecked*. Il compilatore infatti di solito controlla che un metodo lanci solo quelle eccezioni che ha dichiarato di lanciare. Questo controllo non viene però effettuato per tutte le eccezioni della classe `java.lang.RuntimeException` (vedi **Material/Tools/jdk1.2.1-docs/api/java/lang/RuntimeException.html**) o di classi da essa derivate.

In Java (come in C++) un'eccezione può venire generata esplicitamente attraverso l'operatore:

```
 throw oggetto
//oggetto deve essere di tipo compatibile con Exception
```

La gestione della eccezione segue la metafora *lancia un oggetto e cattura un tipo* e consiste nella esecuzione delle azioni definite dal lato cliente che corrispondono al corpo della *parte catch* che ha un argomento dello stesso tipo (o compatibile) con quello dell'oggetto specificato dalla *throw*. Ad esempio, nel codice che segue:

```
try {
   ...;
```

```
}catch( InconsistencyException e  ){
       ...
}catch( Exception e  ){
       ...
}
```

la seconda clausola *catch* cattura eccezioni di un qualsiasi tipo. Una clausola come questa deve sempre comparire per ultima, in quanto il sistema pone in esecuzione il gestore relativo alla prima clausola *catch* che cattura un oggetto di tipo compatibile con l'eccezione: dunque esso si ferma comunque alla clausola *catch(Exception e)* rendendo inutile ogni clausola successiva.

Nel realizzare questo modello di gestione, il sistema effettua automaticamente lo *stack unwinding*, cioè dealloca i record di attivazione delle operazioni attive che non catturano l'eccezione fino a giungere alla prima che lo fa – o al cliente.

C++ In C++ durante lo stack unwinding vengono anche chiamati i distruttori degli oggetti in vita sullo stack.

Se un'operazione *f1* invoca, senza racchiudere la chiamata nel costrutto *try/catch*, un'operazione *f2* che può sollevare un'eccezione di tipo *ExceptionType*, *f1* ha l'obbligo di specificare nella sua signature la frase `throws ExceptionType`, per segnalare ai suoi clienti che, a sua volta, può essere una sorgente di anomalie di quel tipo.

Esempio

La versione ricorsiva a tre argomenti della funzione `upCase` che risolve il problema `VocaliMaiuscole` presuppone che gli argomenti di ingresso siano il relazione tra loro:

```
upCase = function(  s,  cont,  sOut ){
//s: la stringa di ingresso
//cont: numero dei caratteri gi esaminati
//sOut: stringa di uscita per sottostringa s da 1 a cont
```

Infatti, il corpo della funzione non può fornire il risultato atteso se tra gli argomenti di ingresso non è soddisfatto il vincolo:

```
 (sOut.length<= s.length)&&(sOut.length==cont)
```

Il codice della funzione può cautelarsi su invocazioni non corrette lanciando una eccezione.

```
/**
 * Conversione vocali in maiuscole, con eccezioni
```

```
 * Versione ricorsiva tail a tre argomenti.
 *-+-param s = stringa da convertire
 *-+-param cont = numero dei caratteri gi esaminati
 *-+-param sOut = stringa di uscita per sottostringa s da 1 a cont
 *-+-return s con la vocali in maiuscolo
 */
public static String upCaseConstraint(String s,int cont,String sOut)
         throws Exception{
 if (! ((sOut.length()<=s.length()) && (sOut.length()==cont)))
    throw new Exception("wrong call");
 if (s.length() == cont) return sOut;
 if( vocale( s.charAt(cont) ) )
  return
   upCaseConstraint(s,cont+1,sOut+Character.toUpperCase(s.charAt(cont)));
 else return
  upCaseConstraint(s,cont+1,sOut+s.charAt(cont));
}
```

Eccezioni in JavaScript

Per effettuare un esperimento in JavaScript possiamo procedere come in Java senza l'obbligo di dichiarare l'eccezione nella intestazione della funzione. Inoltre la metafora *lancia un oggetto e cattura un tipo* non è evidentemente applicabile, data la natura debolmente tipizzata del linguaggio: ne segue che la clausola `catch` deve essere unica. L'esperimento è utile per quanto riguarda il comportamento, ma mostra le carenze di JavaScript come strumento per organizzare la parte di progetto.

```
upCaseConstraint = function(  s,  cont,  sOut ){
//s: la stringa di ingresso
//cont: numero dei caratteri gi esaminati
//sOut: stringa di uscita per sottostringa s da 1 a cont
if (! ((sOut.length <= s.length) && (sOut.length== cont)) )
   throw "wrong call";
if (s.length == cont) return sOut;
if( vocale( s.charAt(cont) ) )
 return upCaseConstraint(s,cont+1,sOut+s.charAt(cont).toUpperCase());
else return upCaseConstraint(s,cont+1,sOut+s.charAt(cont));
}
```

Il chiamante dovrà ora impostare l'invocazione a `upCase` usando il costrutto `try/catch`:

```
try{  res = upCaseConstraint("abcde",3,"Abc");
}catch(e){  res = e; } //Restituisce "AbcdE"
try{  res = upCaseConstraint("abcde",4,"Abc");
}catch(e){  res = e; } //Restituisce 'wrong call'
```

3.2 Caso di studio: funzioni C e JavaScript

3.2.1 Una funzione in C

Il programma `main` definito nella sezione sezione 1.3.1, pag. 38

è tecnicamente già una funzione. Semanticamente tuttavia essa rappresenta il programma e non un elemento riusabile; inoltre la riusabilità stessa è limitata dall'assenza di argomenti.

[1]

Una riscrittura più appropriata in termini di funzione si ottiene incapsulando il codice del `main` entro il corpo di una funzione dotata di una specifica signature. Ad esempio:

```
int contaParole( FILE* fp ){
 char c; //carattere corrente in lettura
 int nw  = 0;  // numero delle parole
 int inw = 0;  // variabile di stato
   while( (c=getc(fp)) != EOF ) {
   if(c==' ' || c=='\n' || c=='\t' )
     inw = 0;
   else if( inw == 0 ){
     inw = 1;
     ++nw;
   }
  }
  return nw;
}//contaParole
```

La funzione specifica come argomento di ingresso una variabile di tipo *file pointer*; l'uso della operazione `getc` al posto di `getchar` permette di generalizzare il funzionamento della nuova soluzione a una sorgente costituita da un file di testo generico.

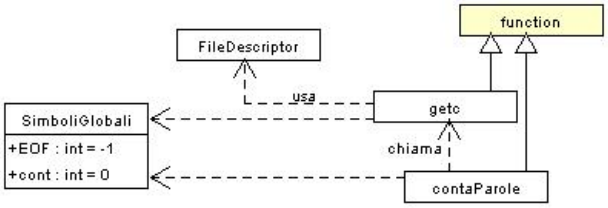
Pur nella sua semplicità, l'esempio mostra il ruolo strategico delle funzioni nel conseguimento di requisiti quali riusabilità, modularità e modificabilità e il ruolo dei simboli nella interazione tra funzioni. La costante `EOF` rappresenta un simbolo di dominio concettualmente condiviso tra `getc` e `contaParole`, e segnala l'esistenza di un *vocabolario di riferimento* comune alle due funzioni.

Il progetto del codice C si basa sull'incapsulamento nella funzione `getc` dei dettagli di accesso alla sorgente dei dati. Questa funzione in realtà opera come una procedura, in quanto ogni chiamata a `getc` deve produrre un effetto

[1] La coppia di argomenti `argc`, `argv` inseribili nella signature del `main` C rappresenta informazione di ingresso ricevuta dal sistema operativo e quindi costituisce una limitazione rispetto al concetto generale.

collaterale sul descrittore di file referenziato da `fp` tale da permettere una scansione progressiva degli elementi del file. Il descrittore di file opera in accordo al *pattern iterator*, che permette l'accesso sequenziale agli elementi di un oggetto composto (struttura di dati) senza esporre la sua rappresentazione interna.

Conteggio delle parole: modello del codice



3.2.2 Una funzione in JavaScript

Lo stile proposto dal C è stato adottato da molti linguaggi introdotti successivamente, come ad esempio Java e JavaScript: infatti, il corpo della funzione `contaParole` rimane identico anche in questi linguaggi. L'unica differenza è connessa alla intestazione della funzione, in quanto Java ha eliminato il concetto di puntatore e JavaScript non prevede una specifica statica di tipo per le variabili.

La versione JavaScript della funzione `contaParole` può assumere la forma che segue.

JavaScript: Funzione contaParole

```
contaParole = function( fp ){
  // fp: sorgente di caratteri
  var c;     // carattere corrente
  var nw =0; // numero delle parole
  var inw=0; // variabile di stato
  while( (c=getc(fp)) != EOF ) {
      if(c==' ' || c=='\n' || c=='\t' )
inw = 0; // non siamo entro una parola
  else if( inw == 0 ){
inw = 1; // siamo entro una parola
++nw;
  }
  }
```

```
  return nw;
}//contaParole
```

Il codice JavaScript invece permette più flessibilità sulla natura della sorgente: l'unico vincolo è che esista una funzione `getc` che, data la sorgente, ne estragga un carattere.

Riscrittura della funzione getc in JavaScript

Poichè il concetto di file è assente dallo spazio concettuale e di lavoro di JavaScript, per le sperimentazioni on-line conviene adottare come sorgente dati una stringa e introdurre una versione ad hoc di `getc`:

```
EOF = -1; //costante di fine sorgente
cont = 0; //variabile globale di conteggio

getc = function( s ){
//s :   stringa

  if( cont == s.length ) return EOF;
  else return s.charAt( cont++ );
}
```

I simboli globali `EOF` e `cont` permettono di riusare senza variazione alcuna la funzione `contaParole` scaturita dal libro sul linguaggio C. Ad esempio:

```
contaParole("questa frase contiene 5 parole");
//chiamata di funzione che restituisce il valore 5
```

L'uso dei simboli globali viola però il principio di località ed espone il programma a grossi rischi in relazione al mantenimento della sua coerenza interna nelle diverse fasi del suo ciclo di vita. Chiarito il ruolo di `EOF` come costante di dominio, particolarmente critico è il ruolo della variabile globale `cont`.

Ho tolto la frase L'uso di funzioni come enti di prima classe sempre per il solito motivo – l'espressione di prima classe non significa nulla per chi legge, se non la si spiega. Ho quindi riformulato la frase sotto.

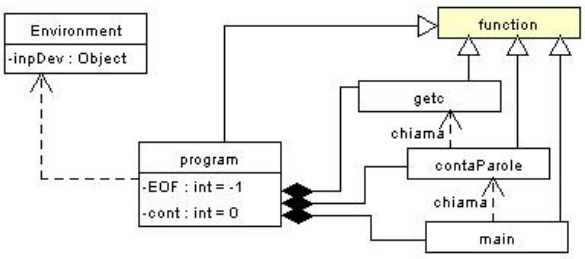
Come vedremo ora, l'adozione di un approccio funzionale permette invece di impostare soluzioni JavaScript *senza* la presenza di variabili globali, ossia senza variabili libere come `cont` all'interno del corpo della funzione `getc`. Spesso ci si riferisce a un tale approccio dicendo che le funzioni divengono **enti di prima classe**.

3.2.3 Eliminazione dei side effects tramite chiusure

Le soluzioni che discuteremo in questa sezione sono, sintassi a parte, le stesse che avrebbe pensato un progettista abituato a lavorare in *Scheme*. Nel procedere a queste nuove definizioni eseguiremo forme, anche se molto limitate, di *refactoring* di codice.

Come prima architettura riproduciamo l'architettura dell'intero programma C all'interno di una singola funzione JavaScritpt.

Incapsulamento delle informazioni in funzioni di prima classe



La possibilità di definire variabili locali aventi come valore enti di tipo funzione permette, unitamente alle regole di visibilità associate alle funzioni, di definire un componente software (riusabile) che elimina la necessità di variabili globali.

JavaScript: Incapsulamento delle informazioni in funzioni

```
/*
la funzione program rappresenta un programma che
conta le parole contenute in una sequenza di caratteri
*/
program = function(){
 var EOF  = -1; //costante di fine sorgente
 var cont = 0;  //contatore di parole

 //procedura di lettura da dispositivo di input
 var getc = function ( input ){
if( cont < input.length){
return input.charAt(cont++);
}
else return EOF;
```

```
 }

 //funzione che decide se un carattere  un separatore
 separatore = function ( c ){
   return (c==' ' || c=='\n' || c=='\t');
 }//separatore

 //Conteggio delle parole
 var contaParole = function( fp ){
 //fp :   sorgente di caratteri
 var EOF = -1; //costante di fine sorgente
 var c;    //carattere corrente
 var nw=0;  //numero delle parole
 var inw=0; //variabile di stato
     while( (c=getc(fp)) != EOF ) {
   if( separatore(c) ) inw = 0;
   else if( inw == 0 ){ inw = 1; ++nw; }
}
   return nw;
 }//contaParole

 /*entry point */
 var main = function (){
   return contaParole( inpDev.value );
 }

 return main();
}
```

La funzione `program` è strutturalmente identica al programma C, con l'incapsulamento della decisione sulla natura di separatore di un carattere all'interno della funzione `separatore`.

La funzione `program` sarà invocata o da un'altra funzione JavaScript, o all'interno di una espressione di ingresso nell'ambiente di lavoro; pertanto essa può essere scritta in puro stile funzionale, facendo coincidere il risutato della propria elaborazione con quello della funzione `main`.

Per verificare l'assenza di effetti collaterali possiamo valutare l'espressione

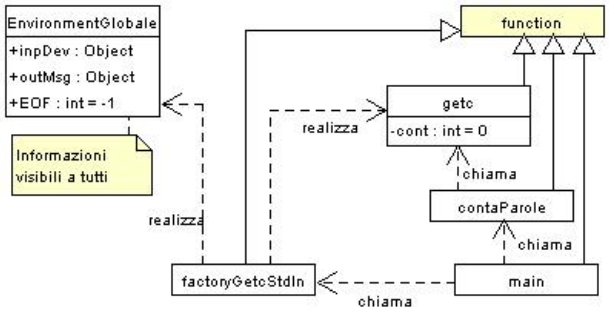
```
program() == program()
```

controllando che dia risultato `true` per qualunque frase scriviamo sul dispositivo di ingresso.

Incapsulamento dello stato in chiusure lessicali

Risolvere un problema costruendo una funzione che incapsula al suo interno tutte le altre impedisce che queste ultime siano riusabili in altri contesti. è dunque preferibile modificare il modello del sistema, cercando di costruire `getc` come una chiusura lessicale che funga da *iteratore* sulla frase, incapsulando in un proprio stato interno la posizione dell'ultimo carattere letto.

Incapsulamento dello stato in una chiusura lessicale



La costruzione della chiusura può essere realizzata da un'altra funzione che assume il ruolo di *fabbrica* (*factory*).

JavaScript: Factory che costruisce una chiusura lessicale di lettura caratteri

```
//Factory
factoryGetc = function(){
 var cont = 0; //contatore di parole
 EOF = -1; //costante di fine sorgente (globale)

 //restituisce una procedura di lettura da dispositivo di input
 return  function (input){
 if( cont < input.length){
return input.charAt(cont++);
 }
 else {
  cont=0;//reinizializza cont
  return EOF;
 }
 }
}//factoryGetc
```

La funzione `contaParole` non cambia rispetto alla versione precedente; al `main` viene invece affidato ora il compito di *configurare* e *attivare* il sistema:

JavaScript: Configurazione iniziale ed esecuzione

```
/*
* Operazione di prova
*/
test = function(){
 var frase = "Hello world";
 return "la frase in input ha " + contaParole(inpDev.value) + " parole" +
  "\n" +
 "la frase " + frase + " ha  " + contaParole(frase) + " parole";
}

main = function(){
 getc=factoryGetc(); // configurazione del sistema
 return test();      // elaborazione
}
```

3.3 Costruire e collaudare funzioni

3.3.1 Fasi di lavoro

Il flusso di lavoro (*workflow*) connesso al progetto e alla costruzione di una funzione si può articolare nelle seguenti fasi:

1. *analisi dei requisiti e definizione della signature (interfaccia) della funzione*:

   pensando alla funzione come un servitore, si stabiliscono le informazioni che questo deve ricevere in ingresso e fornire in uscita;

2. *definizione delle pre- e post-condizioni*:

   essendo noto che la signature stabilisce un contratto parziale tra cliente e servitore, si definiscono le condizioni che il cliente deve rispettare prima di invocare la funzione (*pre-condizioni*) e le condizioni che devono essere verificate al termine della esecuzione della funzione (*post-condizioni*);

3. *pianificazione delle azioni di collaudo relative all'uso della funzione*:

   si stabiliscono le risposte attese in relazione a un insieme di dati di ingresso opportunamente selezionato;

4. *progetto del comportamento della funzione*:

   si imposta la soluzione logica del problema che il corpo della funzione dovrà realizzare, ponendo in luce i criteri adottati e le vie che si è ritenuto opportuno escludere. In questa fase si possono adottare metodologie si scomposizione (*top-down*) oppure di composizione (*bottom-up*) e seguire forme di ragionamento iterativo o ricorsivo.

5. *realizzazione (codifica) della soluzione*:

   la soluzione logica del problema viene espressa in frasi del linguaggio scelto per la implementazione del sistema;

6. *completamento delle azioni di collaudo ed esecuzione del collaudo*:

   si pianificano ulteriori test capaci di sollecitare, insieme ai precedenti, tutte le strade (o almeno le più significative) che la funzione può internamente percorrere e si eseguono tutti i collaudi pianificati, che devono terminare con successo.

3.3.2 Pianificazione del collaudo di funzioni

Nel workflow descritto nella sezione precedente la pianificazione delle azioni di collaudo comincia ancor prima che sia stato scritto il corpo della funzione. Così facendo si ottengono due importanti obiettivi:

- si rafforza la possibilità di validare il prodotto rispetto ai requisiti.

  Pianificare il collaudo ponendosi dal punto di vista dell'utilizzatore significa astrarre dal comportamento interno della funzione e focalizzare l'attenzione solo sul comportamento atteso in relazione a ingressi previsti e ingressi imprevisti. Stabilire in modo chiaro cosa ci si aspetta da una funzione a fronte di un ingresso non previsto è essenziale per la costruzione di *software sicuro* e fornisce indicazioni spesso indispensabili per una corretta impostazione del progetto.

- si promuove una pratica di sviluppo in cui il collaudo viene svolto in modo continuo, componente per componente, senza attendere la costruzione di tutto il sistema

Queste pratiche fanno parte del repertorio promosso dai sostenitori delle metodologie di sviluppo note come *Extreme Programming* e trovano concreto supporto attraverso strumenti appositamente progettati e costruiti a questo scopo, come ad esempio JUnit (vedi **Material/Tools/junit3.8/README.html**) di Beck e Gamma, che è già integrato in molti degli IDE più diffusi.

Sebbene il collaudo permetta di rilevare possibili errori, non dà mai ovviamente la totale garanzia che il codice finale ne sia privo. Per raggiungere questo scopo sarebbe necessario *dimostrare* che il codice è corretto. Nella sezione *Progetto basato su invarianti* vedremo come sia possibile perseguire, in taluni casi, questo obiettivo.

Un esempio di pianificazione del collaudo

Con riferimento al problema *VocaliMaiuscole*, la precedente specifica:
la frase `upCase('abcdè, 0)` deve denotare la stringa `'Abcdè`
la frase `upCase('abcdè, 3)` deve denotare la stringa `'dè`
può essere meglio incapsulata in una funzione di test priva di argomenti e di effetti collaterali (tra cui l'uso di dispositivi di I/O), utilizzando funzioni di asserzione nello stile JUnit (vedi **Material/Tools/junit3.8/javadoc/junit/framework/Assert.htm**

```
public void testUpCaseTwoArgs(){
  assertTrue("abcde", myLibrary.upCase("abcde",0).equals("AbcdE") );
  assertTrue("abcde", myLibrary.upCase("abcde",1).equals("bcdE") );
  assertTrue("abcde", myLibrary.upCase("abcde",3).equals("dE") );
  assertTrue("abcde", myLibrary.upCase("abcde",4).equals("E") );
}
```

Le operazioni `assert`, tra cui spiccano `junit.framework.Assert.assertTrue` e `junit.framework.Assert.assertEquals`, sono procedure Java che vengono attivate usando il meccanismo della *reflection*.

Queste procedure emettono un messaggio (il primo argomento loro trasferito) solo in caso di fallimento.

Non appena una `assert` fallisce, tutta la procedura di test fallisce: se il codice di un'operazione di collaudo contiene altre `assert`, queste non vengono eseguite.

Dualmente, l'operazione `junit.framework.Assert.fail` provoca il fallimento di un test con emissione del messaggio ricevuto come (unico) argomento.

Riproduzione del collaudo JUnit in JavaScript

In JavaScript un comportamento simile a quello di JUnit può essere costruito come segue:

- si introduce una funzione `exec` che, ricevuta come parametri di ingresso una sequenza di chiusure lessicale a zero argomenti, esegue ciascuna

chiusura in sequenza, interrompendosi alla prima che restituisce un messaggio diverso da 'ok' ;

- si definisce la funzione `assertTrue(msg,cond)` (e volendo le altre assert) come una funzione che restituisce una chiusura a zero argomenti;

- si definisce la funzione `fail(msg)` come una funzione che restituisce il messaggio `fail + msg`.

Attenzione: nel codice sotto, il parametro della funzione exec era b. MI sembra improbabile, dato che b è una variabile definita localmente: inoltre, arguments sarebbe indefinita. Mi sembra molto più probabile che il parametro della exec debba essere arguments, quindi CORREGGO il codice in conseguenza. Domanda: ma perchè la variabile b è definita fuori dal ciclo for, se poi tanto serve solo là dentro? –NAT: non avevo aggiornato il codice che ora è il seguente:

```
exec = function( cmd ){
var n = arguments.length;
var cmd;
for( i=0; i < n ; i++ ) {
          try{
 cmd = arguments[i]();
 if( cmd != 'ok' ){
   msgOut.println( cmd );
   break;
}
  }catch( e ){ break; } //restituisce l'eccezione
}
return "done";
}

assertTrue = function( msg, cond ){
return function(){
    if( cond ) return "ok";
    else return msg + " failed";
  }
}

fail = function( msg ){
  return "fail " + msg;
}//fail
```

Una operazione di collaudo assume quindi la forma che segue:

```
testUpCaseTwoArgs = function(){
 return exec(
   assertTrue('abcde 0', upCase('abcd, 0) == 'Abcd),
   assertTrue('abcde 3', upCase('abcd, 3) == 'd)
 );
} //Restituisce 'don

testUpCaseTwoArgs1 = function(){
 return exec(
   assertTrue('abcde 0', upCase('abcd, 0) == 'Abcd),
   assertTrue('abcde 2', upCase('abcd, 2) == 'd),
   assertTrue('abcde 3', upCase('abcd, 3) == 'd)
 );
} //Restituisce 'don
```

3.3.3 Collaudo di funzioni con eccezioni

Nel caso un'operazione specifichi di lanciare una o più eccezioni, il collaudo deve essere impostato in modo che, in casi normali, il sorgere di un'eccezione si traduca nella segnalazione di un fallimento e nei casi eccezionali sia il mancato insorgere dell'eccezione a tradursi nella segnalazione di un fallimento.

Ad esempio, il collaudo di `upCaseConstraint` che lancia l'eccezione `wrong call` nel caso i parametri non siano congruenti può iniziare dal comportamento atteso in caso di chiamate corrette:

```
public void testUpCaseConstraintOk(){
 try{
   assertTrue("0 vuota", myLibrary.upCaseConstraint("abcde",0,"").equals("AbcdE"));
   assertTrue("3 Abc",   myLibrary.upCaseConstraint("abcde",3,"Abc").equals("AbcdE"));
 }
 catch( Exception e ){
   fail("upCaseConstraint non deve dare eccezioni");
 }
}//testupCaseConstraintOk
```

In JavaScript:

```
testUpCaseConstraintOk = function(){
 try{
  return exec(
   assertTrue('0 vuot, upCaseConstraint("abcde",0,"")=='Abcd ),
   assertTrue('3 Abc',   upCaseConstraint("abcde",3,"Abc")=='Abcd)
  )
 }
```

```
 catch( e ){
  return fail("upCaseConstraint non deve dare eccezioni");
 }
}//testUpCaseThreeArgsOk
```

Per collaudare il corretto manifestarsi delle eccezioni, si può impostare una chiamata non corretta e gestire come caso di collaudo positivo il manifestarsi della eccezione `wrong call` e come caso di fallimento la non generazione della eccezione.

```
public void testpCaseConstraintKo(){
 try{
   myLibrary.upCaseConstraint("abcde",3,"Abcd");
 }
 catch( Exception e ){
  assertTrue("cont<sOut.length",e.getMessage().equals("wrong call"));
  return;
 }
 fail("upCaseConstraint: mancata eccezione wrong call");
}//testpCaseConstraintKo
```

In JavaScript:

```
testpCaseConstraintKo = function(){
 try{
   upCaseConstraint("abcde",3,"Abcd");
 }
 catch( e ){
   return exec( assertTrue('cont<sOut.length',e=='wrong call'));
 }
 return fail("upCaseConstraint: mancata eccezione wrong call");
}//testUpCaseThreeArgsKo
```

Se ad esempio riscriviamo `upCaseConstraint` eliminando la generazione della eccezione

```
upCaseConstraint = function(  s,  cont,  sOut ){
  //s: la stringa di ingresso
  //cont: numero dei caratteri gi esaminati
  //sOut: stringa di uscita per sottostringa s da 1 a cont
  if (s.length == cont) return sOut;
  if( vocale( s.charAt(cont) ) )
   return upCaseConstraint(s,cont+1,sOut+s.charAt(cont).toUpperCase());
  else
   return upCaseConstraint(s,cont+1,sOut+s.charAt(cont));
}
```

e rieseguiamo il test `testUpCaseThreeArgsKo()`, otteniamo il risultato `fail upCaseConstraint: mancata eccezione wrong call`.

3.4 Caso di studio: workflow di progetto e costruzione

3.4.1 Definizione della signature

Seguendo il workflow delineato nella sezione 3.3.1, pag. 82 , impostiamo la specifica e la costruzione delle operazioni relative al problema del conteggio delle parole.

Definizione della signature della funzione risolvente

L'ingresso è costituito da una generica sorgente di dati, che renda possibile l'acquisizione dei singoli caratteri in sequenza. L'uscita è un numero intero che rappresenta, se positivo o nullo, il numero delle parole riconosciute sulla sorgente oppure, se `-1`, l'indicazione di assenza di parole.

```
int contaParole( inputSource inp)
```

In relazione al tipo `inputSource` si assume la disponibilità di una procedura `getc` che ad ogni invocazione restituisca un carattere successivo della sorgente o il valore `-1` se la sorgente è vuota o ha esaurito i caratteri.

Definizione delle pre- e post- condizioni

Precondizione: la sorgente `inputSource` esiste.

Postcondizione: la sorgente è nello stato logico completamente esplorata.

3.4.2 Pianificazione del collaudo di uso

Mantenendo l'idea che una *parola* sia una qualsiasi sequenza di caratteri, in cui ciascun carattere *non è* un separatore, il collaudo dipende dall'insieme dei separatori selezionato. Per i casi che seguono l'insieme dei separatori è quello stabilito dalla funzione `separatore`:

JavaScript: Decisore sui separatori

```
separatore = function ( c ){
   return (c==' ' || c=='\n' || c=='\t');
}//separatore
```

Su queste basi si possono impostare le operazioni di collaudo:

JavaScript: Pianificazione del collaudo

```
testFraseNonVuota = function(){
 return exec(
  assertTrue("a b 12; -+", contaParole("a b 12; -+")==4),
  assertTrue("1 ;+ +2x4",  contaParole("1 ;+ +2x4")==3),
  assertTrue(" ;  \n ",    contaParole(" ;  \n ")==1)
 );
}

testFraseVuota = function(){
 return exec(
  assertTrue("",       contaParole(" ")==0),
  assertTrue("   ",    contaParole("     ")==0),
  assertTrue("  \n ", contaParole("  ")==0)
 );
}
```

3.4.3 Progetto del comportamento

La soluzione si può ottenere impostando un procedimento iterativo basato sul seguente invariante:

Se `P` è la posizione del carattere corrente `ch` nella sorgente dati, allora `nw` è il numero delle parole presenti nel tratto di sorgente che va dall'inizio fino a `P`.

Lo scopo del processo iterativo è mantenere vera la relazione tra `P` e `nw` variando in modo progressivo `P` dall'inizio alla fine della sorgente con l'operazione `getc`. In questo modo, al termine della scansione, `nw` conterrà il numero delle parole presenti in tutta la sorgente.

Il problema che si pone è come decidere quando incrementare `nw`. Per il momento cerchiamo di esprimere il ragionamento che ha prodotto il codice che già conosciamo. In altre sezioni vedremo ragionamenti completamente diversi.

Sia `inw` una variabile di stato con il seguente significato:

```
 inw=1 : la scansione sta avvenendo entro una parola
inw=0 : la scansione sta avvenendo al di fuori di una parola
```

Per mantenere vero l'invariante si può ragionare in questo modo:

```
Inizialmente sia nw=0 e inw=0;
Sia ch una variabile che denota un carattere;
ch = getc(inp); //si acquisisce un nuovo carattere dalla sorgente
/*
Invariante:
 nw  il numero di parole dalla posizione di inizio della sorgente inp
 alla posizione del carattere ch
*/
if ( !separatore(ch) ){
  if (inw=0) {//non siamo ancora entro una parola
   nw  = nw + 1; // abbiamo trovato l'inizio di una parola
   inw = 1;      // commutiamo di stato
  }
  //se inw=1 siamo gi entro una parola e nulla cambia
}
else
/*il carattere corrente non  un separatore. Dunque:
  nw non deve cambiare;
  lo stato, se era 1, deve essere posto a 0;
*/
if( inw = 1 ) inw = 0;
//Se ci sono ancora caratteri da esplorare si itera il procedimento
```

3.4.4 Costruzione di una versione prototipale

Per pervenire in modo rapido e sintetico ad una versione prototipale funzionante della soluzione, si può costruire una funzione tail recursive caratterizzata dalla seguente signature:

```
 int contaParole( inputSource inp, int nw, int inw)
/*
inp: sorgente con i dati di ingresso
nw:  numero delle parole trovate
inw: variabile di stato
    inw=1 : la scansione sta avvenendo entro una parola
    inw=0 : la scansione sta avvenendo al di fuori di una parola
*/
```

Per definire il corpo della funzione iniziamo dal caso base:

```
se  la sorgente  terminata restituisci nw
```

Nel caso generale di sorgente non terminata, diciamo

```
 ch = getc(inp); // si acquisisce un nuovo carattere dalla sorgente
 if ( !separatore(ch)){
   if (inw=0) //non siamo ancora entro una parola
    il risultato coincide con il risultato di contaParole( inp,  nw+1,  1);
   else //inw=1 : siamo gi entro una parola e nulla cambia
    il risultato coincide con il risultato di contaParole( inp,  nw,  inw);
 }
 else //ch non  un separatore
  il risultato coincide con il risultato di contaParole( inp,  nw,  0);
```

Per non modificare l'interfaccia concordata con i clienti, che prevedeva un solo argomento (e non tre), il codice finale sarà una coppia di funzioni.

JavaScript: Codice di un prototipo basato su ricorsione tail

```
contaParole = function( inp ){
 return contaParoleRicIt( inp, 0, 0 );
}

contaParoleRicIt = function( inp,nw,inw ){
/*
inp: sorgente con i dati di ingresso
nw: numero delle parole trovate
inw: variabile di sato
    inw=1 : la scansione sta avvenendo entro una parola
    inw=0 : la scansione sta avvenendo al di fuori di una parola
*/
 var ch = getc(inp); //carattere corrente
 if( ch = EOF ) return nw;
 if ( !separatore(ch)){
   if (inw=0) //non siamo ancora entro una parola
     return contaParoleRicIt( inp,  nw+1,  1);
   else //inw=1 : siamo gi entro una parola e nulla cambia
     return contaParoleRicIt( inp,  nw,  inw);
 }
 else //ch non  un separatore
   return contaParoleRicIt( inp,  nw,  0);
}
```

Nel caso il linguaggio di codifica sia dotato dell'ottimizzazione relativa alla tail recursion, il prototipo costituisce anche la versione definitiva.

Se così non è, volendo fornire una soluzione più efficiente si può mettere a punto il tipico codice basato su cicli:

JavaScript: Codice iterativo risultante

```
contaParoleIt = function (inp){
//fp: sorgente (esiste per ipotesi)
 var inw = 0;//variabile di stato
 var nw= 0;  //numero corrente delle parole
 var ch;     //carattere corrente
 while( (ch=getc(fp) ) != EOF ){
   if ( ! separatore(c)){
    if (inw==0) { //non siamo ancora entro una parola
   nw  = nw + 1;
   inw = 1;
    }
else inw=0; //it test  meno efficiente
 }//while
 return nw;
}//contaParoleIt
```

3.4.5 Completamento del collaudo

Questa parte manca

Capitolo 4

Dai costrutti alle metodologie

4.1 Metodologie bottom-up e top-down

4.1.1 Metodologie per la risoluzione di problemi

Per risolvere un problema vi sono due modi principali di procedere. Il primo parte dall'analisi del problema e cerca di scomporre il problema stesso in sottoproblemi più semplici, individuando durante questo processo operazioni, componenti, infrastrutture utili alla soluzione.

Il secondo modo prende atto delle risorse (operazioni, componenti, infrastrutture) disponibili e comincia ad utilizzarle per cercare di pervenire ad una configurazione che risolva il problema.

Di queste due metodologie, dunque, una procede per *analisi* (decomposizione del problema) l'altra per *sintesi* (aggregazione di componenti).

Le due metodologie devono essere viste come complementari e non mutuamente esclusive.

Iterazione e ricorsione costituiscono tecniche di soluzione correlate. Queste tecniche sono ancora al cuore della costruzione di ogni sistema software e hanno profonda influenza sul piano del progetto, della struttura del sistema e dei processi computazionali che da esso derivano.

4.1.2 Metodologia bottom-up

La metodologia di progettazione e sviluppo *bottom-up* enfatizza un processo di costruzione per sintesi, basato sulla interconnessione di componenti già disponibili e funzionanti. Da sempre usata nella progettazione e sviluppo di sistemi hardware e nell'ingegneria tradizionale (edile, meccanica), questa metodologia promuove la costruzione di componenti standard efficienti, affidabili ed il più possibile generici e riutilizzabili.

4.1.3 Metodologia top-down

La metodologie di progettazione e sviluppo top-down enfatizza metodi di risoluzione dei problemi per analisi. Essa parte da una descrizione di alto livello e procede riducendo il problema iniziale in un insieme di sotto-problemi più semplici. Ciascun sotto-problema viene poi decomposto, ancora in modo top-down, fino a giungere ad uno stadio facilmente risolubile con le operazioni e meccanismi elementari disponibili. Nel caso della progettazione del software, il punto di arrivo è costituito dalle mosse elementari della macchina virtuale prescelta.

4.1.4 Il ragionamento iterativo

Concettualmente, l'iterazione espressa da costrutti *while* o *for* realizza un **processo risolutivo di tipo accumulativo** caratterizzato da precise proprietà:

- il processo risolutivo si articola in un numero finito di passi computazionali;
- il numero di passi può essere stabilito a priori o dipendere dal soddisfacimento di una relazione logica, che può comprendere una o più *variabili di controllo*;
- esiste sempre una variabile detta *accumulatore* che denota, ad ogni passo computazionale, un valore progressivamente più vicino al risultato, secondo il seguente schema: *sapendo che al passo generico k l'accumulatore esprime il valore corrente della soluzione fino al passo k, la soluzione al passo k+1 si ottiene modificando l'accumulatore come segue: ...*
- esiste una relazione tra variabili di controllo, accumulatore e dati del problema, che viene detta *invariante* (di iterazione o di ciclo) in quanto deve essere sempre verificata ad ogni passo computazionale.

Quasi sempre, l'invariante di ciclo non è espresso in modo esplicito, anche perchè non è sempre facile farlo, soprattutto in modo formale.

Con riferimento alla versione iterativa della funzione `upCase` che risolve il problema *VocaliMaiuscole*:

```
upCase = function( s ){
//s denota la stringa da convertire
 var s1; //denota la stringa risultato
```

```
 for( i=0, s1=""; i < s.length; i++ ){
   if( vocale( s.charAt(i) ) ) s1 = s1+s.charAt(i).toUpperCase();
   else s1 = s1+s.charAt(i);
 }
 return s1;
}
```

il progettista può avere ragionato come segue:

- il numero di passi computazionali è uguale al numero dei caratteri che compongono la stringa di ingresso;
- la variabile `i` che governa il ciclo `for` costituisce la *variabile di controllo* del procedimento;
- la variabile `s1` costituisce l'*accumulatore*
- l' *invariante di ciclo* è espresso dalla condizione che al passo *i-mo* `s1` denoti la stringa di uscita corrispondente alla sottostringa di ingresso dal carattere di indice `0` al carattere di indice `i`.

Lo scopo del ciclo è fare in modo che `i` divenga uguale a `s.length`: in questo stato, infatti, se l'invariante viene preservato, la stringa `s1` denota necessariamente il risultato voluto.

4.1.5 Il ragionamento ricorsivo

La ricorsione consiste nella possibilità di definire un ente in termini di sè stesso.

Questa tecnica costituisce la forma più radicale di riuso, in quanto il progetto di un'operazione o di una struttura può essere impostato includendo l'operazione o la struttura stessa tra le mosse primitive. Questa apparente assurdità è superata dal fatto che il riuso dell'operazione avviene con dati diversi da quelli iniziali e il riuso di una struttura termina in casi particolari.

Il ragionamento ricorsivo è riconducibile al *principio di induzione matematica*:

- se una proprietà P vale per un intero $n = n_0$,
- e si può provare che, assumendola valida per n , essa vale per $n + 1$,
- allora la proprietà P vale per ogni $n >= n_0$.

Seguendo questo principio, il ragionamento ricorsivo si sviluppa come segue:

1. si identificano uno o più *casi base* la cui soluzione è nota o già disponibile;

2. si disgrega il problema relativo al caso generico in sotto problemi dello stesso tipo, ma con dati di ingresso tali da ricondurre progressivamente la soluzione ai casi base.

Con riferimento alla versione ricorsiva della funzione `upCase`:

```
upCase = function( s,  cont ){
//s denota la stringa di ingresso
//cont denota il numero dei caratteri gi esaminati
  if (s.length == cont) return "";
  if( vocale( s.charAt(cont) ) )
     return s.charAt(cont).toUpperCase() + upCase(s,++cont);
  else
     return s.charAt(cont) + upCase(s,++cont);
}
```

il progettista può avere ragionato come segue:

1. il *caso base* è il caso in cui la stringa di ingresso è vuota;

2. nel caso di stringa non vuota, il problema viene disgregato in due parti: l'analisi del primo carattere corrente (di posizione `cont`) e l'analisi del resto della stringa (caratteri da `cont+1` a `s.length`). La seconda parte del problema è ovviamente risolta dalla funzione `upCase` stessa, in quanto il dato di ingresso si riduce di lunghezza e ciò riconduce progressivamente il problema al caso base.

4.1.6 La ricorsione di coda

La versione sintatticamente ricorsiva ma computazionalmente iterativa della soluzione al problema:

```
upCase = function(  s,  cont,  sOut ){

//s: la stringa di ingresso
//cont: numero dei caratteri gi esaminati
//sOut: stringa di uscita per sottostringa s da 1 a cont
 if (s.length == cont) return sOut;
```

```
 if( vocale( s.charAt(cont) ) )
  return upCase(s, cont+1, sOut+s.charAt(cont).toUpperCase());
 else
  return upCase(s, cont+1, sOut+s.charAt(cont));
}
```

viene costruita impostando la variabile di controllo (`cont`) e l'accumulatore (`sOut`) come argomenti della funzione risolvente.

Ogni chiamata (ricorsiva) non è ora seguita da alcuna ulteriore operazione. Quindi, ricevuto di nuovo il controllo da una invocazione a sè stessa, l'attivazione corrente non deve più accedere ai suoi dati locali, in quanto restituisce subito a sua volta il controllo (e il risultato) al chiamante. Si parla in questo caso di chiamate o di ricorsione di coda (*tail recursion*).

Un compilatore intelligente può accorgersi se tutte le chiamate nel corpo di una procedura o funzione sono di tipo tail e produrre in tal caso codice macchina che riusi un unico record di attivazione: se questo accade, il comportamento a tempo di esecuzione – e quindi il costo computazionale – è identico a quello della versione iterativa. è questo il caso di molti compilatori *Lisp* e del compilatore *Prolog*.

Un nuovo ragionamento

L'aspetto più interessante è nella differenza di ragionamento che porta al codice. Infatti, ora lo scopo del progettista è mantenere soddisfatta una *relazione* tra gli argomenti della funzione, scelti in modo da rappresentare in modo completo l'informazione di ingresso, l'informazione di uscita e l'informazione di stato e/o controllo.

Nel caso di `upCase`, si mira a mantenere una precisa relazione tra l'informazione di ingresso `s`, l'informazione di uscita `sOut` e la variabile di stato/controllo `cont`, facendo in modo che `sOut` rappresenti in ogni istante la trasformazione voluta di `s` per la parte che va dal carattere di indice `0` al carattere di indice `cont-1`. La funzione deve continuare a mantenere vera questa relazione facendo in modo che `cont` raggiunga il valore `s.length`: quando ciò accade, `sOut` rappresenta la risposta voluta.

Come vedremo, questo è il tipico ragionamento indotto dal Prolog, un linguaggio privo di costrutti *while* e in cui le funzioni sono relazioni di verità tra oggetti del dominio rappresentati da termini.

```
upCase = function(  s,  cont,  sOut ){
//s: la stringa di ingresso
//cont: numero dei caratteri gi esaminati
//sOut: stringa di uscita per sottostringa s da 1 a cont
 if (s.length == cont) la risposta deve essere sOut, per ipotesi;
 altrimenti, detto ch il carattere di posizione cont:
```

```
   if vocale(ch)
     la relazione viene mantenuta incrementando cont di 1 e
     concatenando alla variabile che denota l'uscita la maiuscola
     che corrisponde a ch.
   if non vocale(ch)
     la relazione viene mantenuta incrementando cont di 1 e
     lasciando alla variabile che denota l'uscita il carattere ch.
```

La frase *la relazione viene mantenuta* può operazionalmente diventare una invocazione alla funzione `upCase` stessa, che ricomincerà a lavorare partendo da uno stato più vicino al risultato finale.

I casi estremi del collaudo sono quelli in cui l'informazione di uscita `sOut` è tutta ignota o già tutta nota:

```
testUpCaseThreeArgsEstremi = function(){
 return exec(
   assertTrue('abcde 0', upCase('abcde',0,"") == 'Abcde'),
   assertTrue('abcde all', upCase('abcde','abcde'.length,'Abcde') == 'Abcde')
 );
}
```

Le altre situazioni di collaudo d'uso sono casi intermedi, in cui è evidentemente responsabilità del chiamante garantire la consistenza iniziale della relazione:

```
testUpCaseThreeArgs = function(){
 return exec(
   assertTrue('abcde 1',   upCase('abcde',1,'a') == ('Abcde')),
   assertTrue('abcde 3', upCase('abcde',3,'Abc') == ('Abcde'))
 );
}
```

Nel caso della operazione di moltiplicazione discussa in precedenza, l'impostazione tail recursive si distingue per la sinteticità e l'immediatezza con cui riflette il ragionamento di progetto:

```
/**
 * Operazione di prodotto con ricorsione tail
 *@param x = moltiplicando (x>=0)
 *-+-param y = moltiplicatore (y>=0)
 *-+-param v = accumulatore (v>=0)
 *-+-return x*y+v
 */
multiply = function( x, y, v ){
```

```
 if( x == 0 ) return 0;
 if( y == 0 ) return v;
 else  //(y % 2)==0 significa che y e' pari
   return ((y%2)==0)?multiply(x<<1,y>>1,v):multiply(x<<1,y>>1,v+x);
}
```

Poichè è sempre possibile trasformare una versione ricorsiva in iterativa e viceversa, il progettista dispone di un grado di libertà.

Di solito (benchè a prima vista sembri il contrario) conviene utilizzare la ricorsione per impostare in modo rapido e sintetico una soluzione di primo livello del problema: il passaggio ad un versione iterativa espressa in termini di costrutti *while* o *for* può essere visto come un passo di ottimizzazione successiva, che ha come momento intermedio la produzione di una versione *sintatticamente ricorsiva* ma *computazionalmente iterativa*.

4.1.7 Progetto basato su invarianti

Nella fase di costruzione di un algoritmo o di una funzione, l'invariante di ciclo dovrebbe costituire il punto di partenza per il progetto di ogni processo iterativo: il codice che realizza il ciclo può essere ricavato come conseguenza logica del mantenimento dell'invariante a fronte delle trasformazioni volte a far convergere il valore dell'accumulatore verso il risultato.

Una volta reso esplicito l'invariante di ciclo, la consistenza della condizione di invarianza può essere verificata anche a tempo di esecuzione, con l'emissione di una eccezione nel caso di insuccesso.

In alcuni domini, come quello numerico, è possibile correlare in modo formale il codice del ciclo alla relazione di invarianza, mostrando il modo in cui ogni passo modifica la relazione, fino alla riproduzione della relazione stessa. Se questo fosse sempre possibile, non vi sarebbe più bisogno di collaudo inteso come testing in quanto ogni programma sarebbe corretto *per dimostrazione*.

Esempio: la moltiplicazione di due numeri naturali

Il progetto di un algoritmo che calcoli il risultato della moltiplicazione di due interi naturali `X` e `Y` può avere come punto di partenza la relazione:

```
X*Y =  x*((y div B)*B + y mod B)
```

ove `x` e `y` denotano due copie rispettivamente di `X` e `Y`, `B` denota un numero intero qualsiasi maggiore di `1`, `div` l'operatore di divisione tra due interi e `mod` l'operatore che dà il resto della divisione tra due interi. Questa relazione non cambia se la si riscrive come segue, con `v=0`:

```
X*Y = x*(y div B)*B + x*(y mod B) + v
```

Il progetto ruota intorno all'idea di modificare i valori di `x` e `y` in modo da mantenere *invariante* il risultato dell'espressione a destra dell'operatore `=`, fino a pervenire ad una situazione in cui il solo valore di `v` possa denotare il risultato cercato.

A questo fine si possono effettuare le seguenti modifiche: (metterei x', y', v' per far capire meglio. –NAT si usa proprio l'assegnamento)

```
y = y div B;
x = x * B;
v = v + x*(y mod B)
```

Poichè `y` diminuisce, *iterando* questo procedimento di trasformazione di `x`, `y` e `v` si giunge prima o poi al punto in cui `y=0`: in tale situazione, `v` denoterà il risultato cercato.

```
X*Y = x*(0 div B)*B + x*(0 mod B) + v = v
```

Una prima versione dell'algoritmo in Java può essere espressa dalla funzione iterativa che segue, in cui `v` è l'*accumulatore* del processo iterativo di soluzione:

```
/**
* Calcola il prodotto tra due interi
*-+-param x = moltiplicando (x>=0)
*-+-param y = moltiplicatore (y>=0)
*-+-return x*y
*/
int multiply( int x, int y ){
//{restituisce il valore x*y, con x,y >= 0 }
 int B = 10; //base
 int v = 0;
 if( x == 0 ) return 0;
 while( y > 0 ){
    v = v + x * (y%B); //% operatore resto divisione tra interi
    x = x*B;
    y = y/B; //divisione tra interi
 }
 return v;
}
```

Notiamo che, nel caso particolare in cui `B=2`,
$ymod(xxx)B$ vale o `0` (se `y` è pari) o `1` (se `y` è dispari):

perciò, `y/B` si può ottenere *senza bisogno di svolgere realmente la divisione,* semplicemente traslando (*shift*) di una posizione a destra la rappresentazione binaria di `y`. Analogamente, `x*B` si può ottenere traslando di una posizione a sinistra la rappresentazione binaria di `x`.

Sfruttando gli operatori di *shift*, si può dunque giungere al codice che segue:

```
int multiply( int x, int y ){
//restituisce il valore x*y, con x,y >= 0
//Invariante: x*y==x*(y div B)*B + x*(y mod B) + v
//opera assumendo come base B=2
 int v = 0;
 if( x == 0 ) return 0;
 if( y == 1 ) return x+v;
 while( y > 0 ){
   if( y % 2 != 0 ) /* y dispari */ v = v+x;
   x = x << 1; //moltiplica x per 2
   y = y >> 1; //divide y per 2
 }
 return v;
}

multiply = function(x, y){
//restituisce il valore x*y, con x,y >= 0
//Invariante: x*y==x*(y div B)*B + x*(y mod B) + v
//opera assumendo come base B=2
 var v = 0;
 if( x == 0 ) return 0;
 if( y == 1 ) return x+v;
 while( y > 0 ){
   if( y % 2 != 0 ) //y dispari
    v = v+x;
   x = x << 1; //moltiplica x per 2
   y = y >> 1; //divide y per 2
 }
 return v;
}
```