# Riassunti del corso di Linguaggi e Modelli Computazionali

Federico Marchetti
Lorenzo Tullini
Leonardo Lelli

14 settembre 2020

# Indice

# Note

Questo documento **non** è un riassunto ufficiale, ma raccoglie il contenuto delle slide e degli appunti presi a lezione. Inoltre non ha la pretesa di essere una sostituzione alle lezioni, in quanto non è possibile raccogliere in tempo utile la miriade di esempi e precisazioni fatte a lezione, pertanto sono stati riportati in questo documento solamente i principali. Tutti gli altri esempi possono essere letti nelle slide
Infine non è detto che tutto ciò che è racchiuso all'interno di questi riassunti sia completamente corretto.

Buono studio e che la forza sia con te!

# Capitolo 1

# Introduzione

Un linguaggio di alto livello è uno strumento per far eseguire ad un elaboratore le funzioni desiderate lavorando ad un livello concettuale più elevato di quello messo a disposizione (o indotto) dall'hardware con lo scopo di permettere una impostazione del problema più chiara. Tuttavia l'introduzione di un linguaggio porta con sé la necessità di opportuni traduttori che si occupino di colmare la distanza tra il linguaggio e quanto può essere direttamente compreso dalla macchina.
Oggigiorno i sistemi sono però caratterizzati da una sempre maggior interattività da cui consegue un legame più forte tra il *linguaggio* ed i livelli sottostanti. Ogni livello costituisce una macchina virtuale a sé stante in grado di interpretare simboli. Per il corretto funzionamento di tale macchina tuttavia è necessario che essa sia dotata di opportuni sistemi per l'interpretazione dei simboli, i quali richiedono una certa **sintassi**, che fornisce gli strumenti per la verifica della correttezza strutturale, ed una certa **semantica**, che fornisce il significato dei singoli simboli.

La capacità espressiva di un linguaggio ne fa un potente suggeritore di concetti e metodi di soluzione. Esiste infatti una molteplicità di linguaggi, in quanto ognuno di essi porta a sviluppare secondo determinati paradigmi e di conseguenza la risoluzione di un dato problema può risultare semplificata o meno a seconda del linguaggio scelto. Quindi non esiste un linguaggio "migliore" in assoluto, ma esiste un linguaggio migliore per l'ambito di applicazione.

## 1.1 Risoluzione dei problemi

Un modello relativamente semplice è costituito dal **modello a cascata**. Esso prevede una prima fase di enunciazione, analisi e progettazione seguita dall'implementazione (corredata da test estensivi). Per quanto riguarda l'analisi si può procedere sia con metodologia *bottom-up* sia con metodologia *top-down*. Queste due metodologie non sono mutuamente esclusive, ma possono essere usate entrambe nell'ambito dello stesso progetto tipicamente in fasi diverse della risoluzione di un problema. Solitamente in fase di analisi conviene scomporre il problema in più componenti base, mentre in fase di progettazione conviene procedere alla composizione di questi sottoparti al fine di ottenere il risultato finale.

La risoluzione di un problema implica sempre la creazione di un modello della realtà, spesso usando le metafore od i concetti tipici del linguaggio che verrà poi usato per l'implementazione. Questo accade poiché l'uso di un linguaggio spinge a ragionare in termini dei suoi costrutti. Ad esempio programmando in C difficilmente la soluzione farà uso di concetti estranei al C, come oggetti ed ereditarietà.
L'insieme delle metafore e dei concetti legati ad un linguaggio è chiamato **spazio concettuale**, e contiene sia i concetti legati all'ambiente supposto dal linguaggio (file, interfacce, processi, ...) sia i concetti propri del linguaggio stesso (funzioni, oggetti, ...). Poiché gli spazi concettuali possono essere differenti diversa sarà anche la visione del mondo durante l'utilizzo di questi linguaggi, portando a tecniche di progettazione differenti. Questo porta alla scelta di uno

specifico linguaggio per la risoluzione di uno specifico problema, in quanto ogni linguaggio è in grado di esprimere meglio certi particolari aspetti e concetti.

## 1.2 Linguaggi

I linguaggi nascono da sorgenti di ispirazione in genere molto comuni.

- I linguaggi imperativi nascono direttamente dall'architettura di Von Neumann in quanto rappresentano in astratto le singole istruzioni che il processore dovrà eseguire per portare a termine un determinato compito, non lasciando alla macchina alcuna libertà di esecuzione. Questi linguaggi esprimono il *controllo* sulla macchina, di conseguenza sono error prone e difficili da seguire o debuggare
- I linguaggi logici ed i linguaggi funzionali invece nascono dalla logica matematica, consentendo più ampi gradi di libertà. Questi linguaggi in genere hanno un approccio dichiarativo, ovvero il programmatore esprime il risultato desiderato senza preoccuparsi eccessivamente delle modalità con cui verrà ottenuto. Di conseguenza il programma diviene più semplice da leggere e la programmazione si basa sulla delegazione delle responsabilità, permettendo di concentrarsi sull'obiettivo e non sulla specifica modalità di risoluzione

## 1.3 Stili di programmazione

Un automa concepito secondo uno qualunque di questi formalismi può supportare linguaggi ispirati a un qualsiasi modello computazionale, tuttavia ogni modello computazionale promuove uno specifico stile di programmazione, caratterizzante per i linguaggi di tale famiglia. L'adozione di uno di questi stili ha conseguenze sulle proprietà dei programmi scritti, sulla metodologia di risoluzione dei problemi e sulle caratteristiche dei sistemi software (efficienza, modificabilità, manutenibilità ...).
Poiché ogni modello computazionale ha diversi punti di forza ha senso usarne più d'uno, ad esempio sviluppando parti diverse della medesima applicazione con modelli diversi. Per far ciò è sempre più comune ricorrere a linguaggi "blended", ovvero che uniscono più modelli computazionali, con l'obiettivo di prendere il meglio da ognuno dei mondi (es. oggetti + funzionale).

# Capitolo 2

# Linguaggi e Macchine Astratte

**Definizione 2.1: algoritmo**

Si dice **algoritmo** una sequenza **finita** di istruzioni con lo scopo di risolvere una *classe* di problemi in un tempo finito. Ogni algoritmo deve essere poi descritto mediante un insieme ordinato di frasi appartenenti ad un linguaggio (di programmazione) che specificano quali siano le azioni da svolgere

**Definizione 2.2: programma**

Si dice programma un testo scritto in accordo alla sintassi e alla semantica di un linguaggio di programmazione

Non è detto che un **programma** sia un algoritmo, in quanto è sufficiente che il programma non termini (tempo di esecuzione infinito, basta pensare ad un server) o che non dia un risultato specifico per violare la definizione di algoritmo. Sebbene siano differenti, un programma è utile nei momenti in cui è necessario agire in un determinato modo (magari sulla base di uno o più algoritmi).
In questo senso un algoritmo esprime la soluzione ad un problema a partire da un insieme di dati in ingresso, mentre un programma ne è la formulazione scritta in un linguaggio di programmazione.

L'esecuzione delle azioni descritte dall'algoritmo presuppone l'esistenza di una macchina astratta in grado di eseguirlo (**automa esecutore**), ma poiché le istruzioni facenti parte dell'algoritmo provengono dall'esterno e poiché esse sono scritte in un particolare linguaggio l'automa deve essere un **interprete di linguaggio** (chiamato **linguaggio macchina**). Inoltre, poiché l'automa deve essere realizzabile fisicamente, se è composto da parti esse devono essere in *numero finito* e i dati in ingresso ed in uscita devono essere esprimibili tramite un numero *finito* di simboli.

Formalizzando il concetto di automa si giunge a definire il concetto stesso di **computabilità**. Esistono vari approcci per fornirne una definizione:

- Approccio a *gerarchia di macchine astratte*: si costruiscono via via macchine sempre più potenti determinando alla fine il concetto di computabilità sulla base delle capacità computazionali di queste macchine
- Approccio *funzionale*: fondato sul concetto di funzione matematica, ha come scopo quello di capire quali funzioni siano computabili

- Sistemi di *riscrittura*: l'automa è descritto come un insieme di regole di riscrittura (successive) che trasformano frasi in altre frasi equivalenti tramite cui è possibile giungere ad una soluzione del problema

## 2.1 Gerarchia di macchine astratte

È possibile definire una gerarchia di macchine in quanto è possibile che macchine diverse abbiano potenzialità diverse e che siano in grado di risolvere problemi diversi, generalmente via via più complessi salendo verso la sommità della gerarchia. Di conseguenza è utile saper scegliere quale sia la macchina più adatta a risolvere il problema corrente.

Dato un problema non è detto che esso sia risolvibile (in generale o da una macchina specifica), se nemmeno la macchina più potente è in grado di risolverlo allora potrebbe essere che questo problema non possa essere risolto in assoluto (da una macchina almeno). A questo punto diventa importante conoscere quali siano i problemi risolvibili e quali non lo siano (più nello specifico quale sia la macchina migliore per poter risolvere un problema).

- Macchina base (combinatoria): essa è formalmente definita dalla tripla

  $$\langle I, O, mfn \rangle$$

  dove:

  - $I$ è l'insieme finito dei simboli di ingresso
  - $O$ è l'insieme finito dei simboli di uscita
  - $mfn : I \to O$ è la funzione di macchina

  Questo tipo di macchina non è dotato del concetto di *storia passata*, pertanto è in grado solamente di risolvere problemi in tempo reale. Inoltre è necessario enumerare esplicitamente tutte le possibile configurazioni delle uscite sulla base degli ingressi

- Automa a stati finiti: è formalmente definito da

  $$\langle I, O, S, mfn, sfn \rangle$$

  dove:

  - $I$ è l'insieme finito dei simboli di ingresso
  - $O$ è l'insieme dei finito simboli di uscita
  - $S$ è l'insieme finito degli stati
  - $mfn : I \times S \to O$ è la funzione di macchina
  - $sfn : I \times S \to S$ è la funzione di stato

  L'automa consente di tener traccia della storia passata mediante un numero finito di stati interni, di conseguenza le operazioni future sono influenzate da quelle precedenti. Inoltre il concetto di stato consente di tener traccia dell'ordine con cui vengono eseguite le azioni.

  È possibile avere rappresentazioni differenti di un automa a stati (Mealy/Moore, sincroni/asincroni, ...), che tuttavia possono anche essere equivalenti tra di loro.

  Il principale limite di questi automi sta nella finitezza della loro memoria: poiché essa non è illimitata sorgono problemi ogni qual volta non sia possibile determinarne a priori la quantità necessaria (es. bilanciamento delle parentesi)

- Push Down Automaton

- Macchina di Turing

## 2.2 Macchina di Turing

Al fine di superare il limite della memoria finita si introduce un nastro esterno come supporto di memorizzazione. Questo nastro non è infinito ma ha la possibilità di essere esteso indefinitamente secondo la necessità. Di conseguenza la MDT è una macchina a stati la cui memoria è esterna; essa rappresenta un potente modello matematico per la comprensione di quali problemi possano essere risolti e quali no.

Questa macchina è dotata di una testina in grado di leggere simboli dal nastro, scrivere il risultato dato dalla $mfn$ sul nastro, transitare in un nuovo stato sulla base della $sfn$ e spostarsi lungo il nastro sulla base del risultato della $dfn$.
Una MDT è definita univocamente dalla quintupla

$$\langle A, S, mfn, sfn, dfn \rangle$$

dove:

- $A$ è l'insieme finito dei simboli di ingresso ed uscita (l'alfabeto). Si utilizza un unico insieme in quanto i risultati prodotti dalla MDT devono poter essere nuovamente letti dalla stessa macchina per continuare ad operare
- $S$ è l'insieme finito degli stati caratterizzanti la macchina, tra i quali vi è necessariamente lo stato $HALT$ (indica che la macchina ha finito di operare e che quindi è possibile leggere il risultato delle operazioni)
- $mfn : A \times S \to A$ è la funzione di macchina
- $sfn : A \times S \to S$ è la funzione di stato
- $dfn : A \times S \to D = \{Left, Right, None\}$ è la funzione di direzione

Risolvere un problema con una MDT si riduce quindi alla definizione di una rappresentazione di partenza dei dati (definizione dell'insieme $A$) ed alla definizione delle funzioni che ne determinano il comportamento.

**Esempio 2.1: riconoscimento di un palindromo**

Un semplice algoritmo per la macchina di Turing consiste nei seguenti passi:

1. Leggere il primo simbolo a sinistra
2. Ricordare se tale simbolo sia uno 0 oppure un 1 e marcare la casella come già visitata
3. Spostarsi sull'ultimo simbolo a destra
4. Se tale simbolo non coincide con quello ricordato scrivere un simbolo d'errore e terminare, altrimenti marcare la casella come visitata e tornare al primo simbolo a sinistra non ancora visitato
5. Ripetere dal primo punto

Se, giunti al passo 4 sono stati consumati tutti i simboli allora la frase è palindroma, in caso contrario non lo è.
Questo algoritmo può essere espresso tramite il seguente alfabeto ed i seguenti stati

$$A = \{0, 1, \bullet, \boxtimes, \boxtimes\}$$

$$S = \{HALT, s0, s1, s2, s3, s4, s5\}$$

La funzione di macchina può essere espressa tramite la seguente tabella:

| | $s0$ | $s1$ | $s2$ | $s3$ | $s4$ | $s5$ |
|---|---|---|---|---|---|---|
| 0 | 0 | • | 0 | 0 | • | ⊠ |
| 1 | 1 | • | 1 | 1 | ⊠ | • |
| • | • | ☑ | • | • | ☑ | ☑ |

La funzione di stato può essere espressa tramite la seguente tabella:

| | $s0$ | $s1$ | $s2$ | $s3$ | $s4$ | $s5$ |
|---|---|---|---|---|---|---|
| 0 | $s0$ | $s2$ | $s2$ | $s3$ | $s0$ | $HALT$ |
| 1 | $s0$ | $s3$ | $s2$ | $s3$ | $HALT$ | $s0$ |
| • | $s1$ | $HALT$ | $s4$ | $s5$ | $HALT$ | $HALT$ |

La funzione di direzione può essere espressa tramite la seguente tabella:

| | $s0$ | $s1$ | $s2$ | $s3$ | $s4$ | $s5$ |
|---|---|---|---|---|---|---|
| 0 | $L$ | $R$ | $R$ | $R$ | $L$ | $N$ |
| 1 | $L$ | $R$ | $R$ | $R$ | $L$ | $L$ |
| • | $R$ | $N$ | $L$ | $L$ | $N$ | $N$ |

Ad ogni passo la macchina può spostare la testina di una posizione, dunque eseguendo un numero arbitrario di passi è possibile spostare la testina di $N$ passi. Questa capacità di movimento non ha vincoli, dunque la macchina può leggere e scrivere qualsiasi cella

A questo punto ci si può chiedere se effettivamente la macchina si fermi dando un risultato corretto ogni volta per ogni problema. Ipotizzando di non aver commesso errori durante la definizione delle funzioni e dei simboli è possibile che la macchina entri in un loop infinito non dando mai la risposta al problema che le è stato sottoposto. Sebbene sembri una limitazione, la possibilità di entrare in loop infiniti è positiva in quanto deriva dalla potenza della macchina stessa, infatti solo se la macchina fosse meno potente, e quindi fosse in grado di risolvere meno problemi, si eviterebbe questa evenienza.

**Tesi 2.1: di Church-Turing**

Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella risolta dalla Macchina di Turing

Sebbene sia una tesi, quindi mai dimostrata o confutata, essa significa che la MDT è la macchina più potente che si possa immaginare. Ciò è importante in quanto permette di capire quali siano i limiti della computazione.

## 2.3 Push down automaton

Non sempre è necessario utilizzare una macchina di Turing per la risoluzione dei problemi. Per alcune categorie di problemi infatti può essere sufficiente l'utilizzo di altre categorie di macchine più semplici con un modello di memoria più limitato. Infatti sebbene la MDT sia molto potente ha alcuni problemi di efficienza dovuti alla possibilità di accedere ad una qualsiasi cella del nastro. Un'alternativa più efficiente è il *push down automaton* (PDA o macchina a stack) che, sfruttando il principio di località, ha la possibilità di accedere solamente alla cella in cima alla pila.

## 2.4 Macchine specifiche e macchine universali

Una volta definita la parte di controllo, la MDT è in grado di risolvere uno specifico problema, ma così facendo la MDT stessa diventa specifica per quel dato problema. Una macchina specifica ha vari vantaggi, ad esempio può essere prodotta facilmente per il mercato di massa ad un basso costo. Tuttavia nel caso volessimo ottenere una macchina in grado di risolvere *qualsiasi* (o almeno il maggior numero possibile) problema occorrerebbe progettare una **MDT universale**.

Infatti se la macchina non avesse lo specifico algoritmo cablato al suo interno ma fosse in grado di cercarlo sul nastro e poi di eseguirlo otterremmo proprio una MDT universale. Tale macchina sarebbe dunque in grado di eseguire un compito differente a seconda del contenuto del nastro di memoria permettendone una maggior versatilità. L'unico algoritmo cablato nella macchina sarebbe quello per la lettura dal nastro dell'algoritmo da eseguire. Una macchina di questo tipo dovrebbe essere inoltre in grado di leggere ed interpretare il linguaggio in cui viene scritto l'algoritmo sul nastro, rendendola di fatto un *interprete di linguaggio*, e rappresenta dunque l'automa esecutore descritto nel capitolo precedente.

### 2.4.1 Macchina di Turing universale ed architettura di Von Neuman

Una MDT universale modella dunque il concetto di elaboratore di uso generale, confrontabile con la macchina di Von Neuman, dotato delle fasi di:

1. **fetch**: ricerca delle istruzioni da svolgere
2. **decode**: interpretazione delle istruzioni
3. **execute**: esecuzione delle istruzioni

La scrittura di un simbolo sul nastro si traduce nella scrittura di un dato sulla RAM (equivalentemente per la lettura). La transizione in un nuovo stato della MDT universale si traduce in una nuova configurazione dei registri della CPU, mentre il movimento lungo il nastro si traduce nella possibilità di accedere ad una qualsiasi delle celle di memoria.

Nonostante le similitudini la MDT universale non è in grado di catturare tutti gli aspetti dell'architettura di Von Neumann in quanto per la MDT universale non esiste il concetto di "mondo esterno" né di istruzioni di I/O. In altre parole la MDT universale modella unicamente l'aspetto computazionale dei problema ignorando l'aspetto interazionale.

### 2.4.2 Computazione ed interazione

Computazione ed interazione sono due aspetti distinti ed ortogonali, espressi da due linguaggi diversi:

- Linguaggio di computazione: esprime le primitive necessarie all'elaborazione delle informazioni
- Linguaggio di interazione: esprime le primitive per l'interazione con il mondo esterno
  - Linguaggio di coordinazione: fornisce le primitive di comunicazione (come dire le cose)
  - Linguaggio di comunicazione: definisce le informazioni che verranno trasmesse (cosa dire)

# Capitolo 3

# Introduzione alla teoria della computazione

Poiché la tesi di Church-Turing postula l'impossibilità di creare un automa più potente della macchina di Turing si può dedurre che se nemmeno la MDT sia in grado di risolvere un problema (ovvero non è in grado di fermarsi fornendo una risposta), allora quel problema è irresolubile.

**Definizione 3.1: problema risolubile**

Si definisce problema risolubile un problema la cui soluzione può essere espressa da una MDT (o da un formalismo equivalente)

Poiché però una MDT è in grado di computare solamente funzioni e non problemi occorre creare un ponte tra i due mondi definendo una funzione (chiamata **funzione caratteristica di un problema**) che associ ad ogni dato in ingresso del problema la corrispondente risposta corretta:

**Definizione 3.2: funzione caratteristica di un problema**

Dato un problema $P$ e detti $X$ l'insieme dei suoi dati in ingresso ed $Y$ l'insieme delle risposte corrette, si dice funzione caratteristica del problema $P$:

$$F_p : X \to Y$$

In questo modo un problema (ir-)resolubile diventa sinonimo di una funzione caratteristica (non) computabile.

**Definizione 3.3: funzione computabile**

Si definisce **funzione computabile** una funzione $f : A \to B$ per cui esiste una MDT in grado di produrre un risultato in un numero finito di passi

Poiché la tesi di Church-Turing implica che i problemi che non possono essere risolti con una MDT non sono risolvibili da nessun'altra macchina occorre comprendere se tutte le funzioni siano computabili oppure se esistano funzioni definibili ma non computabili. Per semplicità ci limitiamo a considerare solo le funzioni sui numeri naturali. Questo non è limitativo in quanto, poiché ogni informazione è finita, essa può essere codificata tramite una collezione di numeri naturali, la quale a sua volta può essere espressa tramite un unico numero naturale tramite il procedimento di Gödel.

**Definizione 3.4: procedimento di Gödel**

Data una collezione di numeri naturali $(N_1, N_2, \ ... \ , N_k)$ e dati i primi $k$ numeri primi si ottiene il valore $R$ secondo il procedimento di Gödel:

$$R = \prod_{i=1}^{k} P_i^{N_i}$$

Dall'unicità della scomposizione in fattori primi di un valore naturale deriva il fatto che $R$ identifichi univocamente la collezione originale

Limitandoci alle sole funzioni sui naturali è noto dall'analisi matematica che l'insieme rappresentante le funzioni definibili su $\mathbb{N}$

$$F = \{f : \mathbb{N} \to \mathbb{N}\}$$

non è numerabile. Tuttavia l'insieme delle funzioni computabili è numerabile in quanto la cardinalità dei simboli dell'alfabeto e degli stati è finita e associabile univocamente ad un intero tramite il procedimento di Gödel. Ciò significa che gran parte delle funzioni definibili non sono computabili.

Sebbene ciò sia vero, ai fini pratici sono interessanti solo quelle funzioni definibili per mezzo di un linguaggio basato su un alfabeto finito di simboli, da cui deriva che anche l'insieme delle funzioni definibili (utili ai fini pratici) è numerabile. Purtroppo i due insiemi non coincidono, in quanto vi sono delle funzioni definibili ma non computabili, ovvero vi sono problemi irresolubili da qualsiasi MDT, come il "problema dell'halt".

## 3.1 Problema dell'halt

Il problema consiste nello stabilire se una data macchina di Turing con un generico ingresso $X$ si ferma oppure no. Il problema è facilmente definibile ma nel caso generale non è computabile.

**Dimostrazione 3.1: non computabilità del problema dell'halt**

Data una macchina di Turing $m \in M$ e un generico ingresso $x \in X$ definiamo la funzione caratteristica

$$f_{halt}(m, x) = \begin{cases} 1 & \text{se } m \text{ con ingresso } x \text{ si ferma} \\ 0 & \text{se } m \text{ con ingresso } x \text{ non si ferma} \end{cases}$$

Questa funzione è definibile, ma calcolarla conduce ad un assurdo.
Se questa funzione fosse computabile allora dovrebbe per forza esistere una qualche MDT in grado di calcolarla. Definiamo quindi la funzione $g_{halt}$ avente come unico parametro una generica macchina di Turing $n$:

$$g_{halt}(n) = \begin{cases} 1 & \text{se } f_{halt}(n, n) = 0 \\ \perp & \text{se } f_{halt}(n, n) = 1 \end{cases}$$

In sostanza $g_{halt}$ si ferma solo quando la MDT $n$ con ingresso $n$ non si ferma, mentre non si ferma quando la MDT $n$ con ingresso $n$ si ferma. Prendiamo ora come ingresso $n_g$, ovvero proprio la MDT che calcola $g_{halt}$. Da ciò deriva un *assurdo* poiché sostituendo:

$$g_{halt}(n_g) = \begin{cases} 1 & \text{se } f_{halt}(n_g, n_g) = 0 \\ \perp & \text{se } f_{halt}(n_g, n_g) = 1 \end{cases}$$

Si ottiene che:

- Se la MDT si ferma (poiché $g_{halt}$ dà come risultato 1), $f_{halt}$ dice che **non** avrebbe dovuto fermarsi (poiché vale 0)
- Se la MDT **non** si ferma (poiché $g_{halt}$ è indefinita), $f_{halt}$ dice che avrebbe dovuto fermarsi (poiché vale 1)

Questo particolare problema è quindi **indecidibile**, ovvero non computabile.

## 3.2 Generabilità e decidibilità

Poiché un linguaggio è un insieme di frasi, risulta importante indagare in generale il problema di generabilità e decidibilità di un insieme.

L'analisi matematica introduce il concetto di **insieme numerabile**, ovvero di insiemi a cui può essere associata una funzione $F$ in grado di metterne in corrispondenza gli elementi con i numeri naturali. Tuttavia quest'unica definizione non è sufficiente, in quanto è necessario che tale funzione sia anche computabile affinché l'insieme sia effettivamente generabile da una macchina di Turing. Si introduce quindi il concetto di insieme **ricorsivamente enumerabile**.

**Definizione 3.5: insieme ricorsivamente enumerabile**

Un insieme è ricorsivamente numerabile (o semidecidibile) se l'insieme è numerabile e la funzione $F$ è anche computabile

In questo ultimo caso una MDT potrà inoltre generare l'insieme elemento per elemento, rendendo l'insieme **effettivamente generabile**.

Va notato, tuttavia, che il fatto che un insieme sia generabile non implica il fatto che sia possibile decidere se un generico elemento appartenga o meno all'insieme.

**Esempio 3.1**

Supponiamo di costruire una MDT in grado di generare tutti i numeri pari e di chiederle se il numero $x$ sia pari. Questa comincerà a generare tutti i numeri pari e, uno ad uno, li confronterà con $x$ nella speranza che coincidano. Supponiamo ora che il numero $x$ sia dispari. L'algoritmo appena descritto rende impossibile il raggiungimento dello stato di $HALT$ da parte della macchina, in quanto la MDT continuerà a generare numeri pari all'infinito non trovandone mai uno uguale ad $x$

Il problema appena descritto è definito **problema semidecidibile**, infatti è possibile ricavare la soluzione solo per una restrizione dell'insieme dei possibili input (si riesce a decidere se "appartiene" in positivo, ma non se "non appartiene"). Occorre quindi un concetto più potente della semidecidibilità di un insieme, ovvero la **decidibilità**.

**Definizione 3.6: insieme decidibile**

Un insieme $S$ è decidibile se la sua funzione caratteristica è computabile:

$$f(x) = \begin{cases} 1 & x \in S \\ 0 & x \notin S \end{cases}$$

Ovvero se esiste una macchina di Turing in grado di comprendere, in un numero finito di passi, se un elemento appartenga o meno all'insieme $S$

Riprendendo l'esempio precedente, per ottenere questo effetto è sufficiente fare generare alla macchina di Turing anche l'insieme dei numeri dispari.

**Teorema 3.1**

Se un insieme è decidibile allora è anche semidecidibile ma non viceversa

**Teorema 3.2**

Un insieme $S$ è decidibile se e solo se sia $S$ che il suo complemento $U \setminus S$ sono semidecidibili

Tutto questo è importante in quanto i linguaggi sono costruiti a partire da un alfabeto finito e sono caratterizzati dall'insieme infinito delle sue frasi lecite. Non basta che tale insieme di frasi possa essere generato (semidecidibilità), ma è indispensabile poter decidere se una frase sia lecita o meno senza entrare in un loop infinito (decidibilità).

# Capitolo 4

# Linguaggi e Grammatiche

**Definizione 4.1: linguaggio**

Insieme di parole e di metodi di combinazione delle parole usate e comprese da una comunità di persone

Questa definizione, sebbene corretta dal punto di vista formale e linguistico, presenta vari problemi all'atto pratico. Per prima cosa è una definizione poco precisa che si presta ad interpretazioni differenti, che quindi non evita le ambiguità del linguaggio naturale. Inoltre essa mal si presta alla descrizione di processi computazionali meccanizzabili né aiuta a descriverne le proprietà e le caratteristiche. Di conseguenza occorre fornire una definizione formale di *linguaggio*, inteso come sistema formale, che consenta di discriminare facilmente le frasi lecite da quelle illecite e che consenta di determinare agilmente il significato di ogni frase in maniera non ambigua.
Un linguaggio possiede:

- **Sintassi**: insieme di regole formali per la scrittura di programmi in un linguaggio che dettano le modalità per costruire frasi corrette nel linguaggio stesso. Solitamente questa viene espressa tramite notazioni formali quali BNF o EBNF oppure tramite diagrammi sintattici
- **Semantica**: insieme dei significati da attribuire alle frasi (sintatticamente corrette) costruite nel linguaggio. Una frase, infatti, può essere sintatticamente corretta ma non avere alcun significato. La semantica di un linguaggio è esprimibile in vari modi:
  - A parole, presenta le ambiguità del linguaggio naturale che rendono la definizione poco precisa e mal interpretabile
  - Mediante azioni, ovvero mediate **semantica operazionale** (descrivere passo passo cosa fa una certa istruzione)
  - Mediante funzioni matematiche, ovvero mediante **semantica denotazionale**
  - Mediante formule logiche, ovvero mediante **semantica assiomatica**

## 4.1 Interpretazione e compilazione

Una volta definito un linguaggio è possibile costruire un compilatore o un interprete in modo da renderlo eseguibile:

- **Compilatore**: accetta in ingresso un intero programma scritto nel linguaggio $L$ e lo riscrive in un altro linguaggio (generalmente più semplice). Il risultato è quindi una riscrittura della macro-frase

- **Interprete**: accetta in ingresso le singole frasi del linguaggio e le esegue una per volta, restituendone una valutazione

Interprete e compilatore dunque sono simili per molti versi, la differenza sta principalmente nella dimensione dei dati in ingresso e nel risultato dell'elaborazione. Entrambi gli elaboratori effettuano analisi di vario tipo:

1. **Analisi lessicale**: consiste nell'individuazione delle singole parole di una frase. L'analizzatore lessicale (*lexer* o *scanner*), data una sequenza di caratteri li aggrega in token di opportune categorie stabilendo se tutti i token appartengano al linguaggio (controlla errori di ortografia sul linguaggio)
2. **Analisi sintattica**: consiste nella verifica che la frase, intesa come sequenza di token, rispetti le regole grammaticali del linguaggio. L'analizzatore sintattico (*parser*) data una sequenza di token prodotta dallo scanner genera una rappresentazione interna della frase (generalmente sotto forma di albero)
3. **Analisi semantica**: consiste nel determinare il significato semantico di una frase. L'analizzatore semantico, sulla base della rappresentazione intermedia prodotta dal parser, controlla la coerenza logica della frase (es. se le variabili sono usate solo dopo la loro definizione). Eventualmente può trasformare ulteriormente la rappresentazione delle frasi per agevolare la generazione di codice

### 4.1.1 Significato di una frase

Chiedersi quale sia il significato di una frase, per una persona, significa associare a quella frase un concetto nella nostra mente, concetto determinato dalla nostra esperienza di vita e in base alla nostra cultura. All'atto pratico deve esistere una funzione in grado di associare ad ogni frase appartenente al dominio un concetto appartenente al codominio, assegnando significato prima ad ogni simbolo appartenente all'alfabeto usato, poi ad ogni parola (intesa come sequenza lecita di caratteri) ed infine ad ogni frase (intesa come sequenza lecita di parole).

## 4.2 Linguaggi

**Definizione 4.2: alfabeto**

Insieme finito e non vuoto di simboli atomici (es. $A = \{a, b\}$)

**Definizione 4.3: stringa**

Sequenza di simboli appartenenti all'alfabeto, ovvero un elemento del prodotto cartesiano $A^n$. Ogni stringa possiede una lunghezza, eventualmente nulla (in questo caso si parla della stringa vuota $\epsilon$ generata dal prodotto cartesiano $A^0$)

**Definizione 4.4: linguaggio $L$ su un alfabeto $A$**

Insieme di stringhe su A

**Definizione 4.5: frase di un linguaggio**

Ognuna delle stringhe che compongono un linguaggio

**Definizione 4.6: cardinalità di un linguaggio**

Numero di frasi ammesse da un particolare linguaggio; in generale interessa sapere soltanto se essa sia *finita* o *infinita*

**Definizione 4.7: chiusura ($A^*$) di un alfabeto $A$**

Insieme infinito di tutte le stringhe composte con i simboli appartenenti ad $A$

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

**Definizione 4.8: chiusura positiva ($A^+$) di un alfabeto $A$**

Insieme infinito di tutte le stringhe non nulle composte con i simboli appartenenti ad $A$

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

Per specificare il sottoinsieme di $A^*$ che costituisce un linguaggio si possono adottare diversi approcci a seconda che il linguaggio abbia cardinalità finita o infinita. Nel caso la cardinalità sia finita è sufficiente enumerare tutte le frasi lecite, mentre se la cardinalità e infinita è necessario utilizzare una qualche notazione in grado di descrivere in modo finito un insieme infinito di elementi, ovvero serve una grammatica formale.

## 4.3 Grammatiche

Si nota che per esprimere il sotto insieme di $A^*$ che costituisce un linguaggio finito è sufficiente elencare tutti gli elementi che lo compongono, tuttavia per rappresentare un linguaggio infinito occorre ricorrere ad altre notazioni in grado di fornire questa descrizione al meglio. Nasce così il concetto di grammatica formale.

**Definizione 4.9: grammatica**

Una grammatica è una notazione formale con cui esprimere in modo rigoroso la sintassi di un linguaggio. Formalmente è esprimibile tramite la quadrupla

$$\langle V_T, V_N, P, S \rangle$$

dove:

- $V_T$ è l'insieme finito dei simboli *terminali*, ovvero dei simboli che verranno ritrovati alla fine della procedura di analisi. I simboli possono essere caratteri e stringhe su un alfabeto $A$
- $V_N$ è l'insieme finito dei simboli *non terminali*, ovvero dei simboli usati per le procedure di analisi e di sostituzione. Questi simboli sono dei *metasimboli* che rappresentano diverse *categorie sintattiche*
- $P$ è l'insieme finito di produzioni, ovvero di regole di riscrittura $\alpha \to \beta$ dove $\alpha$ e $\beta$ sono stringhe tali che $\alpha \in V^+$ e $\beta \in V^*$, ovvero ogni regola esprime una trasformazione lecita che permette di scrivere una stringa non vuota al posto di un'altra in un dato contesto
- $S$ è un particolare simbolo non terminale detto *simbolo iniziale* o *scopo* della grammatica

Si noti che gli insiemi $V_T$ e $V_N$ devono essere disgiunti ($V_T \cap V_N = \emptyset$). L'unione di $V_T$ e $V_N$ ($V = V_T \cup V_N$) invece viene detta vocabolario della grammatica

Nelle formule teoriche per convenzione si indicano i simboli terminali tramite lettere minuscole, i simboli non terminali tramite lettere maiuscole e le stringhe mixed tramite lettere greche.

**Definizione 4.10: forma di frase**

Si dice **forma di frase** una qualsiasi stringa comprendente sia simboli terminali sia metasimboli ottenibile tramite una o più regole di produzione. In questo senso un forma di frase è un prodotto intermedio

**Definizione 4.11: frase**

Si dice **frase** una forma di frase composta solo da simboli terminali. In questo senso una frase è un prodotto finale, senza più sezioni trasformabili

**Definizione 4.12: derivazione**

Siano $\alpha, \beta \in (V_N \cup V_T)^*$, $\alpha \neq \epsilon$ due stringhe. Si dice che $\beta$ deriva direttamente da $\alpha$ ($\alpha \to \beta$) se:

- Le stringhe $\alpha, \beta$ si possono decomporre in: $\alpha = \eta A \delta$, $\beta = \eta \gamma \delta$
- Esiste la produzione $A \to \gamma$

Si dice che $\beta$ deriva da $\alpha$ se esiste una sequenza di $N$ produzioni dirette che da $\alpha$ possono produrre $\beta$

**Definizione 4.13: sequenza di derivazione**

Si dice **sequenza di derivazione** la sequenza di passi che producono una forma di frase $\sigma$ dallo scopo $S$. In questo caso non è tanto importante quali produzioni siano state applicate, quanto quale sia il risultato finale

- $S \Rightarrow \sigma$ $\sigma$ deriva da $S$ con una sola applicazione di produzioni, ovvero in un solo passo
- $S \overset{+}{\Rightarrow} \sigma$: $\sigma$ deriva da $S$ con una o più applicazioni di produzioni, ovvero in uno o più passi
- $S \overset{*}{\Rightarrow} \sigma$: $\sigma$ deriva da $S$ con zero o più applicazioni di produzioni, ovvero in zero o più passi

**Definizione 4.14: linguaggio generato da una grammatica**

Data una grammatica $G$ si dice **Linguaggio $L_g$ generato da** $G$ l'insieme delle frasi derivabili dal simbolo iniziale $S$ tramite applicazioni delle produzioni $P$

$$L_g = \{s \in V_T^* : S \overset{*}{\Rightarrow} s\}$$

**Definizione 4.15: grammatiche equivalenti**

Date due grammatiche esse si dicono equivalenti se generano lo stesso linguaggio

Tuttavia, sebbene siano equivalenti, una potrebbe essere preferibile all'altra in termini di semplicità. Stabilire se due grammatiche siano equivalenti in generale è un problema *indecidibile* sebbene il problema diventi computabile nel caso in cui ci si restringa a tipi particolari di grammatiche.

Si noti che grammatiche diverse in termini di struttura determinano la creazione di diversi linguaggi aventi diverse proprietà. Questo implica che siano necessari diverse tipologie di automi (diversi per quanto concerne la potenza computazionale) per riconoscere tali linguaggi.

### 4.3.1 Classificazione delle grammatiche secondo Chomsky

Secondo Chomsky le grammatiche sono classificabili in 4 tipi sulla base della struttura delle produzioni usate.

**Tipo 0** Queste grammatiche non impongono alcuna restrizione sulle produzioni. In particolare è possibile utilizzare anche produzioni che accorciano la lunghezza della forma di frase corrente. Per manipolare questo tipo di grammatica è necessaria una macchina di Turing, ma non è detto che ne esista una in generale. Questo è dovuto al fatto che viene data la possibilità di accorciare la lunghezza delle stringhe a piacere.

**Esempio 4.1: grammatica di tipo 0**

$$\begin{aligned} S &\to aSBC \\ CB &\to BC \\ SB &\to bF \\ FB &\to bF \\ FC &\to cG \\ GC &\to cG \\ G &\to \epsilon \end{aligned}$$

La derivazione della frase *abc* è data da:

$$S \to aSBC \to abFC \to abcG \to abc$$

**Tipo 1** Queste grammatiche sono dipendenti dal contesto e possiedono produzioni vincolate alla forma

$$\beta A \delta \to \beta \alpha \delta, \qquad \beta, \delta, \alpha \in (V_T \cup V_N)^*, A \in V_N, \alpha \neq \epsilon$$

Di conseguenza si può trasformare un metasimbolo per volta ($A$) lasciando intatto ciò che gli sta intorno (il *contesto*, appunto). Inoltre le riscritture non accorciano mai la lunghezza della forma di frase. Questa categoria permette di definire regole molto puntuali e precise. Esiste una seconda definizione, grammatica caratterizzata da produzioni del tipo:

$$\alpha \to \beta \ se \ |\beta| \geq |\alpha|$$

Quest'ultima definizione però non cattura l'idea di *contesto* della produzione, permettendo ad esempio produzioni che invertono la posizione di due meta-simboli (es. $BC \to CB$), tuttavia

esiste sempre una grammatica equivalente che rispetta la definizione di Chomsky che può essere utilizzata al suo posto (es. $BC \to BD$, $BD \to CD$, $CD \to CB$).

Data la potenza di queste grammatiche non è garantito che qualunque sequenza di derivazione porti ad una frase, infatti è possibile imboccare dei "rami morti" che non consentono di applicare altre regole rendendo impossibile proseguire nel processo e rendendo necessario tornare indietro[1]. Per questo tipo di grammatica è garantita l'esistenza di una macchina di Turing in grado di manipolarla, ma non è possibile, nel caso generale, utilizzare una macchina più semplice. L'utilizzo della macchina di Turing, sebbene renda possibile la risoluzione di tutti i problemi computabili, è estremamente inefficiente.

**Esempio 4.2: grammatica di tipo 1**

$$
\begin{aligned}
S &\to abC \mid aSBC \\
CB &\to DB \\
DB &\to DC \\
DC &\to BC \\
aB &\to ab \\
bB &\to bB \\
bC &\to bc \\
cC &\to cc
\end{aligned}
$$

Come si può notare dalle produzioni, questa grammatica consente di cambiare al più un simbolo per volta e inoltre la lunghezza della forme di frase non potrà mai accorciarsi

**Tipo 2** Queste grammatiche possiedono produzioni libere dal contesto ma vincolate alla forma:

$$A \to \alpha, \qquad \alpha \in (V_T \cup V_N)^*, A \in V_N$$

Si noti che non è più presente il vincolo riguardante la lunghezza delle stringhe, e che $A$ può essere sostituita anche con una stringa vuota. Inoltre non esistendo più l'idea di contesto le regole diventano più grossolane ma anche più semplici da elaborare. Un caso particolare si ha quando $\alpha$ è nella forma $u$ oppure $uBv$ con $u, v \in V_T^*, B \in V_N$, in questo caso la grammatica si dice **lineare**. Questo tipo di grammatica può essere processata da una macchina a stack, fornendo quindi una modalità estremamente efficiente.

**Tipo 3** Queste grammatiche si dicono grammatiche regolari, con produzioni vincolate alla forma (in maniera mutuamente esclusiva):

- **Lineare a destra** nel caso in cui le produzioni abbiano la forma:

$$A \to \sigma \mid A \to \sigma B \qquad A, B \in V_N, \sigma \in V_T^*$$

- **Lineare a sinistra** nel caso in cui le produzioni abbiano la forma:

$$A \to \sigma \mid A \to B\sigma \qquad A, B \in V_N, \sigma \in V_T^*$$

Solitamente conviene riscrivere queste grammatiche in forme **strettamente lineari** ovvero non facenti uso di forme di frase mischiate a meta-simboli, ma usando simboli terminali assieme

[1] Questo comportamento non si osserva per grammatiche di tipo 2 e 3

a meta-simboli (es. non $X \to Y\sigma$, ma $X \to Ya$, con $a \in V_T$). Questo tipo di grammatiche può essere processato da un automa a stati, di fatto necessitando della più semplice macchina con memoria possibile.

**Esempio 4.3: grammatiche lineari**

- $G1$ lineare a sinistra:

$$S \to a \mid S + a \mid S - a$$

- $G2$ lineare a destra:

$$S \to a \mid a + S \mid a - S$$

- $G3$ lineare a destra e a sinistra:

$$S \to ciao$$

- $G2$ resa strettamente lineare a destra:

$$S \to a \mid aA$$
$$A \to +S \mid -S$$

- $G3$ resa strettamente lineare a destra:

$$S \to cT$$
$$T \to iU$$
$$U \to aV$$
$$V \to o$$

Le grammatiche appartenenti a questa categoria sono poco espressive e consentono pochi gradi di libertà, tuttavia i casi che possono esprimere sono, all'atto pratico, particolarmente frequenti, dunque ricorrere a tali grammatiche risulta di vitale importanza per migliorare l'efficienza del processo di riconoscimento.

## 4.4 Il problema della stringa vuota

Le grammatiche possono essere poste in una relazione gerarchica:

- Una grammatica di tipo 3 (regolare) è un caso particolare di una grammatica di tipo 2 (senza contesto) poiché aggiunge il vincolo che le stringhe debbano crescere solamente da un lato
- Una grammatica di tipo 2 (senza contesto) è un caso particolare di una grammatica di tipo 1 (dipendente dal contesto), in quanto essa aggiunge il vincolo di poter manipolare un solo simbolo per volta (di conseguenza rendendo impossibili gli scambi tra simboli)
- Una grammatica di tipo 1 è un caso particolare di una grammatica di tipo 0, in quanto aggiunge il vincolo di non poter accorciare la lunghezza della stringa

Tuttavia va notato che se una grammatica di tipo 1 ha vincoli circa le stringhe vuote, una grammatica di tipo 2 non ne ha. Questa in apparenza è una contraddizione, ma in realtà grazie al teorema 4.1 essa non è presente.

**Teorema 4.1**

In una grammatica di tipo 2 la sostituzione di una stringa vuota non crea problemi poiché le produzione di grammatiche di tipo 2 (e di tipo 3 di conseguenza) possono sempre essere riscritte in modo da evitare la stringa vuota, mantenendo al più la produzione $S \to \epsilon$

**Teorema 4.2**

Sia $G$ una grammatica di tipo 2 contenente produzioni della forma

$$A \to \alpha, \qquad \alpha \in V^*$$

allora esiste una grammatica equivalente $G'$ le cui produzioni hanno le forme:

- $A \to \alpha$, $\alpha \in V^+$
- $S \to \epsilon$

in cui $S$ non compare nella parte destra di alcuna produzione

Tipicamente le produzioni che ammettono la stringa vuota sono comode per gli umani per scrivere in maniera più semplice insiemi di produzioni che altrimenti sarebbero complessi oppure per descrivere parametri opzionali senza dover elencare ogni combinazione possibile dei parametri opzionali. Tuttavia poiché la gestione della stringa vuota è difficoltosa per le macchine conviene adottare un approccio differente, sfruttando il precedente teorema, atto ad eliminare le produzioni che ammettono la stringa vuota prima di far elaborare la grammatica all'elaboratore. In questo modo si ottiene una grammatica:

- Semplice da leggere e scrivere per un umano
- Semplice da elaborare per una macchina

### 4.4.1 Eliminazione della stringa vuota

Per determinare la grammatica equivalente $G'$ è possibile seguire il seguente procedura:

**Algoritmo 4.1**

Data una grammatica $G$ e definiti gli insiemi $Y_\epsilon$ come l'insieme dei metasimboli da cui è possibile ricavare la stringa vuota $\epsilon$ e l'insieme $N_\epsilon$ come l'insieme dei metasimboli da cui non è possibile ricavare la stringa vuota $\epsilon$ allora:

1. Se $G$ contiene la produzione $S \to \epsilon$ allora anche $G'$ la conterrà
2. Se $G$ contiene altre regole della forma $X \to \epsilon$ allora $G'$ non le conterrà
3. Se $G$ contiene una produzione della forma $X \to c_1 c_2 \dots c_r$ con $(r \geq 1)$ allora $G'$ conterrà la produzione $X \to a_1 a_2 \dots a_r$, $(r \geq 1)$ dove:
   - $a_i = c_i$ se $c_i \in V_T \cup N_\epsilon$
   - $a_i = c_i \mid \epsilon$ se $c_i \in Y_\epsilon$

   Con il vincolo che non tutti gli $a_i$ possono essere uguali a $\epsilon$

**Esempio 4.4**

Definiamo la grammatica $G$ data dalle seguenti produzioni:

$$S \to AB \mid B$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid c$$

In questo caso avremo $Y_\epsilon = \{A\}$ e $N_\epsilon = \{B, S\}$. $G$ non contiene nessuna regola del tipo $S \to \epsilon$, quindi nemmeno $G'$ la conterrà. $G$ però contiene una regola nella forma $X \to \epsilon$ che va quindi eliminata. Esaminiamo le altre regole, la regola $A \to aA$ contiene un metasimbolo contenuto in $Y_\epsilon$, quindi questo metasimbolo va sostituito con $(A \mid \epsilon)$, ottenendo $A \to a\ (A \mid \epsilon)$ che può essere semplificato come $A \to aA \mid a$. Per la produzione $S \to AB$ può essere fatto un discorso analogo. In definitiva si ottengono le seguenti produzioni:

$$S \to AB \mid B$$
$$A \to aA \mid a$$
$$B \to bB \mid c$$

Quindi la grammatica $G'$ non genera **mai** una stringa vuota

**Esempio 4.5**

Definiamo la grammatica $G$ con le seguenti produzioni:

$$S \to AB$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid \epsilon$$

Mediante gli stessi passi si possono trovare le seguenti produzioni intermedie:

$$S \to (A \mid \epsilon)\ (B \mid \epsilon)$$
$$A \to a\ (A \mid \epsilon)$$
$$B \to b\ (B \mid \epsilon)$$

Che semplificate danno il seguente insieme finale:

$$S \to AB \mid A \mid B \mid \epsilon$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

In questo caso la nuova grammatica $G'$ può generare una stringa vuota solo al primo passaggio di derivazione, ma non nei passi intermedi. Di conseguenza il linguaggio comprende la stringa vuota, ma le forme di frase **non** possono accorciarsi mai

## 4.5 Grammatiche e linguaggi

Poiché le grammatiche sono in relazione gerarchica può accadere che un linguaggio possa essere generato da più grammatiche equivalenti, ma di tipo diverso con la seguente invariante: un linguaggio di tipo $n$ può essere generato da grammatiche di tipo $g \leq n$. Questo implica che,

poiché processare una grammatica implica l'utilizzo di un certo tipo di macchina, conviene utilizzare la grammatica più semplice per descrivere quel linguaggio tra le tante possibili, in modo da poter usare la macchina più semplice ed efficiente.

Da questo deriva anche che il tipo del linguaggio non coincide necessariamente con il tipo della specifica grammatica che lo genera, ma coincide con il tipo della grammatica **più semplice**. In questo modo dire che un linguaggio è di tipo $n$ implica che la grammatica più semplice che può generarlo è di tipo $n$.

**Esempio 4.6**

Trovare una grammatica per il linguaggio

$$L = \{a^n b^n c^n,\ n \geq 0\}$$

Per generare questo linguaggio si può far ricorso a varie grammatiche, più o meno complesse, ma la grammatica più semplice ottenibile ha tipo 1 in quanto il linguaggio consiste in tre "pezzi" che devono crescere assieme.
Una grammatica relativamente semplice per generare questo linguaggio è data dalle seguenti produzioni:

$$\begin{aligned} S &\to abc \mid aBSc \\ Ba &\to aB \\ Bb &\to bb \end{aligned}$$

Esiste una caratteristica cruciale che distingue le grammatiche di tipo 1 e 2, ossia la possibilità di *scambiare simboli* (es. $BC \to CB$). Questa caratteristica è impossibile da esprimere nelle grammatiche di tipo 2 in quanto esse non ammettono produzioni aventi due elementi sul lato sinistro, ma solo produzioni aventi un unico metasimbolo su tale lato.

Analogamente esiste una caratteristica che distingue i linguaggi di tipo 2 e di tipo 3, ovvero la possibilità di inserire metasimboli in qualsiasi posizione, anche in mezzo alla forma di frase. Questa caratteristica è impossibile da esprimere nei linguaggi di tipo 3 in quanto i metasimboli possono trovarsi solo o in testa o in coda al risultato delle produzioni. Questa caratteristica consente l'**autoinclusione** (*self embedding*), ovvero la possibilità di avere produzioni della forma

$$A \overset{*}{\Rightarrow} \alpha_1 A \alpha_2, \qquad \alpha_1, \alpha_2 \in V^+$$

il cui ruolo è l'introduzione di una ricorsione in cui si aggiungono **contemporaneamente** simboli sia a sinistra che a destra, garantendo di procedere di pari passo nella costruzione di una stringa bilanciata. Per ogni grammatica di tipo 2 che non possiede l'autoinclusione in nessuna delle sue produzioni esiste una grammatica di tipo 3 equivalente.

**Esempio 4.7**

La grammatica $G$ dotata delle produzioni

$$\begin{aligned} S &\to aSc \mid A \\ A &\to bAc \mid \epsilon \end{aligned}$$

presenta l'autoinclusione e genera il linguaggio

$$L = \{a^n b^m c^{n+m},\ n, m \geq 0\}$$

**Esempio 4.8: bilanciamento delle parentesi**

Il self-embedding è fondamentale per definire linguaggi le cui frasi devono contenere dei simboli bilanciati, come ad esempio le parentesi:

$$S \to (S)$$
$$S \to a$$

Questa grammatica genera il linguaggio

$$L = \{({}^n a)^n,\ n \geq 0\}$$

**Teorema 4.3**

Una grammatica di tipo 2 che non contenga autoinclusione genera un linguaggio regolare

Si noti che non è detto che valga il viceversa, ovvero una grammatica con autoinclusione potrebbe comunque generare un linguaggio regolare nel caso contenga delle particolari produzioni in grado di "disattivare" l'autoinclusione stessa.

**Esempio 4.9: finto self-embedding**

Prendiamo la grammatica $G$ descritta dalle produzioni

$$S \to aSa \mid X$$
$$X \to aX \mid bX \mid a \mid b$$

A prima vista sembra che la grammatica crei frasi del tipo $a^n Y a^n$, ma in realtà la parte centrale si espande come una qualsiasi sequenza di $a$ e $b$, vanificando il controllo sul numero delle $a$. Come risultato $L_G$ è un linguaggio regolare, in quanto comprende una qualsiasi sequenza di $a$ e di $b$

**Esempio 4.10: finto self-embedding**

Prendiamo la grammatica $G$ descritta dalle produzioni

$$S \to aSa \mid \epsilon$$

In questo caso l'autoinclusione è completamente inutile in quanto utilizzando un alfabeto formato da un solo carattere si possono ottenere solamente frasi estremamente semplici. Infatti distinguere tra $a$ che si trova a sinistra o $a$ che si trova a destra è inutile. Il medesimo linguaggio potrebbe essere espresso tramite un grammatica $G'$ di tipo 3

$$S \to aaS \mid \epsilon$$

**Teorema 4.4**

Ogni linguaggio context-free di alfabeto unitario è in realtà un linguaggio regolare

### 4.5.1 Riconoscibilità dei linguaggi

I linguaggi generati da grammatiche di tipo 0 possono non essere riconoscibili, tuttavia per i linguaggi generati da grammatiche di tipo 1 (e quindi anche da grammatiche di tipo 2 e 3) è

garantita l'esistenza di una macchina di Turing in grado di riconoscerlo.

Per ottenere un traduttore efficiente, tuttavia, occorre adottare dei linguaggi generati da grammatiche di tipo 2 (o da sottoinsiemi del tipo 2 con alcune restrizioni). Tutti i linguaggi di programmazione infatti sono generati da grammatiche senza contesto. Il riconoscitore per questo tipo di grammatiche è detto **parser**. Per ottenere particolare efficienza in alcune sottoparti (particolarmente ricorrenti, come identificatori e numeri) si adottano, per la loro costruzione, grammatiche di tipo 3. Il riconoscitore per questo tipo di grammatiche è detto **scanner** (o **lexer**).

| Grammatiche | Automi riconsocitori |
|---|---|
| Tipo 0 | MDT se $L$ è riconoscibile |
| Tipo 1 | MDT con nastro sufficientemente lungo |
| Tipo 2 | PDA (ASF + stack) |
| Tipo 3 | ASF |

Tabella 4.1: Corrispondenze tra grammatiche e automi riconoscitori

### 4.5.2 Notazioni e grammatiche

Per esprimere le grammatiche di tipo 2 (e quindi anche di tipo 3) è utile rifarsi alla notazione **BNF**. In questa notazione:

- le produzioni sono della forma $\alpha ::= \beta$ con $\alpha \in V^+$, $\beta \in V^*$
- i meta-simboli $X \in V_N$ hanno la forma $< nome >$
- il meta-simbolo "|" indica l'alternativa, permettendo di descrivere regole aventi la medesima parte sinistra come una serie di alternative

**Esempio 4.11**

Tale grammatica è in grado di generare la frase "il gatto mangia il topo"

$$\begin{aligned} P = \{ & < frase > ::= < soggetto >< verbo >< compl - ogg > \\ & < soggetto > ::= < articolo >< nome > \\ & < articolo > ::= il \\ & < nome > ::= gatto \mid topo \mid sasso \\ & < verbo > ::= mangia \mid beve \\ & < compl - ogg > ::= < articolo >< nome > \} \end{aligned}$$

Per esprimere in maniera più comprensibile per un umano certe produzioni è possibile utilizzare una notazione estesa che prende il nome di **EBNF** (vedi Tabella 4.2).

**Esempio 4.12: generazione dei numeri interi non negativi**

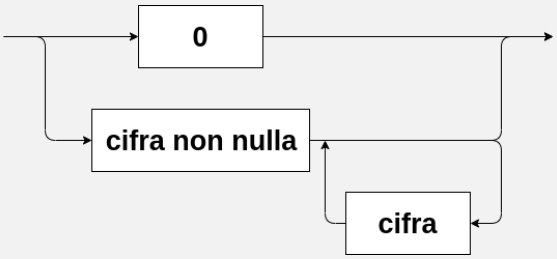
La sintassi degli interi non negativi può essere espressa in forma EBNF tramite

$$\begin{aligned} P = \{ & < num > ::= < cifra > \mid < cifra\ non\ nulla > \{< cifra >\} \\ & < cifra > ::= 0 \mid < cifra\ non\ nulla > \\ & < cifra\ non\ nulla > ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\} \end{aligned}$$

La stessa sintassi può essere espressa tramite forma grafica (diagramma sintattico)

| Forma EBNF | Forma BNF | significato |
|---|---|---|
| $X ::= [a]\ B$ | $X ::= B \mid aB$ | a può comparire 0 o 1 volta |
| $X ::= a^n\ B$ | $X ::= B \mid ab \mid \ldots \mid a^nB$ | a può comparire da 0 a $n$ volte |
| $X ::= \{a\}\ B$ | $X ::= B \mid aX$ | a può comparire 0 o più volte |
| $X ::= B\ \{a\}$ | $X ::= B \mid Xa$ | a può comparire 0 o più volte |
| $X ::= (a \mid b)\ D \mid c$ | $X ::= aD \mid bD \mid c$ | raggruppa categorie |

Tabella 4.2: Corrispondenze tra le forme EBNF e BNF



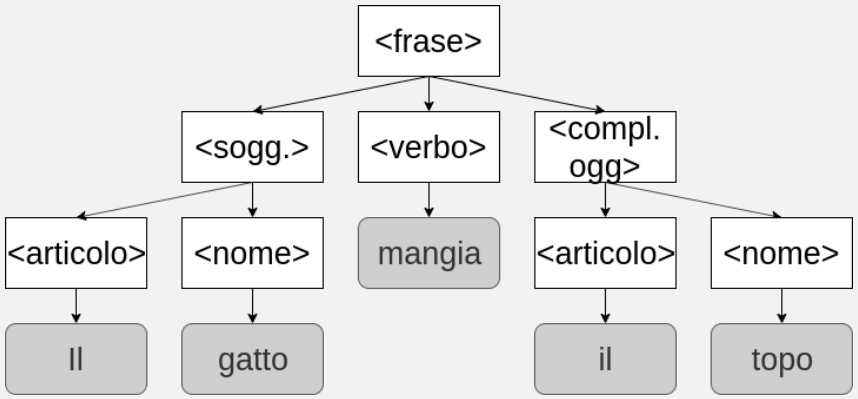

Per le grammatiche di tipo 2 si introduce il concetto di *albero di derivazione*:

- Ogni nodo dell'albero è associato ad un simbolo del vocabolario $V = V_T \cup V_N$
- La radice dell'albero coincide con lo scopo $S$
- Se $\{a_1, a_2, \ldots, a_k\}$ sono i $k$ figli ordinati di un dato nodo $X$ significa che la grammatica contiene la produzione $X ::= A_1A_2 \ldots A_k$ dove $A_i$ è il simbolo associato al nodo $a_i$

Si noti che non è possibile ottenere un albero di derivazione per grammatiche di tipo 0 e 1 poiché in esse il lato sinistro delle produzioni può avere più di un simbolo e dunque i nodi dell'albero avrebbero più di un padre (non si otterrebbe un albero, ma un grafo).

**Esempio 4.13: albero di derivazione**

Riprendendo l'esempio 4.11




Nel caso in cui ci siano delle regole EBNF occorre mapparle sulle corrispondenti BNF per poter generare l'albero di derivazione.

### 4.5.3 Derivazioni e ambiguità

**Definizione 4.16: derivazione canonica sinistra (left-most)**

Derivazione per cui a partire dallo scopo della grammatica viene riscritto sempre metasimbolo più a sinistra

**Definizione 4.17: derivazione canonica destra (right-most)**

Derivazione per cui a partire dallo scopo della grammatica viene riscritto sempre metasimbolo più a destra

**Definizione 4.18: ambiguità**

Una grammatica si dice **ambigua** se esiste almeno una frase che ammette due o più derivazioni canoniche sinistre distinte. In questo caso si dice *grado di ambiguità* il numero di alberi sintattici distinti

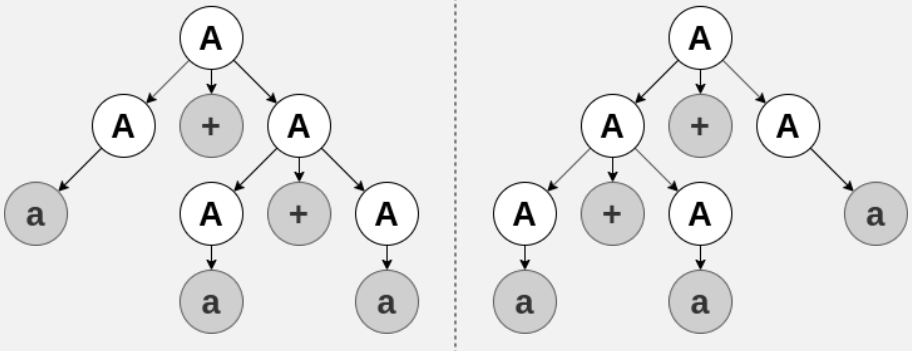
Si noti che l'ambiguità è una caratteristica non desiderabile in quanto rende possibile la scelta di più percorsi per raggiungere lo stesso punto. Nel caso di un errore infatti la macchina deputata al controllo della correttezza della frase dovrà tornare indietro di un certo numero di passi di derivazione, ma se esiste una derivazione alternativa la macchina probabilmente la riprenderà facendo diminuire le prestazioni della macchina.

**Esempio 4.14: grammatica ambigua**

Data la grammatica:

$$A ::= A + A$$
$$A ::= a$$

la frase $a + a + a$ risulta ambigua in quanto è possibile trovare due alberi di derivazione sinistri distinti




Purtroppo stabilire se una grammatica sia ambigua è un problema indecidibile, ma tipicamente osservato un numero sufficiente di derivazioni è possibile avere una buona stima del grado di ambiguità della grammatica stessa.

Nel caso una grammatica sia ambigua a volte è possibile trovare una grammatica equivalente che non lo sia, ma non sempre.

**Definizione 4.19: linguaggio intrinsecamente ambigua**

Un linguaggio si dice intrinsecamente ambiguo se tutte le grammatiche che lo generano sono ambigue

### 4.5.4 Stringa vuota

Sebbene si cerchi di lavorare sempre con grammatiche di tipo 2 e 3, talvolta risulta comodo poter utilizzare la stringa vuota per esprimere parti opzionali nelle derivazioni. Come già detto questo non crea problemi perché è sempre possibile ottenere un'altra grammatica equivalente senza regole che comprendono la stringa vuota (a parte l'eventuale produzione $S \to \epsilon$). Questa proprietà è catturata dal teorema 4.5.

**Teorema 4.5**

Dato un linguaggio $L$ di un tipo qualsiasi, i linguaggi $L \cup \{\epsilon\}$ e $L \setminus \{\epsilon\}$ sono del medesimo tipo

### 4.5.5 Forme normali

Si dimostra che un linguaggio di tipo 2 non vuoto può sempre essere generato da una grammatica di tipo 2 in cui:

- Ogni simbolo compare nella derivazione di qualche frase di $L$, ovvero non esistono simboli inutili
- Non ci sono produzioni della forma $A \to B$, con $A, B \in V_N$, ossia non esistono produzioni che cambiano nome ad un meta-simbolo
- Se il linguaggio non comprende la stringa vuota, allora non ci sono produzioni della forma $A \to \epsilon$

In particolare si può far in modo che tutte le produzioni abbiano una forma ben precisa:

**Definizione 4.20: forma normale di Chomsky**

Questa forma contiene produzioni della forma

$$A \to BC \mid a, \qquad A, B, C \in V_N,\ a \in V_T \cup \epsilon$$

**Definizione 4.21: forma normale di Greibach**

Questa forma, utilizzabile solo per i linguaggi privi della stringa vuota, contiene produzioni della forma

$$A \to a\alpha, \qquad A \in V_N,\ a \in V_T, \alpha \in V_N^*$$

Se un linguaggio ha solo produzioni in questa forma la sua elaborazione risulta fortemente agevolata, permettendo un'analisi deterministica delle frasi del linguaggio. Questo accade perché tutte le produzioni hanno come primo simbolo un simbolo terminale, quindi un riconoscitore generico è in grado a priori di capire quali produzioni applicare e quali no per ottenere il risultato desiderato

Per facilitare la costruzione di riconoscitori inoltre si può far riferimento ad alcune trasformazioni atte a modificare la struttura delle regole per renderle più adatte allo scopo, utilizzabili anche per trasformare una grammatica in una delle due forme normali. Alcune trasformazioni importanti sono:

- La **sostituzione**: consiste nell'espansione di un simbolo non terminale che compare nella parte destra di una regola di produzione, sfruttando a tale scopo un'altra produzione

**Esempio 4.15**

La grammatica data dalle produzioni

$$S \to Xa$$
$$X \to b \mid Sc$$

può essere trasformata nella grammatica

$$S \to Xa$$
$$X \to b \mid Xac$$

- Il **raccoglimento a fattor comune**: consiste nell'isolare il prefisso più lungo comune a due o più produzioni

**Esempio 4.16**

La grammatica data dalle produzioni

$$S \to aSb \mid aSc \mid b$$

può essere trasformata nella grammatica

$$S \to aS(b \mid c) \mid b$$

ed introducendo un nuovo metasimbolo

$$S \to aSX \mid b$$
$$X \to b \mid c$$

- L'**eliminazione della ricorsione sinistra**: consiste nella trasformazione di tutte le ricorsioni sinistre in ricorsioni destre, permettendo l'ottenimento di una forma normale di Greibach. Questa operazione è desiderabile in quanto la ricorsione sinistra nasconde l'inizio delle forme di frase. È articolata in due fasi distinte:

  1. Eliminazione dei cicli ricorsivi a sinistra mediante sostituzione, ovvero si modificano tutte le produzioni del tipo $X \to X\alpha$ in cui $Y > X$, sostituendo a $X$ le forme di frase delle relative produzioni
  2. Eliminazione della ricorsione sinistra diretta. Si nota che le produzioni della forma $X \to X\alpha \mid p$ possono essere riscritte introducendo il metasimbolo $Z$ e scrivendo $X \to p \mid pZ$ e $Z \to \alpha \mid \alpha Z$

**Esempio 4.17: eliminazione della ricorsione sinistra**

Data una grammatica descritta dalle seguenti produzioni

$$A \to Ba$$
$$B \to Cb$$
$$C \to Ac \mid p$$

Per prima cosa occorre stabilire una relazione d'ordine tra i metasimboli coinvolti nel ciclo ricorsivo, nel nostro caso sia $C > B > A$. Dopodiché occorre passare alla prima fase, ottenendo:

$$A \to Ba$$
$$B \to Cb$$
$$C \to Cbac \mid p$$

Dove per ottenere $C \to Cbac$ per prima cosa è stata sostituita $A$ in $Ac$ ottenendo $Bac$, successivamente è stata sostituita $B$ in questa forma intermedia ottenendo la forma finale $Cbac$.
Con il secondo passaggio infine si ottiene:

$$C \to p \mid pZ$$
$$Z \to bac \mid bacZ$$

Teoricamente la ricorsione sinistra è sempre rimovibile, tuttavia all'atto pratico non sempre ciò è desiderabile. Infatti sostituendo una ricorsione sinistra con una destra è possibile ottenere le stesse frasi ma mediante regole differenti alterando quindi il linguaggio ottenuto. Quindi se interessa solamente il risultato ai morsetti (come in un puro riconoscitore che deve solo dire se una frase appartenga o meno al linguaggio) è sempre possibile applicare questo procedimento, mentre se interessa anche come si è arrivati al risultato (come in un parser che deve dare un significato alle frasi e quindi regole diverse implicano un significato diverso per alcune frasi) questa trasformazione non è applicabile

**Esempio 4.18**

Nelle espressioni matematiche la cultura matematica comune richiede associatività sinistra. Infatti si ha $13-5-4 = (13-5)-4$ e non $13-5-4 = 13-(5-4)$. Queste regole, per essere espresse, richiedono una grammatica con ricorsione sinistra. La sostituzione della ricorsione sinistra con una ricorsione destra altera completamente il significato delle frasi, implicando un'associatività destra delle espressioni

### 4.5.6 Comprensione del tipo del linguaggio

Per comprendere di quale tipo sia un linguaggio è possibile utilizzare il *pumping lemma* per un dato tipo (PL per il tipo 3, PL per il tipo 2, ...). Questo lemma dà una condizione **necessaria** ma **non sufficiente** per determinare il tipo di un linguaggio, di conseguenza viene generalmente utilizzato per capire di quale tipo un linguaggio *non* è, ovvero se un linguaggio non rispetta il PL di tipo 3 allora quel linguaggio non è di tipo 3.

Di base il lemma si basa sul fatto che in un linguaggio infinito ogni stringa sufficientemente lunga ha delle parti che si ripetono, e che quindi può essere ripetuta un qualunque numero di volte per ottenere nuove stringhe del linguaggio.

**Lemma 4.1: pumping lemma per linguaggi di tipo 2**

Se $L$ è un linguaggio di Tipo 2, esiste un intero $N$ (dipendente dallo specifico linguaggio) tale che, per ogni stringa $z$ di lunghezza almeno pari a $N$:

- $z$ è decomponibile in 5 parti: $z = uvwxy,\ |z| \geq N$
- la parte centrale $vwx$ ha lunghezza limitata $|vwx| \leq N$
- $v$ e $x$ non sono entrambe nulle: $|vx| \geq 1$
- la seconda e la quarta parte possono essere "pompate" quanto si vuole ottenendo sempre altre frasi del linguaggio; ovvero, $uv^iwx^iy \in L,\ \forall i \geq 0$

**Lemma 4.2: pumping lemma per linguaggi di tipo 3**

Se $L$ è un linguaggio di Tipo 3, esiste un intero $N$ (dipendente dallo specifico linguaggio) tale che, per ogni stringa $z$ di lunghezza almeno pari a $N$:

- $z$ può essere riscritta come: $z = xyw,\ |z| \geq N$
- la parte centrale $xy$ ha lunghezza limitata: $|xy| \leq N$
- $y$ non è nulla: $|y| \geq 1$
- la parte centrale può essere "pompata" quanto si vuole ottenendo sempre altre frasi del linguaggio; ovvero, $xy^iw \in L,\ \forall i \geq 0$

**Esempio 4.19**

Il linguaggio $L = \{a^p\}$ con $p$ numero primo non è un linguaggio regolare, infatti se lo fosse esisterebbe un intero $N$ in grado di soddisfare il pumping lemma. Sia $p$ un numero primo tale che $p \geq N + 2$ e consideriamo la stringa $z = a^p$. Possiamo scomporre $z$ nelle sezioni $xyw$ con $|y| = r$, di conseguenza $|xw| = p - r$. In base al lemma se $L$ fosse regolare allora la stringa $xy^{p-r}w$ dovrebbe appartenere al linguaggio, ma la lunghezza di tale stringa sarebbe $|xy^{p-r}w| = (p - r)(1 + r)$, ovvero non un numero primo. Pertanto $L$ non è un linguaggio regolare

**Esempio 4.20**

Il linguaggio $L = \{a^nb^nc^n\}$ non è di tipo 2, se lo fosse esisterebbe un intero $N$ in grado di soddisfare il pumping lemma. Consideriamo allora la stringa $z = a^Nb^Nc^N$, e scomponiamola nelle sue 5 componenti $uvwxy,\ |vwx| \leq N$. Poiché tra l'ultima "$a$" e la prima "$c$" ci sono $N$ posizioni, la parte centrale non può contenere sia "$a$" che "$c$". Quindi o:

- $vwx$ non contiene delle "$c$", e allora $vx$ è fatta solo di "$a$" e "$b$". A questo punto però la stringa composta da $uwy$, che in base al lemma dovrebbe appartenere al linguaggio contiene tutte le "$c$", ma meno "$a$" e "$b$" del necessario, ergo non appartiene al linguaggio (assurdo)
- $vwx$ non contiene delle "$a$", e allora $vx$ è fatta solo di "$b$" e "$c$". A questo punto però la stringa composta da $uwy$, che in base al lemma dovrebbe appartenere al linguaggio contiene tutte le "$a$", ma meno "$b$" e "$c$" del necessario, ergo non appartiene al linguaggio (assurdo)

Entrambe le opzioni conducono ad un assurdo, pertanto $L$ non è un linguaggio context-free.
Prendiamo ad esempio $N = 6$, otteniamo la stringa: $z = aaaaaabbbbbbcccccc$. Scomponiamola quindi nei cinque pezzi $uvwxy$ con $|vwx| \leq N$ e supponiamo di prendere quest'ultima lunga 5. La suddivisione può essere una delle seguenti illustrate in tabella.

| u | vwx | y |
|---|---|---|
| ... | ... | ... |
| aaaaa | abbbb | bbcccccc |
| aaaaaa | bbbbb | bccccccc |
| aaaaaab | bbbbb | ccccccc |
| aaaaaabb | bbbbc | ccccc |
| ... | ... | ... |

Come si può osservare, $vwx$ non può contenere sia $a$ che $c$ per motivi di lunghezza, ergo non appartiene al linguaggio

### 4.5.7 Espressioni regolari

Le espressioni regolari sono dei formalismi usati per descrivere linguaggi regolari.

**Definizione 4.22: Espressioni regolari**

Sono tutte e le sole espressioni ottenibili tramite le regole:

- La stringa vuota è un'espressione regolare
- Dato un alfabeto $A$ ogni elemento di $A$ è un'espressione regolare
- Se $X$ e $Y$ sono espressioni regolari allora lo sono anche:
  - L'unione $X + Y = \{x \mid x \in X \vee x \in Y\}$
  - La concatenazione $X \cdot Y = \{x \mid x = ab,\ a \in X,\ b \in Y\}$
  - La chiusura $X^* = \bigcup_{0}^{N} X^N$, $X^0 = \epsilon$, $X^k = X^{k-1} \cdot X$

**Esempio 4.21**

Dati gli alfabeti:

$$X1 = \{00, 11\}$$
$$X2 = \{01, 10\}$$

Si ottiene:

- $X1 + X2 = \{00, 11, 01, 10\}$
- $X1 \cdot X2 = \{0001, 1101, 0010, 1110\}$
- $X2 \cdot X1 = \{0100, 0111, 1000, 1011\}$
- $X1^* = \{\epsilon, 00, 11, 0000, 0011, 1100, 1111, 000000, 000011, 001100, ...\}$

È importante studiare le espressioni regolari in quanto collegate ai linguaggi regolari. Le grammatiche regolari permettono di descrivere pochi tipi di strutture, ma tali strutture sono tipicamente estremamente comuni nei linguaggi di programmazione, pertanto comprenderle bene aiuta ad ottimizzare l'analisi di questi linguaggi.

**Teorema 4.6**

I linguaggi generati da grammatiche regolari coincidono con i linguaggi descritti da espressioni regolari

Grammatiche e espressioni regolari danno una diversa prospettiva sulla realtà, nello specifico una grammatica offre una rappresentazione *costruttiva*[2] dall'altra parte le espressioni regolari offrono una rappresentazione *descrittiva* [3].

Matematicamente si dimostra che è possibile passare da una rappresentazione all'altra per mezzo di algoritmi e/o regole, tuttavia queste operazioni risultano spesso complesse, specie per linguaggi reali. Pertanto generalmente si preferisce passare per una terza forma e usare quest'ultima come punto di partenza per le conversioni. Si usa effettuare una conversione verso una macchina a stati finiti (che è ottimizzabile, riducibile ai minimi termini, e perfettamente nota) per poi effettuare la conversione finale verso la forma di destinazione.

Per passare **dalla grammatica alle espressioni regolari** è necessario risolvere le cosiddette equazioni sintattiche. In questo genere di equazioni i simboli terminali sono i termini noti, mentre i simboli non terminali sono le incognite da trovare, in particolare, per ottenere un linguaggio completo e non un suo sottoinsieme, occorre risolvere in funzione dello scopo della grammatica.

La risoluzione di queste equazioni si basa su un algoritmo (diverso per le grammatiche regolari a destra o sinistra solamente per quanto riguarda un raccoglimento a fattor comune in cui l'elemento è raccolto a destra nel caso di grammatiche regolari a destra, a sinistra nel caso di grammatiche regolari a sinistra). L'algoritmo sotto riportato riguarda grammatiche regolari a destra.

1. Riscrivere ogni gruppo di produzioni del tipo

   $$X \to \alpha_1 \mid \alpha_2 \mid \ \dots \ \mid \alpha_n$$

   come

   $$X = \alpha_1 + \alpha_2 + \dots + \alpha_n$$

2. Poiché la grammatica è lineare a destra, ogni $\alpha_k$ ha la forma $uX_k$ dove $X_k \in V_N \cup \epsilon$, $u \in V_T^*$. Ergo, si raccolgano a destra i simboli non terminali dei vari $\alpha_1 \dots \alpha_n$ scrivendo

   $$X = (u_1 + u_2 + \dots)X_1 \cup \dots \cup (z_1 + z_2 + \dots)X_n$$

   dove

   $$X_k \in V_N, \ u_k, z_k \in V_T^*$$

   Ciò porta a un sistema di $M$ equazioni in $M$ incognite, dove $M$ è la cardinalità dell'alfabeto $V_N$

3. Eliminare dalle equazioni le ricorsioni dirette, data l'equivalenza

   $$X = uX \cup \delta \equiv X = (u)^*\delta$$

   Ognuna delle forme di frase potrà contenere dei simboli terminali, ma mai $X$

[2] Dice come costruire un linguaggio ma non quale linguaggio si otterrà alla fine del processo
[3] Dice quale sarà la forma del risultato ma non come ottenerla

4. Risolvere il sistema rispetto a $S$ per eliminazioni successive (usando il metodo di Gauss)
5. La soluzione del sistema è il linguaggio regolare cercato.

**Esempio 4.22**

Data la grammatica

$$S \to aB \mid aS$$
$$B \to dS \mid b$$

si può ottenere l'espressione regolare equivalente tramite

1. Scrittura di un'equazione per ogni regola

$$\begin{cases} S = aB + aS \\ B = dS + b \end{cases}$$

2. Eventuali raccoglimenti a fattor comune
3. Eliminazione della ricorsione diretta

$$\begin{cases} S = a^*aB \\ B = dS + b \end{cases}$$

4. Sostituzione della seconda equazione nella prima

$$S = a^*a(dS + b) = a^*adS + a^*ab$$

5. Eliminazione della nuova ricorsione

$$S = (a^*ad)^*a^*ab$$

Data una grammatica è possibile ottenere espressioni regolari diverse. Esse sono, a meno di errori di calcolo, tutte equivalenti, ma all'atto pratico alcune saranno più semplici da riconoscere per una macchina di altre. In generale una macchina preferisce espressioni il cui inizio è noto, in modo da poter procedere deterministicamente nella ricerca di una conferma.

Per passare **dalle espressioni regolari alla grammatica** si interpretano gli operatori dell'espressione in base al loro significato mappandoli in opportune regole della grammatica:

- Sequenza: simboli accostati
- Operatore $+$: simbolo di alternativa (regole distinte)
- Operatore $*$: regola ricorsiva (ciclo)

**Esempio 4.23: dall'espressione regolare alla grammatica**

Data l'espressione regolare

$$L = \{a^*b\}$$

Si ha che il prefisso $a$ può mancare e può essere prodotto da una regola ricorsiva, mentre

il suffisso $b$ è sempre presente. Si può quindi ottenere, ad esempio, la grammatica:

$$S \to Ab \mid b$$
$$A \to a \mid A$$

# Capitolo 5

# Automi riconoscitori

In generale i linguaggi di tipo 0 non sono riconoscibili, ovvero non è garantita l'esistenza di una macchina di Turing in grado di determinare se una frase appartenga o meno al linguaggio. Invece per i linguaggi di tipo 1 (e quindi anche di tipo 2 e 3) è garantita l'esistenza di una macchina di Turing in grado di effettuare questa operazione, tuttavia non è determinabile l'efficienza del processo. Per ottenere un'efficienza accettabile occorre rifarsi sui linguaggi generati da grammatiche di tipo 2 (o da classi speciali di grammatiche di tipo 2), in questo caso il riconoscitore prende il nome di **parser**. Per ottenere un grado ancora maggiore di efficienza, in particolare per sottoparti particolarmente ricorrenti dei linguaggi come gli identificatori, occorre utilizzare linguaggi generati da grammatiche di tipo 3, in questo caso il riconoscitore prende il nome di **scanner** o di **lexer**.

## 5.1 Linguaggi regolari

### 5.1.1 Riconoscitori

Un linguaggio regolare è riconoscibile da un automa a stati finiti (vedi Sezione 2.1). In particolare per questo tipo di linguaggi è sufficiente una specializzazione detta **riconoscitore a stati finiti** (RSF).

**Definizione 5.1: riconoscitore a stati finiti**

Un riconoscitore a stati finiti è una specializzazione di un automa a stati finiti descritto dalla quintupla

$$\langle A, S, S_0, F, sfn^* \rangle$$

dove:

- $A$ è l'alfabeto
- $S$ è l'insieme degli stati
- $S_0$ è lo stato iniziale ($S_0 \in S$)
- $F$ è l'insieme degli stati finali ($F \subseteq S$). Sono anche chiamati stati di accettazione in quanto sono gli stati che corrispondono ad una risposta affermativa da parte dell'automa sottostante
- $sfn^* : A^* \times S \to S$ è la funzione di stato. Essa, a differenza della funzione di stato dell'automa a stati finiti, è in grado di elaborare intere stringhe e non singoli caratteri. In realtà tale funzione è definita in funzione della funzione $sfn$ dell'automa a

stati finiti, ma permette di raggiungere un maggior grado di astrazione durante la fase di progettazione del riconoscitore.

Questa funzione definisce l'evoluzione del riconoscitore a partire dallo stato iniziale $s_0$ in corrispondenza di ogni sequenza di ingresso (cioè stringa) $x \in A^*$.
Si pone:

$$\begin{cases} sfn^*(\epsilon, s) = s \\ sfn^*(xa, s) = sfn(a, sfn^*(x, s)) \end{cases}$$

Ovvero prima viene considerata la sotto stringa $x$ ricorsivamente fino a che non si giunge a considerare la stringa vuota $\epsilon$, e poi tutti i caratteri successivi uno ad uno. Tale definizione sembra al contrario, ma ricorda la definizione di "funzione composta" $f(g(x))$, dove prima si calcola la funzione $g$ e poi la funzione $f$

**Definizione 5.2: frase accettata**

Si dice frase accettata da un RSF una frase $x \in A^*$ che porta il riconoscitore a partire dallo stato iniziale $s_0$ in uno degli stati finali, ovvero

$$sfn^*(x, s_0) \in F$$

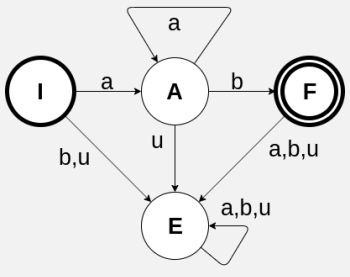
**Definizione 5.3: linguaggio accettato**

Insieme delle frasi accettate dall'automa riconoscitore

**Esempio 5.1**

Siano:

- $A = \{a, b, u\}$
- $S = \{I, A, E, F\}$
- $S_0 = I$
- $F = \{F\}$
- $sfn : A \times S$




Questo automa riconosce come valide tutte, e sole, le frasi della forma "$ab$", "$aab$", "$aaab$" eccetera (in termini di espressioni regolari può essere tradotto come $aa^*b$). Si può notare un auto anello per quanto riguarda la coppia ingresso-stato $\{a, A\}$: Si osserva che ogni qualvolta si ha un anello nel grafo degli stati, questo corrisponde ad una espressione asteriscata nelle espressioni regolari, rendendo quindi semplice la traduzione di una forma nell'altra.

Si noti che in questo primo esempio è stato riportato anche lo stato di errore, tuttavia spesso esso viene omesso per snellire il grafico, intendendo che le sequenze di input specificate sono quelle corrette, mentre quelle non riportate sul grafico conducono tutte nello stato di errore

**Teorema 5.1**

Un linguaggio $L(R)$ è non vuoto se e solo se il riconoscitore $R$ accetta una stringa $x$ di lunghezza $L_x$ minore del numero di stati $N$ dell'automa.

**Dimostrazione**
Se $L(R)$ non è vuoto allora accetterà una stringa $x$. Se $L_x > N$ allora l'automa $R$ dovrà passare più di una volta per almeno uno stesso stato $s_1$. Decomponendo la stringa $x = x_1x_2x_3$ con $x_2 \neq \epsilon$ si deve avere:

$$sfn^*(x_1, s_0) = s_1, \qquad sfn^*(x_2, s_1) = s_1, \qquad sfn^*(x_3, s_1) \in F$$

Ma allora anche la stringa $y = x_1x_3$ deve essere accettata, in quanto l'applicazione della funzione di stato alla stringa $x_2$ riconduce allo stato $s_1$. Si nota poi che $L_y < L_x$. Nel caso in cui $L_y < N$ il teorema è dimostrato, in caso contrario è possibile ripetere il procedimento appena descritto per trovare una stringa $z$ tale che $L_z < N$.
Dimostrare l'opposto è banale, in quanto se $R$ accetta una stringa, di qualsiasi lunghezza essa sia, $L(R)$ è ovviamente non vuoto

**Nota 5.1**

Un modo rapido e semplice per capire se un linguaggio sia vuoto o meno è provare tutte le stringhe possibili aventi lunghezza minore di $N$, nel caso *nessuna di esse* conduca allo stato finale allora il linguaggio è *vuoto*

**Teorema 5.2**

Un linguaggio $L(R)$ è infinito se e solo se il riconoscitore $R$ accetta una stringa $x$ di lunghezza $N \leq L_x < 2N$, con $N$ numero di stati dell'automa

**Dimostrazione**
Sia $L(R)$ infinito. Se $x \in L(R)$ è una stringa avente lunghezza $L_x > N$, ripetendo il ragionamento del teorema 1, essa deve poter essere scomposta come $x = x_1x_2x_3$ con $x_2 \neq \epsilon$ e con:

$$sfn^*(x_1, s_0) = s_1, \qquad sfn^*(x_2, s_1) = s_1, \qquad sfn^*(x_3, s_1) \in F$$

Poiché $sfn^*(x_2, s_1) = s_1$ $R$ accetta in realtà tutte le stringhe della forma $x_1x_2^kx_3,\ k \geq 0$.
Per dimostrare che tra di esse ne esiste certamente una avente lunghezza minore di $2N$ supponiamo per assurdo che $L_x > 2N$, allora in virtù di quanto sopra si deve poter scrivere $x = x_1yx_3$, dove $L_y < N$. Ma anche $x' = x_1x_3$ è una stringa accettata. Se $L_{x'} < N$ allora il teorema è dimostrato in quanto $L_x < N + N = 2N$, altrimenti è possibile iterare il ragionamento al fine di ottenere una stringa $x''$ accettata tale che $L_{x''} < N$.
Dimostrare l'opposto è invece banale. $R$ accetta una stringa di lunghezza $N \leq L_x < 2N$ allora esiste una parte di questa stringa che può essere ripetuta infinite volte, dunque $L(R)$ è un linguaggio infinito

**Nota 5.2**

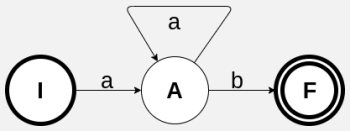
Un modo rapido e semplice per capire se un linguaggio sia infinito o meno è provare tutte le stringhe possibili aventi lunghezza $N \leq L_x < 2N$. Nel caso *nessuna di esse* conduca allo stato finale allora il linguaggio è *finito*, in quanto verrebbe contraddetto il precedente teorema

Come conseguenza di questi due teoremi decidere se un linguaggio regolare sia vuoto o infinito è un problema **risolubile**.

### 5.1.2 Generatori per grammatiche

Per ottenere un generatore per una grammatica di tipo 3 a partire da un riconoscitore è sufficiente sostituire la parola "accetta" con la parola "genera" in tutte le transizioni

**Esempio 5.2**




- Nello stato $I$ l'automa *genera* $a$ e si porta in $A$
- Nello stato $A$ l'automa *genera* $a$ e si riporta in $A$
- Nello stato $A$ l'automa *genera* $b$ e si ferma

Si noti l'assenza dello stato di errore all'interno del grafico. Con questo fatto, come già descritto, si intende che tutte le transizioni non specificate dagli archi del grafo conducono a tale stato

È possibile definire un correlazione forte tra:

- Stati e simboli non terminali
- Transizioni e produzioni
- Scopo della grammatica e uno degli stati

Dunque è possibile automatizzare la costruzione di un RSF a partire da una grammatica oppure risalire ad una grammatica dato un RSF. Per ottenere una grammatica regolare a destra occorre partire dal simbolo iniziale e guardare "dove si va", ovvero cercare quale sia lo stato successivo e quale sia il simbolo terminale che conduce a quello stato. Nel caso di una grammatica regolare a sinistra invece occorre partire dallo stato finale e guardare "da dove si proviene", ovvero cercare quale sia lo stato precedente e quale sia il simbolo terminale che ha permesso di effettuare la transizione.

**Esempio 5.3: continua esempio 5.2**

Nel caso di una grammatica *regolare a destra* si otterrebbe:

$$S \to aA$$
$$A \to aA \mid b$$

Lo scopo della grammatica è lo stato iniziale, mentre lo stato finale non viene considerato in quanto non ha archi uscenti.

Nel caso di una grammatica *regolare a sinistra* si otterrebbe:

$$S \to Ab$$
$$A \to Aa \mid a$$

Lo scopo della grammatica è lo stato finale, mentre lo stato iniziale non viene considerato in quanto non ha archi entranti

Si ottiene dunque il seguente mapping tra RSF e grammatica:

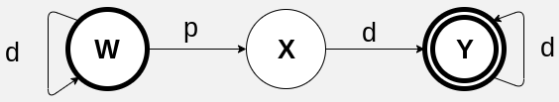
| RSF | Grammatica |
|---|---|
| Stati dell'automa | Meta-simboli |
| Transizioni | Produzioni |
| Uno stato | Scopo della grammatica |

Tabella 5.1: Corrispondenza tra automa e grammatica

Questo mapping presenta alcuni gradi di libertà, infatti se la grammatica è regolare a destra si ottiene un automa riconoscitore *top down*, mentre se la grammatica è regolare a sinistra si ottiene un automa riconoscitore *bottom up*. Viceversa dallo stesso RSF è possibile ottenere una grammatica regolare a destra o sinistra a seconda che lo si interpreti come *top down* o *bottom up*.

Non sempre però risulta semplice ricavare una grammatica mediante approccio bottom up in quanto se è sempre vero che lo stato iniziale è unico, non è possibile dire lo stesso per gli stati finali, anzi spesso è desiderabile avere più stati finali in modo da dare anche un significato semantico alle frasi riconosciuti (es. questa frase è giusta ed ha questo preciso significato, quella frase è giusta ed ha quel significato). Altri problemi possono presentarsi nei casi in cui si effettui un'analisi top down e lo stato iniziale abbia archi entranti, oppure nel caso in cui si effettui un'analisi bottom up e gli stati finali abbiano degli archi uscenti.

**Esempio 5.4**




Ripetendo il ragionamento prima espresso si può ottenere il seguente generatore:

- nello stato $W$ l'automa genera $d$ e si riporta nello stato $W$ (per poi proseguire)
- nello stato $W$ l'automa genera $p$ e si riporta nello stato $X$ (per poi proseguire)
- nello stato $X$ l'automa genera $d$ e si ferma
- nello stato $X$ l'automa genera $d$ e si riporta nello stato $Y$ (per poi proseguire)
- nello stato $Y$ l'automa genera $d$ e si ferma
- nello stato $Y$ l'automa genera $d$ e si riporta nello stato $Y$ (per poi proseguire)

A partire da queste frasi è possibile ottenere la grammatica regolare a destra:

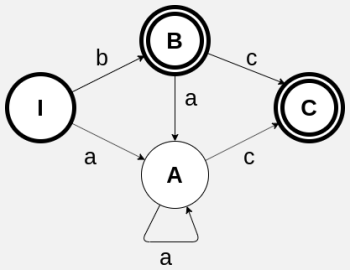
$$W \to dW \mid pX$$
$$X \to d \mid dY$$
$$Y \to d \mid dY$$

e la grammatica regolare sinistra:

$$Y \to Xd \mid Yd$$
$$X \to p \mid Wp$$
$$W \to d \mid Wd$$

Si noti come in questo esempio sia lo stato iniziale che lo stato finale siano anche stati di transito

**Esempio 5.5: più stati finali**




Sia $B$ che $C$ sono stati finali. A questo punto è possibile considerare separatamente le grammatiche che hanno $B$ come stato finale da quelle che hanno $C$ come stato finale. La grammatica finale sarà l'unione delle due.

**Mediante analisi top down**
Considerando la grammatica avente $B$ come stato finale si ha:

- $S_1 \to b$

Considerando invece la grammatica avente $C$ come stato finale invece:

- $S_2 \to bB \mid aA$
- $B \to cC \mid aA$
- $A \to aA \mid cC$

Di conseguenza la grammatica sarà l'unione dei due insiemi di regole $S \to S_1 \mid S_2$

- $S \to bB \mid aA \mid b$
- $B \to cC \mid aA$
- $A \to aA \mid cC$

**Mediante analisi bottom up**
In questo caso occorre separare i due stati finali. Considerando $C$ come scopo della grammatica si ottiene:

- $S_1 \to Bc \mid Ac$
- $B \to Ib$
- $A \to Ia \mid Aa \mid Ba$

Considerando invece $B$ come scopo della grammatica:

- $S_2 \to b$

La grammatica complessiva si ottiene mediante l'unione delle due sotto grammatiche $S \to S_1 \mid S_2$:

- $S \to Bc \mid Ac \mid b$
- $B \to b$
- $A \to a \mid Aa \mid Ba$

Occorre infine fare attenzione al momento in cui queste grammatiche unite vengono rimappate su di un automa, infatti è possibile che esso risulti *non deterministico*

Più in generale nel caso in cui si voglia effettuare un'analisi bottom up ed il grafo degli stati presenti più stati finali è possibile seguire il seguente algoritmo (**automatizzabile**):

1. Per ogni stato finale assumere come scopo della grammatica quel singolo stato $S_k$ ed esprimere le regole relative a quello scopo
2. Esprimere il linguaggio complessivo come unione dei singoli linguaggi $S \to S_1 \mid \ldots \mid S_n$

### 5.1.3 Implementazione di automi deterministici

Questo tipo di RSF è facilmente realizzabile all'interno di un linguaggio imperativo, tanto che vi sono più alternative per l'implementazione (es. ciclo while con if annidati, ciclo while con degli switch, ciclo while con tabella separata, ...). Le prime due soluzioni sono pessime, mentre la terza è migliore in quanto è facilmente leggibile, testabile ed estendibile.

### 5.1.4 Implementazione di automi non deterministici

Certe grammatiche possono però portare alla realizzazione di un automa non deterministico (vedi Figura 5.1), ovvero un automa per cui almeno una delle celle della tabella delle transizioni contiene più stati futuri. Nell'esempio contenuto nell'immagine in corrispondenza dallo stato $B$ con ingresso $a$ l'automa non è in grado di decidere quale sia lo stato futuro corretto, ovvero se $B$ oppure $F$.

Un riconoscitore non deterministico deve essere in grado di *scegliere* quale sia lo stato futuro corretto, in quanto una frase riconosciuta come corretta è certamente corretta, mentre una frase riconosciuta come errata diventa realmente errata solo nel caso in cui siano state esplorate tutte le possibili diramazioni dell'albero di esplorazione (nel caso di un linguaggio infinito sono infinite) e nessuna di esse risulti corretta. Se il linguaggio usato per la realizzazione dell'automa è in grado di supportare il *non determinismo* (es. Prolog) l'implementazione diventa relativamente semplice, in caso contrario (es. linguaggio imperativo) diventa necessario implementare meccanismi di backtracking.

In generale un riconoscitore *non deterministico* risulta meno efficiente di uno *deterministico* in quanto è costretto a mantenere traccia del percorso fatto durante l'esplorazione dell'albero mediante varie strutture dati in modo da essere in grado di ritornare sulle proprie decisioni esplorando un ramo differente. Tuttavia per linguaggi di **tipo 3** è sempre possibile ricondurre un automa non deterministico ad uno deterministico, eventualmente aggiungendo degli stati (in ogni caso è sempre possibile ottenere la forma minima di un automa mediante un procedimento meccanico).

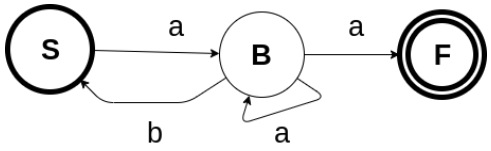


Figura 5.1: Automa non deterministico

**Teorema 5.3**

Un automa non deterministico può sempre essere ricondotto a un automa deterministico equivalente

**Esempio 5.6**

Riprendendo l'automa in figura 5.1 si può ottenere la seguente tabella delle transizioni:

| | $a$ | $b$ |
|---|---|---|
| $S$ | $B$ | $E$ |
| $B$ | $B/F$ | $S$ |
| $F$ | $E$ | $E$ |
| $E$ | $E$ | $E$ |

Procedendo ad accorpare gli stati, in modo da ritardare il più possibile la decisione riguardante quale strada intraprendere, si ottiene la seguente tabella delle transizioni:

| | $a$ | $b$ |
|---|---|---|
| $[S]$ | $[B]$ | $[E]$ |
| $[B]$ | $[B,F]$ | $[S]$ |
| $[B,F]$ | $[B,F,E]$ | $[S,E]$ |
| $[S,E]$ | $[B,E]$ | $[E]$ |
| $[B,E]$ | $[B,F,E]$ | $[S,E]$ |
| $[B,F,E]$ | $[B,F,E]$ | $[S,E]$ |
| $[E]$ | $[E]$ | $[E]$ |

A partire da questa seconda tabella è poi possibile ottenere un nuovo automa, minimo, che non presenta più il non determinismo. Da esso è poi possibile ricavare una nuova grammatica anch'essa non deterministica

Se invece il linguaggio supporta il non determinismo implementare il riconoscitore è semplice.

**Esempio 5.7: non determinismo in Prolog**

L'automa precedente identifica la grammatica

$$S \to aB$$
$$B \to aB \mid bS \mid a$$

In Prolog possiamo mappare ogni produzione su una regola. Il motore del linguaggio è in grado di tentare una strada e tornare indietro nel caso essa si riveli sbagliata, provando

via via tutte le possibili alternative. Il programma Prolog può essere quindi strutturato come segue:

```
regolaS([a|B]) :- regolaB(B).
regolaB([a|B]) :- regolaB(B).
regolaB([b|S]) :- regolaS(S).
regolaB([a]).
```

Ora basta interrogare il sistema Prolog con alcune sequenze di caratteri per vedere se appartengono o meno al linguaggio generato dalla grammatica:

```
?- regolaS([a,a,b,a]).
> no

?- regolaS([a,a,b,a,a]).
> yes
```

Come detto, lo svantaggio dell'approccio puramente dichiarativo è l'inefficienza, infatti è necessario disporre di opportune strutture interne per "ricordare" il percorso fatto e per poter "tornare indietro" nel caso in cui questa operazione si renda necessaria. Per questo motivo può essere conveniente prima ricondursi ad un automa *deterministico* e poi implementarlo rapidamente in Prolog, ricordandosi di usare il predicato `!` (*cut*) che indica al motore Prolog di *tagliare* le alternative poiché non saranno necessarie trattandosi di un automa deterministico. Ad esempio tramite regole del tipo:

```
s1([a|S2]) :- !, s2(S2).
```

### 5.1.5 Conclusioni

**Teorema 5.4**

L'insieme dei linguaggi riconosciuti da un ASF coincide con l'insieme dei linguaggi regolari, ossia quelli descritti da espressioni regolari

Di conseguenza espressioni regolari e automi a stati finiti sono metodi descrittivi appartenenti a due differenti categorie che però portano allo stesso risultato. In particolare si osserva che è possibile effettuare le seguenti conversioni:

- Per passare da una grammatica alle corrispondenti espressioni regolari occorre risolvere le equazioni sintattiche
- Per passare dalle equazioni sintattiche alla corrispondente grammatica occorre mappare le singole espressioni sulle produzioni della grammatica
- Per passare dalla grammatica all'automa corrispondente e viceversa occorre utilizzare il mapping sopra descritto
- Per passare dalle espressioni regolari all'automa corrispondente e viceversa occorre utilizzare il mapping sopra descritto

Si hanno quindi a disposizione tre formalismi che forniscono tre punti di vista diversi della stessa realtà:

- **Grammatica**: formalismo *costruttivo* che dice **come si fa** ma **non cosa si ottiene**
- **Espressione regolare**: formalismo *descrittivo* che dice **cosa si ottiene** ma **non come si fa**

- **Automa**: formalismo *operazionale* che dice **come si realizza** ma **non cosa si ottiene**

## 5.2 Linguaggi context-free

Come anticipato un RSF non è in grado di riconoscere un linguaggio di tipo 2 poiché è strutturalmente limitato dalla propria capacità di memorizzazione (proporzionale al numero di stati). Di conseguenza un RSF non è in grado di riconoscere sezioni la cui lunghezza massima non è nota a priori (es. bilanciamento delle parentesi).

**Esempio 5.8: frasi palindrome**

Data la grammatica

$$S \to 0S0 \mid 1S1 \mid c$$

Essa genera il linguaggio:

$$L = \{word\ c\ word^R\}$$

dove $word^R$ indica la prima parola al contrario. Tale linguaggio non è riconoscibile da un RSF in quanto esso dovrebbe conoscere la lunghezza della stringa generata che però non è limitata a priori

Per riconoscere le grammatiche di tipo 2 occorre un diverso sistema che sia in grado di superare i limiti dati da una memoria dimensionata a priori. Una soluzione consiste nell'appoggiarsi ad una struttura a **stack**. L'automa ottenuto è definito **push down automaton**, sostanzialmente una macchina a stati finiti dotata di una memoria esterna (astrattamente un nastro monodirezionale di lunghezza non nota a priori ed estendibile).

**Definizione 5.4: push down automaton**

Formalmente un PDA è definito dalla sestupla

$$\langle A, S, S_0, sfn, Z, Z_0 \rangle$$

dove:

- $A$ è l'alfabeto
- $S$ è l'insieme degli stati
- $S_0 \in S$ è lo stato iniziale
- $sfn : (A \cup \epsilon) \times S \times Z \to W \subseteq S \times Z^*$ è la funzione di stato
- $Z$ è l'alfabeto dei simboli interni
- $Z_0$ è il simbolo iniziale che si trova sullo stack

Si noti che questa definizione include le $\epsilon$-mosse (mosse spontanee che manipolano lo stack senza leggere da input), ma che per rimuoverle è sufficiente rimuovere la stringa vuota dalla definizione di funzione di stato. Questa costruzione introduce un nuovo grado di libertà, infatti è possibile decidere se utilizzare maggiormente gli stati dell'ASF interno o maggiormente lo stack (idealmente richiedendo soltanto due stati per funzionare quindi).

La funzione di stato opera consumando uno dei simboli di ingresso ed effettuando una *pop* dallo stack in modo da ottenere il valore attuale. Sulla base di questi valori e dello stato attuale porta l'automa in un nuovo stato interno scrivendo eventualmente sullo stack nuovi simboli. Inoltre la definizione di linguaggio accettato precedentemente data (vedi Definizione 5.3) può essere ampliata in modo da includere la possibilità di utilizzare lo stack della macchina.

**Definizione 5.5: linguaggio accettato da un PDA**

Il linguaggio accettato da un PDA è definibile in due modi:

- **Criterio dello stato finale**: il linguaggio accettato è l'insieme di tutte le stringhe di ingresso che portano il PDA in uno degli stati finali
- **Criterio dello stack vuoto**: il linguaggio accettato è l'insieme di tutte le stringhe che portano il PDA nella configurazione di stack vuoto

**Esempio 5.9: continua esempio 5.8**

Riprendendo l'esempio precedente sia data la seguente grammatica:

$$S \to 0S0 \mid 1S1 \mid c$$

da essa è possibile ottenere il seguente PDA:

| $A \cup \epsilon$ | $S$ | $Z$ | $S \times Z^*$ |
|---|---|---|---|
| 0 | $Q1$ | $Centro$ | $Q1 \times CentroZero$ |
| 1 | $Q1$ | $Centro$ | $Q1 \times CentroUno$ |
| $c$ | $Q1$ | $Centro$ | $Q2 \times Centro$ |
| 0 | $Q1$ | $Zero$ | $Q1 \times ZeroZero$ |
| 1 | $Q1$ | $Zero$ | $Q1 \times ZeroUno$ |
| $c$ | $Q1$ | $Zero$ | $Q2 \times Zero$ |
| 0 | $Q1$ | $Uno$ | $Q1 \times UnoZero$ |
| 1 | $Q1$ | $Uno$ | $Q1 \times UnoUno$ |
| $c$ | $Q1$ | $Uno$ | $Q2 \times Uno$ |
| 0 | $Q2$ | $Zero$ | $Q2 \times \epsilon$ |
| 1 | $Q2$ | $Uno$ | $Q2 \times \epsilon$ |
| $\epsilon$ | $Q2$ | $Centro$ | $Q2 \times \epsilon$ |

$Q1$ risulta lo stato di "accumulo", in cui il PDA legge simboli prima di raggiungere il simbolo centrale, $Q2$ invece è lo stato di "svuotamento" in cui il PDA controlla che i simboli in ingresso corrispondano a quelli sullo stack

**non deterministici** Così come un RSF anche un PDA può essere **non deterministico**. In questo caso la funzione $sfn$ produce un insieme di elementi appartenenti a $W \subseteq S \times Z^*$, ovvero l'automa a partire da uno stato e con uno specifico ingresso può potenzialmente transitare in più stati futuri.

Il non determinismo dell'automa può emergere in due casi distinti:

- La funzione di stato non produce un singolo valore nel caso generale, ma un insieme di valori
- A partire dallo stato $Q_i$ l'automa ha la libertà di scegliere se leggere o meno l'ingresso attuale, questo accade se sono definite le $\epsilon$-mosse

**Teorema 5.5**

La classe dei linguaggi riconosciuti da un PDA non deterministico coincide con la classe dei linguaggi context-free, perciò qualunque linguaggio context-free può sempre essere riconosciuto da un opportuno PDA non deterministico

Tuttavia la complessità computazionale di un PDA non deterministico può anche essere esponenziale (usando algoritmi non ottimizzati). Si dimostra che con i migliori algoritmi essa scende fino ad una complessità cubica (o quadratica nel caso di grammatiche non ambigue). Questi valori sono tuttavia lontani dal caso ideale, ovvero la complessità lineare.

In generale non è possibile rinunciare a dei riconoscitori non deterministici in quanto vi sono grammatiche riconoscibili soltanto da questo tipo di PDA, tuttavia per molti casi di interesse pratico esistono linguaggi di tipo 2 riconoscibili da un PDA **deterministico**, i quali riconoscono le stringhe con complessità **lineare**.

**PDA deterministici** Per ottenere un PDA deterministico non deve accadere che l'automa possa portarsi in più stati futuri né optare se leggere o meno il simbolo di ingresso attuale. Questo impone delle limitazioni, ma permette un'efficienza di gran lunga maggiore durante la computazione, fattore maggiormente desiderabile rispetto alla completa libertà.

Dati due linguaggi deterministici tuttavia non è detto che una loro combinazione lo sia, in particolare le operazioni di unione, intersezione e concatenamento non è detto che diano come risultato un linguaggio deterministico, mentre l'operazione di complemento di un linguaggio deterministico restituirà sempre un altro linguaggio deterministico. Si dimostra inoltre che dato un linguaggio deterministico $L$ ed un linguaggio regolare $R$:

- il linguaggio quoziente $L \setminus R$ (l'insieme delle stringhe appartenenti ad $L$ private dei suffissi regolari) è deterministico
- il concatenamento $LR$ (l'insieme delle stringhe di $L$ con suffisso regolare) è deterministico

**Esempio 5.10: unione di linguaggi deterministici**

Dati i linguaggi deterministici

$$L_1 = \{a^n b^n,\ n > 0\}$$
$$L_2 = \{a^n b^{2n},\ n > 0\}$$

la loro unione $L_3 = L_1 \cup L_2$ non è deterministica. Per comprenderlo è sufficiente considerare come dovrebbe operare un riconoscitore. Per prima cosa dovrebbe leggere tutte le $a$ che trova. Successivamente dovrebbe estrarre una $a$ ogni $b$ nel caso si tratti di una stringa appartenente ad $L_1$, mentre dovrebbe estrarre una $a$ ogni due $b$ nel caso si tratti di una stringa appartenente ad $L_2$. Di conseguenza questo linguaggio necessita di un automa non deterministico

**Esempio 5.11: intersezione di linguaggi deterministici**

Dati i linguaggi deterministici

$$L_1 = \{a^n b^n c^*,\ n > 0\}$$
$$L_2 = \{a^* b^n c^n,\ n > 0\}$$

il linguaggio intersezione $L_3 = L_1 \cap L_2 = \{a^n b^n c^n,\ n > 0\}$ non è nemmeno di tipo 2 (vedi Esempio 4.20), di conseguenza non è riconoscibile da un PDA deterministico

Passando da un PDA non deterministico ad uno deterministico vengono meno alcune proprietà:

- Il criterio di terminazione dello stack vuoto risulta meno potente del criterio riguardante gli stati finali
- Una limitazione sul numero di stati interni o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- L'assenza delle $\epsilon$-mosse riduce l'insieme dei linguaggi riconoscibili

Nonostante queste limitazioni è comunque possibile realizzare linguaggi complessi ed espressivi.

### 5.2.1 Riconoscitori $LL$

Vi sono più alternative per la realizzazione di tali PDA deterministici. È possibile utilizzare la definizione di PDA pilotando a mano lo stack, tuttavia questa soluzione risulta poco agevole. Un'altra soluzione consiste nell'utilizzare i meccanismi già presenti in vari linguaggi di programmazione al fine di ottenere lo stack realizzato automaticamente, ad esempio sfruttando lo stack delle chiamate utilizzando opportunamente le chiamate a funzione.

**Definizione 5.6: analisi ricorsiva discendente**

Questa tecnica consiste nella creazione di una funzione per ogni meta-simbolo della grammatica che viene chiamata ogni volta che si incontra quel particolare meta simbolo. La chiamata causa l'inserimento di un record di attivazione all'interno dello stack delle chiamate e si occupa di gestire *tutte* le regole relative a quel simbolo. Ognuna delle funzioni può dunque terminare correttamente, nel caso in cui il ramo con origine in quel simbolo sia corretto, oppure con un errore, nel caso in cui da qualche parte nel sottoramo vi sia stato un errore (eventualmente è possibile specificare dove).
Ogniqualvolta l'analisi incontra un simbolo terminale nella grammatica viene effettuata una lettura da input, ogni volta che viene incontrato un metacarattere viene effettuata una chiamata a funzione

A questo punto diventa semplice realizzare un qualsiasi PDA, tuttavia il risultato conterrà cablata nel codice anche la grammatica specifica per cui era stato costruito quel PDA. Può risultare utile quindi separare il motore (invariante rispetto alle grammatiche) dalle regole della grammatica. A questo scopo è possibile costruire una tabella di parsing (vedi Esempio 5.12), simile alla tabella della transizioni di un RSF ma contenente le indicazioni circa la prossima produzione da applicare. In questo modo il motore (generale) dovrà semplicemente consultare la tabella di parsing (potenzialmente sempre diversa).

**Esempio 5.12: tabella di parsing**

Riprendendo la grammatica vista nell'esempio 5.8 si ottiene la seguente tabella di parsing

| | 0 | 1 | $c$ |
|---|---|---|---|
| $S$ | $S \to 0S0$ | $S \to 1S1$ | $S \to c$ |

Questo tipo di analisi ha sia dei vantaggi che dei limiti. Da un lato rende immediato scrivere il riconoscitore a partire dalla grammatica, utilizzando un mapping semplice che riduce la possibilità di errori e facilitando l'inserzione di azioni anche nelle fasi finali del progetto, dall'altro invece essa non è sempre applicabile, in quanto questo approccio ha l'esigenza di una

grammatica *deterministica*. Di conseguenza occorre determinare quale sia il sottoinsieme delle grammatiche di tipo 2 che presentano determinismo.

Per poter effettuare un'analisi deterministica occorre essere in possesso di un quantitativo di informazioni sufficiente a non dover mai andare a caso, queste riguardano sia il passato, ovvero i simboli precedentemente incontrati, sia il futuro prossimo, ovvero i $k$ simboli che verranno prossimamente incontrati.

**Definizione 5.7: grammatiche $LL(k)$**

Si definiscono grammatiche $LL(k)$ le grammatiche tali da poter essere analizzate deterministicamente:

- Procedendo da sinistra a destra
- Applicando la derivazione canonica sinistra
- Guardando al più $k$ simboli in avanti

Ovvero sono quelle grammatiche che consentono di scegliere con certezza quale produzione scegliere guardando avanti al più $k$ simboli

Rivestono particolare interesse le grammatiche del tipo $LL(1)$, in quanto consentono di operare in modo deterministico anche conoscendo solamente un simbolo futuro.

Una volta finita l'analisi occorre decidere quale sia il ruolo della funzione di top-level: ossia occorre decidere se la frase identificata debba essere completa, ovvero non essere seguita da alcun carattere, oppure se possa essere soltanto un risultato parziale, nel qual caso qualcun altro potrà eseguire altre fasi di analisi.

Generalmente conviene mantenere le varie funzioni assolutamente generali, rimandando la decisione circa la completezza dell'input ad un secondo momento; in questo modo il codice scritto risulta maggiormente riutilizzabile e modulare. Se è necessario imporre che la frase sia completa la grammatica deve essere completata con una produzione di top-level che lo affermi (es. $Z \to S\$$, dove \$ è un simbolo indicante il generico fine linea).

### Generalizzazioni

Spesso le parti destre delle produzioni di uno stesso meta-simbolo non iniziano tutte con un simbolo terminale, quindi non è immediato comprendere quali siano gli input ammissibili ed occorre ragionare i possibili casi futuri per ritardare il più possibile la decisione circa quale strada intraprendere. Si introduce quindi il concetto di **starter symbol set**.

**Definizione 5.8: starter symbol set**

Lo starter symbol set della riscrittura $\alpha$ ($SS(\alpha)$) è l'insieme definito da:

$$SS(\alpha) = \{a \in V_T \mid \alpha \overset{*}{\Rightarrow} a\beta\} \qquad \alpha \in V^+,\ \beta \in V^*$$

Ovvero sono le iniziali di una forma di frase $\alpha$ ricavate applicando anche più produzioni. Si noti che l'operatore $^*$ cattura il caso in cui non venga applicata alcuna produzione, ovvero il caso in cui il simbolo iniziale sia già un simbolo terminale ($\alpha \in V_T$).
In particolare se la stringa $\alpha$ è costituita da un singolo meta-simbolo $A$ (come nelle grammatiche di tipo 2) si ottiene la seguente definizione semplificata:

$$SS(A) = \{a \in V_T \mid A \overset{+}{\Rightarrow} a\beta\} \qquad A \in V_N,\ \beta \in V^*$$

La definizione appena fornita non include però le produzioni che forniscono una stringa vuota (come $\alpha \to \epsilon$), in quanto la loro derivazione non può produrre alcuna forma del tipo $a\beta$. A questo proposito si introduce l'insieme $FIRST$, analogo ai $SS$, ma con la capacità di includere anche le stringhe vuote.

**Definizione 5.9: insieme $FIRST$**

L'insieme $FIRST$ è definito come

$$FIRST(\alpha) = trunc_1(\{x \in V_T^* \mid \alpha \overset{*}{\Rightarrow} x\}), \qquad \alpha \in V^*$$

dove la funzione $trunc_1$ denota il troncamento della stringa al primo elemento

Queste definizioni consentono di ottenere una generalizzazione rispetto alle condizioni necessarie per ottenere grammatiche di tipo $LL(1)$.

**Condizione 5.1**

Condizione necessaria affinché una grammatica sia $LL(1)$ è che gli starter symbol relativi alle parti destre di uno stesso meta-simbolo siano disgiunti

In realtà, se nessun meta-simbolo genera la stringa vuota, tale condizione è anche **sufficiente**.

**Esempio 5.13**

Sia data la grammatica:

$$S \to AB \mid B$$
$$A \to aA \mid d$$
$$B \to bB \mid c$$

Per comprendere se tale grammatica sia $LL(1)$ o meno occorre controllare gli starter symbol set delle varie produzioni:

- Per la produzione $A$ si hanno starter symbol set disgiunti

$$SS(aA) = \{a\} \qquad SS(d) = \{d\}$$

- Per la produzione $B$ si hanno starter symbol set disgiunti

$$SS(bB) = \{b\} \qquad SS(c) = \{c\}$$

- Per la produzione $S$ si hanno starter symbol set disgiunti

$$SS(AB) = SS(A) = \{a, d\} \qquad SS(B) = SS(B) = \{b, c\}$$

Dato che nessuna produzione genera la stringa vuota gli $SS$ di ogni riscrittura coincidono con quelli del relativo meta-simbolo iniziale. Poiché tali insiemi sono disgiunti è verificata la condizione necessaria, inoltre poiché nessuna produzione genera la stringa vuota è anche verificata la condizione sufficiente e dunque la grammatica è $LL(1)$

**Esempio 5.14**

Sia data la grammatica:

$$S \to AB \mid B$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid c$$

Per comprendere se tale grammatica sia $LL(1)$ o meno occorre controllare gli starter symbol set delle varie produzioni:

- Per la produzione $A$ si hanno starter symbol set disgiunti

$$SS(aA) = \{a\} \qquad SS(\epsilon) = \emptyset$$

- Per la produzione $B$ si hanno starter symbol set disgiunti

$$SS(bB) = \{b\} \qquad SS(c) = \{c\}$$

- Per la produzione $S$ **non** si hanno starter symbol set disgiunti

$$SS(AB) = SS(A) \cup SS(B) = \{a, b, c\} \qquad SS(B) = \{b, c\}$$

  Si ha questo comportamento perché nel caso in cui la produzione $A$ generi la stringa vuota il risultato della produzione $S$ inizierà con il meta-simbolo $B$ e quindi avrà come simboli iniziali quelli della produzione $B$, rendendo di fatto indistinguibili le due produzioni destre in alcuni casi

La condizione necessaria non è soddisfatta in quanto $SS(AB)$ e $SS(B)$ (relativamente alla prima produzione) non sono disgiunti, dunque la grammatica non è $LL(1)$

**Nota 5.3**

Si nota che nel caso una grammatica non sia $LL(1)$ la sua tabella di parsing contiene più regole in almeno una delle celle

Dal secondo esempio si nota la necessità di trovare una soluzione al problema posto dalle $\epsilon$-regole. Si possono avere, in termini generali, due soluzioni: o si elimina per sostituzione la stringa vuota ottenendo quindi una grammatica equivalente oppure si amplia la nozione di starter symbol set. In generale il primo metodo può portare a dei risultati in alcuni casi semplici, ma non sempre. Inoltre spesso le produzioni che generano la stringa vuota sono utili per esprimere il concetto di *parte opzionale*, pertanto rimuovere queste regole, sebbene non alteri il significato della grammatica, la rende meno comprensibile ad un essere umano.

Senza dover eliminare a priori le produzione che generano la stringa vuota vi sono due possibilità:

- Agire sulla parsing table, formalizzando il concetto di blocco annullabile ed integrando nella tabella anche le informazioni relative alle stringhe che possono scomparire. Un blocco annullabile è una stringa che può degenerare in una stringa vuota. Di fatto in questi casi anche un meta-simbolo che non sembrava iniziale può diventarlo scegliendo la giusta sequenza di derivazione, di conseguenza per tener conto delle regole concernenti la

stringa vuota occorre considerare i simboli che possono **seguire** quelli annullabili, ovvero andando a costruire l'insieme $FOLLOW$

- Ampliare la nozione di Starter Symbol Set verso la nozione di **Director Symbol Set** (chiamato anche Look-Ahead set), il quale integra gli effetti delle stringhe vuote a priori. Si parte dalla nozione che in presenza di una produzione che genera la stringa vuota lo starter symbol set dà un'informazione incompleta e potenzialmente fuorviante. Di conseguenza occorre tener conto anche dei simboli che potrebbero seguire nel caso in cui si presenti una stringa vuota tramite l'integrazione dell'insieme $FOLLOW$

**Definizione 5.10: Director Symbol set**

Definiamo director symbol set della produzione $A \to \alpha$ come

$$DS(A \to \alpha) = \begin{cases} SS(\alpha) & \text{se } \alpha \text{ non genera } \epsilon \\ SS(\alpha) \cup FOLLOW(A) & \text{se } \alpha \text{ può generare } \epsilon \end{cases}$$

dove l'insieme $FOLLOW$ contiene i simboli che possono seguire la frase generata da $A$

$$FOLLOW(A) = \{a \in V_T \mid S \overset{*}{\Rightarrow} \gamma A a \beta\} \qquad \gamma, \beta \in V^*$$

La definizione del director symbol set permette di riformulare la condizione $LL(1)$:

**Condizione 5.2**

Condizione necessaria e sufficiente perché una grammatica sia $LL(1)$ è che i Director Symbol set relativi a produzioni alternative siano disgiunti

**Esempio 5.15**

Data la grammatica

$$\begin{aligned} &S \to AB \\ &A \to PQ \mid BC \\ &P \to pP \mid \epsilon \\ &Q \to qQ \mid \epsilon \\ &B \to bB \mid d \\ &C \to cC \mid f \end{aligned}$$

Come si può notare la produzione $A \to PQ$ potrebbe generare la stringa vuota, rendendo il metacarattere $B$ della prima produzione effettivamente il primo della stringa. Calcolando i director symbol set si nota che essi non sono disgiunti per le produzioni relativamente al metacarattere $A$:

$$\begin{aligned} &DS(A \to PQ) = SS(PQ) \cup FOLLOW(A) = \{p, q, b, d\} \\ &DS(A \to BC) = SS(BC) = \{b, d\} \end{aligned}$$

I due insiemi non sono disgiunti pertanto la grammatica data non è $LL(1)$

#### Il problema della ricorsione sinistra

La ricorsione sinistra, per sua definizione, non è compatibile con la definizione di grammatica $LL(1)$, in quanto le produzioni della forma $A \to A\alpha \mid a$ per definizione danno sempre starter symbol set identici per le due alternative. In teoria questo non rappresenta un problema, in quanto una ricorsione sinistra è sempre convertibile in una ricorsione destra, tuttavia in pratica non sempre questo è possibile. Infatti cambiare il verso della ricorsione spesso significa cambiare l'albero di derivazione e, di conseguenza, la semantica della frase analizzata[1].

Di conseguenza se la grammatica non è $LL(1)$ è possibile riorganizzarla mediante sostituzioni e raccoglimenti oppure modificandola direttamente.

**Esempio 5.16: raccoglimento ok**

La seguente grammatica non è sicuramente $LL(1)$

$$S \to aSb \mid aSc \mid \epsilon$$

Tuttavia è possibile raccogliere il prefisso $aS$ comune alle prime due produzioni e introdurre un nuovo meta-simbolo $X$ per esprimere la parte che segue il prefisso comune.

$$S \to aSX \mid \epsilon$$
$$X \to b \mid c$$

In questo caso la grammatica ottenuta è $LL(1)$, ma non sempre questo procedimento è applicabile. Alcune volte infatti il raccoglimento può risultare inutile e portare ad un ciclo infinito di riscritture

Un'alternativa è costituita dall'uso di una analisi $LL(k)$, con $k$ grande a sufficienza per rendere l'analisi deterministica. Tuttavia questo potrebbe non essere sufficiente in quanto il problema concernente il determinare se un linguaggio $L$ sia $LL(1)$ è **indecidibile**, mentre il problema, più semplice, concernente il determinare se una grammatica sia $LL(1)$ è computabile.

Esistono, fortunatamente, analisi molto più potenti dell'analisi $LL$, come l'analisi $LR$. Vi sono infatti linguaggi di tipo 2 deterministici non analizzabili in modo deterministico con tecniche $LL$, ma riconoscibili in modo deterministico con tecniche $LR$.

### 5.2.2 Riconoscitori $LR$

La debolezza dell'analisi $LL$ sta nella sua semplicità, infatti per costruire l'albero in maniera top down il riconoscitore deve essere in grado di identificare la produzione corretta solamente mediante i primi $k$ simboli della parte destra. In pratica $LL(1)$ è quasi l'unico caso interessante, $LL(2)$ torna utile in certe particolari situazioni, mentre $LL(0)$ è praticamente inutile in quanto non permette alcun grado di libertà nella scelta. L'analisi $LR$ invece si pone come obiettivo la costruzione dell'albero a partire dalle foglie, ovvero in maniera bottom up. Un riconoscitore di questo tipo attende di avere sufficienti informazioni per poter costruire a colpo sicuro un nodo padre per alcuni dei nodi foglia di cui è a conoscenza, di conseguenza il suo comportamento è meno naturale rispetto a quello di un riconoscitore $LL$, ma risulta molto più potente. Infatti si dimostra che un riconoscitore $LR$ è lo strumento più potente a disposizione per poter riconoscere delle grammatiche di tipo 2, e che ogni grammatica $LL(k)$ è anche $LR(k)$ (ma non viceversa).

Data la complessità dell'analisi $LR$ sono state sviluppate tecniche semplificate atte a ridurne la complessità, ma comunque si fa uso di strumenti automatici per poter gestire la complessità in maniera adeguata.

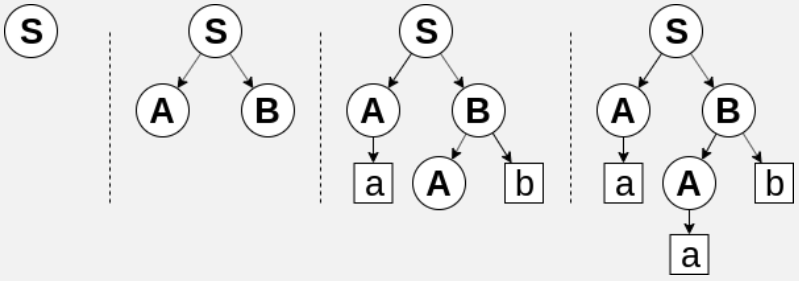
[1] Ad esempio l'espressione $8 - 2 - 5$ cambia significato se analizzata con associatività sinistra o destra degli operatori, gli operatori matematici sono generalmente associativi a sinistra

### Tecniche $LR$

Se l'analisi $LL$ parte dallo scopo della grammatica per poi scendere cercando di coprire l'intera frase, l'analisi $LR$ invece parte dalla frase per cercare di ridurla allo scopo della grammatica. A questo scopo ad ogni passo è possibile decidere se proseguire leggendo l'input senza fare altre operazioni (**SHIFT**) oppure se costruire un pezzo dell'albero senza leggere da input (**REDUCE**) sulla base di *informazioni di contesto*. Concettualmente ogni parser $LR$ ha la necessità di dotarsi di un "oracolo" che ne guidi le decisioni sulla base del **contesto corrente**, di uno stack che contenga i dati necessari e di un controllore che governi le azioni da eseguire.

**Esempio 5.17**

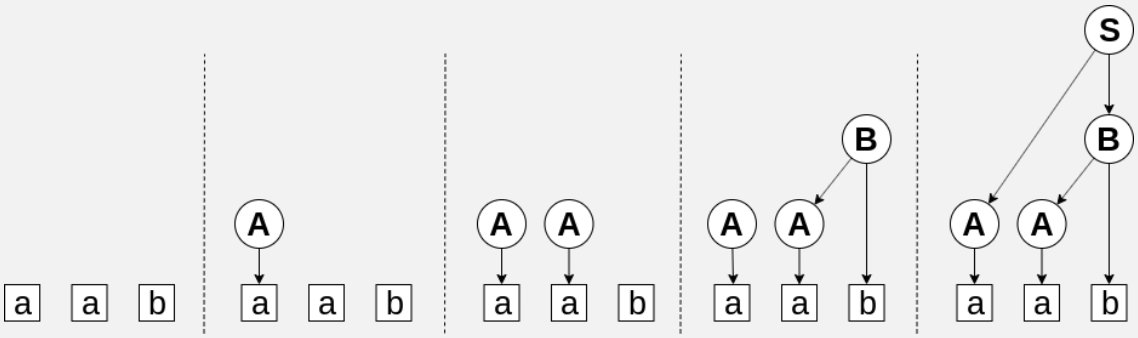
Per riconoscere la frase *aab* un parser $LL$ opererebbe in questo modo:




In questo caso l'albero viene costruito a partire dalla radice secondo la derivazione canonica sinistra

$$S \to AB \to aB \to aAb \to aab$$

Un parser $LR$ invece opera in maniera differente:




In questo caso l'albero viene costruito a partire dalle foglie secondo la derivazione canonica destra a rovescio: $aab \to Aab \to AAb \to AB \to S$

A questo punto occorre chiedersi come operi l'oracolo. Sfruttando le informazioni di contesto esso è in grado di dire con certezza quale sia l'operazione corretta da eseguire; nel caso in cui identifichi un contesto preciso ordinerà un'operazione di REDUCE specificando anche quale sia il risultato, in caso contrario ordinerà un'azione di SHIFT per raccogliere ulteriori informazioni. In questo senso l'oracolo è "semplicemente" un riconoscitore di contesti.

### Analisi $LR(0)$

Se l'analisi $LL(0)$ è pressoché inutile per la comprensione dell'analisi $LL$, per quanto riguarda l'analisi $LR$ conviene invece studiare e comprendere prima l'analisi $LR(0)$. In questo caso infatti essa significa trovare l'azione corretta senza dover guardare il prossimo simbolo di input, ma ciò non implica non aver informazioni in assoluto, poiché possiamo usare come informazione il *contesto* (e le informazioni relative al passato).

Per prima cosa occorre calcolare il contesto $LR(0)$ di ciascuna produzione (un concetto simile a quello dello starter set symbol per l'analisi $LL$). Successivamente occorre controllare che non vi siano collisioni tra i vari contesti, ovvero casi in cui una stringa appartenente ad un contesto è anche un *prefisso proprio* (prefisso di una stringa seguito da un simbolo terminale) di un altro contesto. Nel caso non vi siano collisioni si possono usare questi contesti per guidare l'analisi, in caso contrario non è possibile utilizzare l'analisi $LR(0)$ ma è necessario utilizzare almeno l'analisi $LR(1)$.

**Esempio 5.18**

Data la seguente grammatica, con i relativi contesti, analizzare la frase *abab*.

| Produzione | Contesto |
|---|---|
| $S \to aAb$ | $\{aAb\}$ |
| $S \to aaBba$ | $\{aaBba\}$ |
| $A \to Aa$ | $\{aAa\}$ |
| $A \to b$ | $\{ab\}$ |
| $B \to \epsilon$ | $\{aa\}$ |

Si noti che non c'è collisione tra il secondo ed il quinto contesto poiché la stringa $aa$, pur essendo un prefisso di $aaBba$ non è un prefisso proprio di quest'ultima (infatti in questa produzione la stringa $aa$ è seguita dal non terminale $B$).
Per prima cosa viene letta la prima $a$ della frase, ma ciò non identifica un contesto perciò si effettua una SHIFT che legge il simbolo $b$. Sullo stack ora c'è $\{a, b\}$ che identifica un contesto. Viene usata la produzione $A \to b$ per sostituire la $b$ finale con un nodo $A$, lasciando lo stack nel nuovo stato $\{a, A\}$. Poiché questa situazione non identifica alcun contesto viene eseguita una SHIFT ($\{a, A, a\}$). Questo nuovo stato identifica un contesto, ed applicando la produzione associata lo stack passa nella forma $\{a, A\}$. Leggendo nuovamente si ha $\{a, A, b\}$. Applicando un'ultima trasformazione si ottiene lo scopo $S$ della grammatica, e la frase viene riconosciuta con successo

Vi è però un caso critico, ovvero quello in cui lo scopo viene riutilizzato come parte destra in qualche produzione. In questo caso infatti non è possibile distinguere la situazione in cui sullo stack si trova lo scopo e la frase è riconosciuta da quella in cui sullo stack si trova lo scopo ma occorre leggere altro input per applicare delle altre produzioni. A questo proposito è utile introdurre un nuovo meta-simbolo che faccia da scopo vero e proprio (ovvero che non venga utilizzato nella parte destra di alcuna produzione), in modo tale da rendere immediatamente riconoscibile la situazione in cui una frase è riconosciuta.

**Esempio 5.19: caso critico**

Data la seguente grammatica, con i relativi contesti, analizzare la frase *aaa*.

| Produzione | Contesto |
|---|---|
| $S \to Sa$ | $\{Sa\}$ |
| $S \to a$ | $\{a\}$ |

Apparentemente i contesti $LR(0)$ sono diversi. Tuttavia il problema è che, poiché $S$ può comparire anche in forme di frase intermedie, ridursi a $S$ non comporta aver sicuramente finito la frase. In questo esempio infatti la prima mossa sarebbe sicuramente una riduzione di $a$ ad $S$, ma poi non sarebbe chiaro se l'analisi dovesse terminare o meno: se non ci fossero altre $a$ ci si sarebbe già ridotti allo scopo $S$, se invece ci fossero altre $a$

occorrerebbe applicare altre riduzioni. Tuttavia per saperlo occorrerebbe guardare avanti di un simbolo, cosa non possibile in una grammatica $LR(0)$. Per accorgersi chiaramente di ciò, si introduce un nuovo scopo di top level $Z$ e riformulando la grammatica si ottiene:

| Produzione | Contesto |
|---|---|
| $Z \to S$ | $\{S\}$ |
| $S \to Sa$ | $\{Sa\}$ |
| $S \to a$ | $\{a\}$ |

In questo modo si vede chiaramente che i primi due contesti collidono

A questo punto occorre descrivere il calcolo dei contesti $LR(0)$, i quali sono **sempre definiti** da una opportuna grammatica **regolare sinistra**. Come conseguenza si trova che il riconoscimento dei contesti può essere effettuato da un automa a stati finiti (facile da usare, difficile da progettare). Per ottenere questo riconoscitore vi sono fondamentalmente due vie (una volta definito in modo formale il concetto di contesto): calcolare per ogni produzione il corrispondente contesto e progettare sulla base dei contesti ottenuti il RSF corrispondente, oppure adottare un approccio pratico alla costruzione diretta del RSF.

**Definizione 5.11: contesto $LR(0)$**

$$LR(0)ctx(A \to \alpha) = \{\gamma | \gamma = \beta\alpha,\ Z \overset{*}{\Rightarrow} \beta Aw \Rightarrow \beta\alpha w,\ w \in V_T^*\}$$

Ovvero, il contesto $LR(0)$ della produzione $A \to \alpha$ è l'insieme di tutti i prefissi $\beta\alpha$ di una forma di frase che usi tale produzione all'ultimo passo di una derivazione canonica destra. Di conseguenza tutte le stringhe appartenenti al contesto $LR(0)$ di $A \to \alpha$ hanno la forma $\beta\alpha$ e differiscono per il solo prefisso $\beta$ (dipendente da $A$)

Si può osservare dunque che il contesto $LR(0)$ di una produzione può essere espresso come la concatenazione fra l'insieme dei $\beta$ (ovvero il contesto sinistro) associati a quella produzione ed il suffisso $\alpha$.

**Definizione 5.12: contesto sinistro**

Si definisce il contesto sinistro di un simbolo non terminale $A$ come

$$leftctx(A) = \{\beta | Z \overset{*}{\Rightarrow} \beta Aw,\ w \in V_T^*\}$$

**Definizione 5.13: contesto $LR(0)$**

$$LR(0)ctx(A \to \alpha) = leftctx(A) \cdot \{\alpha\}$$

Esploriamo la prima possibilità per la creazione dell'oracolo. Il calcolo dei contesti sinistri implica investigare tutti i modi in cui può apparire il meta-simbolo $A$ in una forma di frase. A questo scopo si parte da due fatti fondamentali:

1. Poiché lo scopo $Z$ della grammatica non compare mai nella parte destra di alcuna produzione abbiamo
$$leftctx(Z) = \{\epsilon\}$$
2. Data una produzione $B \to \gamma A\delta$ i prefissi che possono essere davanti ad $A$ sono quelli che potevano essere davanti a $B$ seguiti dalla stringa $\gamma$. Di conseguenza un primo contributo

a $leftctx(A)$ è $leftctx(B) \cdot \{\gamma\}$ ovvero $leftctx(A) \supseteq leftctx(B) \cdot \{\gamma\}$. Si parla di un primo contributo in quanto altri potrebbero emergere analizzando le altre regole di produzione. Nel complesso dunque $leftctx(A)$ si ottiene unendo tutti i contributi di tutte le produzioni in cui $A$ compaia nella parte destra

Una volta ottenuta la grammatica dei contesti sinistri[2] è possibile ottenere l'automa a stati finiti. Nel caso l'automa risulti deterministico, ovvero se tra i contesti non vi sono conflitti, è possibile riconoscere tale grammatica mediante analisi $LR(0)$, altrimenti occorre utilizzare degli strumenti più potenti.

Nel caso l'automa risulti deterministico si osserva che ad ognuno degli stati finali corrisponde una sola operazione di riduzione e esso dunque è etichettato da una singola produzione, e che nessuno di essi è dotato di archi uscenti (sono dei veri stati finali).

L'analisi $LR(0)$ di una frase si svolge sottoponendo all'automa caratteristico, dotato di stack, la forma di frase corrente. Durante l'analisi l'automa ordinerà:

- SHIFT: quando viene raggiunto uno stato non finale, segno che le informazioni disponibili non consentono ancora di decidere una riduzione. In questo caso si legge nuovo input e si pone il simbolo letto nello stack
- REDUCE: quando viene raggiunto uno stato finale, comunicando la necessità di applicare quella produzione per effettuare un passo di riduzione. In questo caso, non si legge alcun input in quanto viene effettuata una pura rielaborazione della forma di frase a uso interno, ma il risultato di essa **viene comunque trasferito sullo stack**. Viene effettuato un numero di pop pari al numero di elementi nella parte destra della produzione e viene posto in cima allo stack il meta-simbolo sul lato sinistro della produzione stessa. Quando viene posto in cima allo stack lo scopo della grammatica la frase viene accettata

Questo modello di base presenta in realtà alcune inefficienze. Dopo ogni riduzione l'automa infatti riparte dallo stato iniziale, ma in realtà ciò non è necessario in quanto la forma di frase contenuta nello stack viene modificata solamente in fondo. Per questo motivo il percorso effettuato tra gli stati sarà identico al precedente a meno di qualche variazione nella parte finale, e risulta quindi utile ripartire dall'ultimo stato utile e non sempre dal primo. A tal fine può risultare comodo uno stack ausiliario degli stati per tener traccia del percorso effettuato.

**Condizione 5.3: condizione sufficiente**

Condizione sufficiente perché una grammatica sia $LR(0)$ è che, date due produzioni

$$A \to \alpha \qquad B \to \omega$$

se

$$\theta \in LR(0)ctx(A \to \alpha),\ \theta w \in LR(0)ctx(B \to \omega)$$
$$\theta \in V^*,\ w \in V_T^*$$

risulti:

- $w = \epsilon$
- $A = B$
- $\alpha = \omega$

Ovvero ogni stato di riduzione dell'automa deve essere etichettato da una sola produzione e non deve avere archi di uscita etichettati da simboli terminali

[2]Per un esempio completo dell'utilizzo di questi due postulati si vedano le slide 31-37 del pacco c14

L'approccio appena seguito è difficilmente meccanizzabile e complesso, pertanto risulta opportuna la ricerca di un approccio alternativo. Esso consiste nell'utilizzo di un procedimento operativo che salti il calcolo dei contesti e che proceda direttamente alla costruzione dell'automa partendo dalla regola di top level $Z \to S\$$ ed analizzando, via via che si presentano, le varie situazioni possibili. Ogni situazione verrà scritta all'interno di un $LR$ item, che contiene oltre alla regola "di base" anche le altre regole ricorsivamente chiamate. Inoltre viene introdotta la notazione di *cursore* per indicare la posizione raggiunta in un dato momento nell'analisi di una produzione. Presentiamo un esempio[3] per descrivere meglio il procedimento.

**Esempio 5.20: procedimento operativo**

Si faccia riferimento alla grammatica:

$$\begin{aligned} &Z \to S \\ &S \to aSAB \mid BA \\ &A \to aA \mid B \\ &B \to b \end{aligned}$$

All'inizio tutto l'input è da leggere, inoltre ogni frase inizia da $Z$. Questa situazione è schematizzata da:

$$\Big[Z \to .S\$\Big]$$

A destra del cursore vi è una $S$, ma nulla specifica cosa sia. Pertanto occorre includere all'interno dell'item le produzioni associate a quel meta-simbolo ed ai meta-simboli che potrebbero essere incontrati dopo il cursore all'interno di esse, in modo che tutti i meta-simboli immediatamente incontrabili siano conosciuti. L'item diventa in questo modo:

$$\begin{bmatrix} Z \to .S\$ \\ S \to .aSAB \mid .BA \\ B \to .b \end{bmatrix}$$

Esso costituisce il primo stato dell'automa. A questo punto dobbiamo valutare le possibili evoluzioni dell'automa a partire da questo stato. Leggendo da input (o dallo stack) l'automa potrebbe incontrare (in maniera mutuamente esclusiva) i simboli \$ (terminando la valutazione), $a$, $B$, $b$. Ognuna di queste ipotesi rappresenta uno stato diverso

$$\Big[Z \to S.\$\Big] \qquad \Big[B \to b.\Big]$$

che rappresentano stati finali (rispettivamente di accettazione e di riduzione)

$$\begin{bmatrix} S \to a.SAB \\ S \to a.SAB|.BA \\ B \to .b \end{bmatrix} \qquad \begin{bmatrix} S \to B.A \\ A \to .aA|.B \\ B \to .b \end{bmatrix}$$

che rappresentano stati intermedi.
Ripetendo il procedimento, ovvero facendo via via avanzare il cursore in ognuno degli item, si nota che alcuni stati si ripeteranno.
Al termine del procedimento i vari item rappresentano gli stati dell'automa, mentre il simbolo usato per far avanzare il cursore rappresenta il valore dell'arco necessario ad effettuare la transizione

Utilizzando questo procedimento tecnicamente le azioni da intraprendere non sono due,

[3] Si faccia riferimento alle slide 56-67 del pacco c14 per i passaggi completi

bensì quattro. Ogni arco corrisponde ad uno SHIFT, ma se l'arco corrisponde ad un simbolo terminale è uno *SHIFT* vero e proprio se invece l'arco corrisponde ad un metacarattere si dice una azione di *GOTO* in quanto corrisponde ad una lettura dello stack interno, e dunque non una vera e propria lettura da input, ma una transizione di stato.
Ogni stato finale invece corrisponde ad un'azione di *REDUCE*, che viene chiamata *ACCEPT* nel caso in cui vi sia la riduzione allo scopo della grammatica.

Usando questo approccio è anche possibile costruire la tabella di parsing $LR(0)$ associata alla grammatica, in cui per ogni cella viene indicata quale sia l'azione da intraprendere (SHIFT, GO-TO, REDUCE, ACCEPT). Ovviamente molte celle risulteranno vuote al termine del processo in quanto saranno tutte le azioni proibite (corrispondenti agli stati di errore).

Si noti che, poiché un parser $LR(0)$ o legge da input o riduce, all'interno della tabella tutte le colonne lecite corrispondenti ad uno stato di riduzione conterranno la medesima azione di riduzione

### Analisi $LR(k)$

A livello pratico capita molto spesso che sorgano dei conflitti nel caso si utilizzi l'analisi $LR(0)$. L'analisi $LR(k)$ opera in maniera analoga all'analisi $LR(0)$ ma guardando avanti di $k$ simboli in modo da separare apparenti conflitti. Come conseguenza tutte le riduzioni sono ritardate di $k$ simboli e le definizioni di contesto e contesto sinistro includono i $k$ simboli successivi. Tuttavia questa complessità fa sì che già l'analisi $LR(1)$ sia difficilmente realizzabile, pertanto si è fatto ricorso a semplificazioni atte a rendere il processo più agevole.

**Definizione 5.14: contesto $LR(k)$**

Formalmente, il contesto $LR(k)$ di una produzione $A \to \alpha$ è così definito:

$$LR(k)ctx(A \to \alpha) = \{\gamma \mid \gamma = \beta\alpha u,\ Z \overset{*}{\Rightarrow} \beta A u w \Rightarrow \beta\alpha u w,\ w \in V_T^*,\ u \in V_T^k\}$$

Ovvero tutte le stringhe del contesto per la produzione $A \to \alpha$ hanno la forma $\beta\alpha u$ e differiscono per il prefisso $\beta$ e per la stringa $u$, appartenente all'insieme $FOLLOW_k(A)$

**Definizione 5.15: insieme $FOLLOW_k(A)$**

L'insieme $FOLLOW_k(A)$ è così formalmente definito:

$$FOLLOW_K(A) = \{u \in V_T^k \mid S \Rightarrow \gamma A u \beta\} \qquad \gamma, \beta \in V^*$$

Esso non è altro che l'insieme delle stringhe di lunghezza $k$ che possono seguire il meta-simbolo $A$

**Definizione 5.16: contesto sinistro**

Si definisce il contesto sinistro di un simbolo non terminale $A$ come

$$leftctx(A, u) = \{\beta \mid Z \overset{*}{\Rightarrow} \beta A u w,\ w \in V_T^*,\ u \in V_T^k\}$$

Si ottiene inoltre una ridefinizione dei precedenti postulati:

- $leftctx(Z, \epsilon) = \{\epsilon\}$
- data una produzione $P \to \gamma Q \delta$ si ha
  $leftctx(Q, u) \supseteq leftxctx(P, v) \cdot \{\gamma\},\ v \in FOLLOW_k(P)),\ u = FIRST(L(\delta) \cdot v, k)$

implicando nuovamente una grammatica regolare sinistra

Da questa definizione se ne può ricavare una nuova per il contesto $LR(k)$ di una produzione.

**Definizione 5.17: contesto $LR(k)$**

$$LR(k)ctx(A \to \alpha) = \bigcup_{u \in FOLLOW_k(A)} leftctx(A, u) \cdot \{\alpha u\}$$

Per il calcolo dell'automa riconoscitore per una grammatica $LR(1)$ si procede idealmente come nel caso $LR(0)$, ovvero calcolando le espressioni regolari per i contesti $LR(1)$ e usandole per costruire l'automa[4]. Tuttavia questo approccio, già di per sé non banale né immediato nel caso $k = 0$, diventa ancora più complesso per $k > 0$ in quanto una grammatica $G$ dotata di $n$ meta-simboli e $t$ terminali comporta una grammatica dei contesti sinistri con potenzialmente $(n-1)t^k + 1$ meta-simboli. Da ciò deriva la necessità di trovare approcci alternativi semplificati e meno onerosi.

Per la costruzione di un parser $LR(1)$ ci si può nuovamente affidare ad un procedimento operativo analogo[5] a quello applicabile nel caso $LR(0)$, specificando questa volta anche il simbolo successivo che permette l'applicazione della regola. Di conseguenza per ogni regola è ora necessario computare anche il lookahead set che consente di tener conto anche del simbolo successivo.

### Analisi approssimate

L'esperienza conferma che è praticamente impossibile costruire un parser $LR(1)$ per un "vero" linguaggio, poiché le sue dimensioni sarebbero intrattabili. Per questo motivo esistono delle versioni semplificate come $SLR$ e $LALR(1)$, le quali si basano sull'idea di accorpare gli stati simili, differendo però nel modo in cui questo viene effettuato.

**Analisi $SLR$** L'idea alla base dell'analisi $SLR$ è cercare di distinguere i contesti $LR(0)$ utilizzando il lookahead set (quindi produzioni con il medesimo contesto potrebbero essere distinte grazie a lookahead set differenti).

**Definizione 5.18: contesto $SLR$**

Formalmente, il contesto $SLR(k)$ di una produzione $A \to \alpha$ è così definito:

$$SLR(k)ctx(A \to \alpha) = LR(0)ctx(A \to \alpha) \cdot FOLLOW_k(A)$$

È possibile dimostrare inoltre che

$$SLR(k)ctx(A \to \alpha) \supseteq LR(k)ctx(A \to \alpha)$$

ovvero che il contesto $SLR$ è un sovrainsieme del contesto $LR$, e dunque è leggermente più esposto a potenziali conflitti

L'idea di base dunque è quella di provare a calcolare il contesto $LR(0)$ e nel caso si presentino conflitti calcolare il contesto $SLR$ per tentare di risolverli. Questo calcolo si dimostra sufficientemente rapido, in particolar modo nel caso in cui si possegga già il contesto $LR(0)$ di una data produzione.

[4] Si veda l'esempio a pagina 85-94 del pacco c14
[5] Si veda l'esempio a pagina 96-107 del pacco c14

Esiste però un procedimento più rapido e semplice per ottenere il parser $SLR$ a partire dal parser $LR(0)$, ovvero basta eliminare dalla tabella di parsing tutte quelle riduzioni che risultano incompatibili con il lookahead set. Una produzione $A \to \alpha$ va inserita nella tabella di parsing solamente se il successivo simbolo appartiene all'insieme $FOLLOW(A)$.
In pratica dunque nel parser $LR(0)$ le riduzioni sono presenti per tutti i simboli terminali possibili in quanto si sceglie di non leggere da input il prossimo simbolo, mentre un parser $SLR$ ha la capacità di controllare il carattere successivo e quindi di applicare una produzione solamente nel caso in cui il carattere successivo sia quello corretto. (All'atto pratico questo si traduce in una cancellazione di alcune celle della tabella di parsing, che equivale a dire "se il prossimo simbolo è $x$ allora la mossa giusta è uno SHIFT e non una riduzione").

**Analisi** $LALR(1)$ Un approccio alternativo consiste nel collassare quegli stati[6] che nell'analisi $LR(1)$ sono identici a meno del lookahead set. Questa trasformazione è sempre possibile e spesso molto conveniente in quanto un parser $LALR$ risulta composto da molti meno stati.

Tuttavia eseguendo questa operazione possono comparire nuovi conflitti reduce/ reduce (ovvero a partire dal medesimo stato sono possibili due riduzioni differenti) che spesso sono gestibili. Un altro vantaggio di questo approccio è che permette di eseguire i calcoli a partire dai calcoli fatti per $LR(0)$ aggiungendo i lookahead set.

[6]Per chiarimenti si vedano gli esempi da pagina 130 del pacco c14

# Capitolo 6

# Interpretazione e valutazione

Un interprete è più di un puro riconoscitore in quanto è in grado di eseguire delle azioni (svolte immediatamente o in un secondo tempo) sulla base alla semantica della frase riconosciuta. In questi casi la sequenza di derivazione diventa importante poiché contribuisce ad attribuire significato alle frasi, dunque, ad esempio, non è sempre possibile sostituire le grammatiche ricorsive a sinistra con quelle ricorsive a destra.
Solitamente un interprete è strutturato in due componenti:

- Un **analizzatore lessicale** (*scanner*) con il compito di analizzare le parti *regolari* del linguaggio fornendo al *parser* le singole parole già aggregate, occupandosi dunque dei dettagli relativi ai singoli caratteri
- Un **analizzatore sintattico e semantico** (*parser*) con il compito di valutare la correttezza della sequenza dei token ottenuti dallo scanner. Opera quindi sulle parti *context-free* del linguaggio

Spesso queste due componenti sono organizzanti in un'architettura client/server.

## 6.1 Analisi

### 6.1.1 Analisi lessicale

Consiste nell'individuazione delle singole parole che compongono una frase raggruppando i caratteri di input secondo le produzioni regolari associate alle diverse categorie lessicali. Durante l'analisi lessicale i token possono essere categorizzati sulla base dello stato finale in cui si trova lo scanner, tuttavia questa non è sempre una strategia vincente. Spesso infatti un linguaggio consta di varie categorie di token, e cablarle tutte all'interno della struttura del RSF lo renderebbe eccessivamente complicato.

Una soluzione alternativa consiste nell'utilizzare delle tabelle che contengano questo grado di conoscenza. In prima battuta lo scanner riconosce dunque degli identificatori che, in un successivo momento, possono venir riconsiderati sulla base del contenuto delle tabelle. Come risultato si ottiene uno scanner modulare ed estensibile.

In Java queste funzionalità possono essere ottenute mediante l'utilizzo delle classi `Scanner` e `Regex` (dal pacchetto `java.util`).

### 6.1.2 Analisi sintattica

L'analisi top-down ricorsiva discendente (vedi Definizione 5.6) offre, in presenza di grammatiche $LL(1)$, uno strumento semplice per costruire un riconoscitore, tuttavia nel caso si voglia passare ad un interprete occorre propagare qualcosa di più rispetto ad un valore booleano, ovvero

un singolo valore (se l'obiettivo è la valutazione immediata) o un albero (se l'obiettivo è una valutazione differita).

**Esempio 6.1: espressioni aritmentiche**

Consideriamo la creazione di una grammatica $LL(1)$ per l'analisi di espressioni aritmetiche ed un interprete per tale grammatica (considerando solamente le quattro operazioni e le parentesi). In prima battuta si potrebbe pensare ad una grammatica come

$$\begin{aligned} EXP &::= EXP + EXP \\ EXP &::= EXP - EXP \\ EXP &::= EXP * EXP \\ EXP &::= EXP/EXP \\ EXP &::= num \end{aligned}$$

Questa grammatica è ambigua in quanto consente di avere due alberi sintattici per la derivazione della medesima frase (vedi Esempio 4.14 e Definizione 4.18), in più non consente di esprimere formalmente la priorità tra gli operatori

**Esempio 6.2: espressioni aritmentiche (continua)**

Per esprimere associatività e priorità è possibile adottare una grammatica a "strati" come

$$\begin{aligned} EXP &::= TERM \\ EXP &::= EXP + TERM \\ EXP &::= EXP - TERM \\ TERM &::= FACTOR \\ TERM &::= TERM * FACTOR \\ TERM &::= TERM/FACTOR \\ FACTOR &::= num \\ FACTOR &::= (EXP) \end{aligned}$$

Il primo strato è il meno prioritario mentre il terzo è il più prioritario. È importante notare che la grammatica è ricorsiva sinistra in modo da rispettare l'associatività degli operatori, nel caso in cui venissero aggiunti operatori associativi a destra (come l'esponenziazione) le corrispondenti regole sarebbero ricorsive a destra. Poiché però la grammatica è ricorsiva a sinistra non è possibile applicare l'analisi ricorsiva discendente.
A questo proposito, come anticipato, sarebbe possibile cambiare le ricorsioni sinistre con delle ricorsioni destre cambiando però l'associatività degli operatori. Oppure si potrebbe pensare di rimuovere direttamente l'associatività obbligando l'utente ad utilizzare sempre le parentesi per la scrittura delle espressioni

### Ricorsione sinistra

Come osservato nell'esempio 6.2, a volte la grammatica presenta delle ricorsioni sinistre che non possono essere eliminate a meno di modificare la semantica del linguaggio. In questo casi è possibile utilizzare uno stratagemma atto a rendere possibile l'analisi ricorsiva discendente. È possibile creare una seconda grammatica (che non presenti ricorsioni sinistre) ed utilizzarla solamente per l'implementazione del parser. In questo modo la prima grammatica, ovvero quella

reale, sarà descrittiva delle caratteristiche del linguaggio, mentre la seconda renderà possibile un'implementazione agevole del parser.

**Esempio 6.3: espressioni aritmentiche (continua)**

Partendo dalla grammatica descritta dal precedente esempio il primo passo consiste nel riconsiderare i sotto-linguaggi generati dai diversi strati. Si può osservare che essi sono regolari e pertanto descrivibili tramite espressioni regolari:

$$L(EXP) = TERM\ ((+|-)\ TERM)^*$$
$$L(TERM) = FACTOR\ ((*|/)\ FACTOR)^*$$

Successivamente possiamo utilizzare la notazione EBNF per denotare una ripetizione senza dover far uso diretto della ricorsione.

$$L(EXP) = TERM\ \{(+|-)\ TERM\}$$
$$L(TERM) = FACTOR\ \{(*|/)\ FACTOR\}$$

Come si può notare queste regole non presentano più una ricorsione diretta ma descrivono un processo computazionale iterativo, che può essere implementato senza far uso della ricorsione.
La grammatica così realizzata consente l'analisi ricorsiva discendente, in quanto, prendendo come esempio un $EXP$, o successivamente vi è un segno $(+|-)$ seguito da un $TERM$ oppure vi è la stringa vuota e quindi la fine dell'input.
Questa grammatica può essere dunque utilizzata per l'implementazione del parser. Ad esempio la funzione `parseExp()`, deputata al riconoscimento dei termini $EXP$ può così essere implementata:

```
public boolean parseExp() {
  boolean t1 = parseTerm();
  while (currentToken != null) {
    if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      boolean t2 = parseTerm();
      // Accumulo il risultato a sinistra
      t1 = t1 && t2;
    } else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      boolean t2 = parseTerm();
      t1 = t1 && t2;
    } else {
      return t1;
    }
  }
  return t1;
}
```

### 6.1.3 Analisi semantica

Una volta effettuata l'analisi lessicale e l'analisi sintattica, al fine di ottenere un interprete, occorre fornire una valutazione semantica delle frasi lette. Se il linguaggio è *finito* è sufficiente un semplice elenco che metta in evidenza la relazione tra le frasi (corrette) ed il loro significato, mentre nel caso il linguaggio sia infinito occorre utilizzare una notazione finita in grado di racchiudere la nozione di infinito.

In prima battuta si potrebbe pensare di utilizzare una funzione in grado di mappare le frasi ed i token del linguaggio sui vari significati, tuttavia questo richiederebbe di enumerare vari casi esplicitamente.

Un approccio alternativo consiste nell'associare ad ogni regola sintattica una regola semantica, in modo da non lasciare nulla indietro durante l'elaborazione. In questo modo vengono realizzate funzioni, simili a quelle sintattiche che, secondo l'associatività richiesta dal contesto, sono in grado di attribuire un significato alle varie espressione mentre vengono lette.

**Esempio 6.4: espressioni aritmetiche (continua)**

Riprendendo l'esempio delle espressioni aritmetiche, la funzione per la valutazione semantica dei blocchi $EXP$ può essere così implementata:

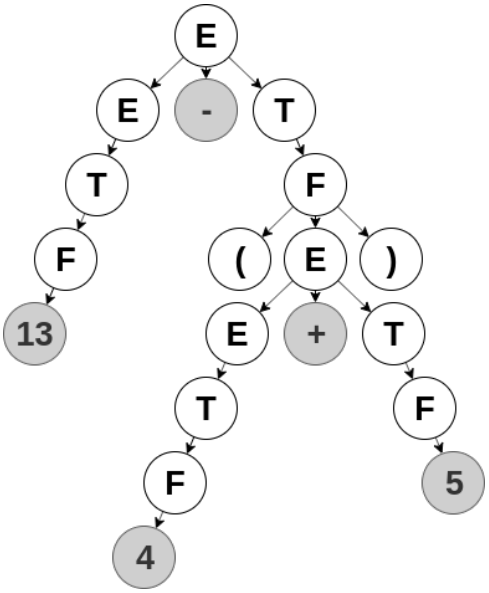
```
public int parseExp() {
  int t1 = parseTerm();
  while (currentToken != null) {
    if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      int t2 = parseTerm();
      // Accumulo risultato a sinistra
      t1 = t1 + t2;
    } else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      int t2 = parseTerm();
      t1 = t1 - t2;
    } else {
      // Il token letto non fa parte di L(Exp)
      return t1;
    }
  }
  // Fine dell'input
  return t1;
}
```

## 6.2 Valutazione differita

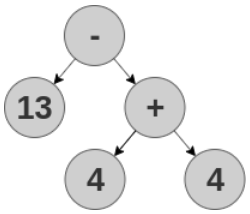
Le tecniche appena descritte possono andar bene nel caso in cui la semantica non cambi o nel caso in cui sia necessaria una valutazione in tempo reale del significato delle espressioni. Tuttavia nel caso si voglia ottenere una valutazione differita queste tecniche non sono più adatte e vanno modificate. Per prima cosa le varie funzioni atte a occuparsi della semantica non restituiranno più dei valori puntuali ma elaboreranno degli alberi (che racchiudono in sintesi il significato della frase).

Gli alberi ottenuti dalle sequenze delle chiamate a funzione contengono tuttavia varie informazioni non essenziali, derivanti da tutti i passi di derivazione. Ad esempio sapere che per ottenere un *num* occorre passare prima per $EXP$ e poi per $TERM$ ed infine per $FACTOR$ risulta importante durante la costruzione dell'albero, tuttavia al momento dell'elaborazione queste informazioni non sono rilevanti. Tale albero, più compatto, viene chiamato *Abstract Syntax Tree* (AST).

Si distingue quindi la **grammatica concreta** delle frasi, la quale contiene le regole ed i simboli utili a rendere chiaro il significato per chi legge, ma non strettamente necessari (ovvero che non aggiungono significato, ma guidano la lettura), dalla **grammatica astratta** degli alberi, la quale viene utilizzata durante l'elaborazione all'interno degli alberi e contiene le informazioni essenziali ad attribuire un significato alle frasi.




(a) Albero concreto




(b) Albero astratto

Figura 6.1: Esempi di albero astratto e di albero concreto

Di tutti i nodi solamente alcuni sono significativi, mentre altri non lo sono e dunque possono essere rimossi:

- Alcuni nodi foglia contengono informazioni necessarie durante la lettura, ma che una volta generato l'albero hanno esaurito la loro funzione(ad esempio per la corretta interpretazione dell'ordine degli operatori nelle espressioni aritmetiche le parentesi non sono più necessarie ottenuto l'albero)
- I nodi aventi un unico figlio, ovvero che non generano biforcazioni, possono essere sostituiti con il figlio, in quanto il percorso per la lettura dell'albero diventa obbligato
- i nodi terminali non legati ad alcun significato sul piano semantico, tipicamente segni di punteggiatura o zucchero sintattico, possono essere rimossi

Questi nodi possono essere rimossi in modo da ottenere l'**albero sintattico astratto** (Figura 6.1b). Una volta ottenuto tale albero la rappresentazione dell'informazione diventa univoca e l'albero stesso fornisce l'ordine *corretto* di valutazione. A partire dal medesimo albero astratto è poi possibile costruire diversi valutatori che, visitandolo nel modo opportuno, produrranno il risultato desiderato (es. sintesi vocale, compilazione del codice, elaborazione dei dati, ...).

Risulta utile inoltre introdurre una **sintassi astratta** avente l'obiettivo di descrivere come sia fatto l'AST. Questa sintassi *non è destinata all'utente*, ma viene utilizzata internamente per dare una rappresentazione dell'albero in termini delle strutture dati che poi lo andranno a costruire. Va notato che questa sintassi non è unica per un dato problema, in quanto dipende fortemente dalla forma che si vuol conferire alla soluzione.

**Esempio 6.5: espressioni aritmetiche (continua)**

Riprendendo l'esempio delle espressioni aritmetiche, si potrebbe utilizzare la seguente

sintassi astratta

$$EXP ::= EXP + EXP$$
$$EXP ::= EXP - EXP$$
$$EXP ::= EXP * EXP$$
$$EXP ::= EXP/EXP$$
$$EXP ::= num$$

Da cui derivano le classi utilizzate all'interno del modello dei dati. Utilizzando tali classi la funzione adibita al parsing delle espressioni diventa:

```
public Exp parseExp() {
  Exp termSeq = parseTerm();
  while (currentToken != null) {
    if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      Exp nextTerm = parseTerm();
      if (nextTerm != null) {
        // costruzione APT a sinistra
        termSeq = new PlusExp(termSeq, nextTerm);
      } else {
        return null;
      }
    } else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      Exp nextTerm = parseTerm();
      if (nextTerm != null) {
        // costruzione APT a sinistra
        termSeq = new MinusExp(termSeq, nextTerm);
      } else {
        return null;
      }
    } else {
      return termSeq;
    }
  } // end while
  return termSeq;
}
```

### 6.2.1 Valutazione degli alberi

Una volta ottenuto l'AST esso può essere valutato in più modi da più valutatori differenti. Ad esempio un primo valutatore potrebbe restituire la lettura del testo fornito (es. "Roberto" di Trenitalia), oppure fornire una valutazione del contenuto della frase inserita.
Esistono, inoltre, più modalità con cui valutare un albero, nel caso specifico un AST:

- **Pre-order**: viene valutata prima la radice e poi i figli nell'ordine. Produce una notazione prefissa
- **Post-order**: vengono valutati i figli nell'ordine e solo successivamente la radice. Produce una notazione postfissa
- **In-order** (solo per gli alberi binari): viene valutato il figlio sinistro, poi la radice e successivamente il figlio destro. Produce una notazione infissa, la quale ha il difetto però di appiattire l'albero facendo perdere la nozione di livello. Per questo motivo, nel caso

in cui venga prodotta una notazione infissa occorre utilizzare degli stratagemmi grafici, quali parentesi o colori, atti a riportare l'informazione circa i livelli che altrimenti verrebbe perduta

La notazione infissa è quella che, culturalmente, ci risulta più semplice nell'uso matematico, tuttavia questa rende necessaria l'introduzione della priorità degli operatori e delle parentesi per indicare quale sia l'ordine desiderato per l'esecuzione delle operazioni. Questi problemi sono assenti nel caso vengano utilizzate la notazione prefissa o la notazione infissa, esse infatti hanno il pregio di indicare quali siano gli operandi separatamente da quali siano gli operatori, in modo che risulti semplice l'esecuzione dell'operazione e che non sia necessario specificare esplicitamente quale sia la priorità degli operatori.

**Esempio 6.6: confronto fra le notazioni**

| Notazione infissa | Notazione prefissa | Notazione postfissa |
|---|---|---|
| $9-4-1$ | $--9\ 4\ 1$ | $9\ 4-1-$ |
| $9-(4-1)$ | $-9-4\ 1$ | $9\ 4\ 1--$ |

Nel caso della notazione prefissa, ovvero la classica notazione funzionale, infatti viene specificata prima quale sia l'operazione da effettuare e solo successivamente quali siano i due operandi. Nella notazione postfissa invece vengono specificati prima gli operandi e solo successivamente l'operazione da svolgere. Questa ultima notazione ha il pregio di essere interpretabile in maniera estremamente semplice da una macchina.

Da quest'ultima idea si può pensare di introdurre una macchina a stack per l'esecuzione dei programmi: ogni volta che viene incontrato un simbolo (numerico in questo caso) esso viene aggiunto sullo stack, ogni volta che viene incontrato un operatore esso implica due pop dallo stack, l'esecuzione dell'operazione ed infine una push per inserire il risultato sullo stack.

**Esempio 6.7: valutazione da parte di una macchina a stack**

Data l'operazione

$$13\ 4\ 5\ -\ -$$

si potrebbe ottenere il seguente codice macchina:

```
push 13
push 4
push 5
sub \\include due pop per estrarre i dati ed una push per inserire
    il risultato
sub
pop \\pop finale per restituire il risultato
```

### 6.2.2 Implementazione del valutatore

Il valutatore incorpora la **funzione di valutazione**, ovvero deve visitare l'albero applicando ad ogni nodo la semantica prevista per quel tipo di nodo (discriminando quale tipo di nodo stia visitando).

Un primo approccio di implementazione prevede una funzione statica che esegua la giusta operazione in base al tipo del nodo. Di conseguenza la struttura sarà costituita da una lunga serie di blocchi condizionali necessari a discriminare i vari casi, rendendo la soluzione verbosa ed inefficiente.

Un secondo approccio è quello di rompere la serie di blocchi condizionali incapsulando ognuno di essi in un metodo di un oggetto: questa metodologia è più efficiente, pratica e orientata

ad oggetti. Se questo elenco di funzioni viene aggiunto al parser non si hanno sostanzialmente miglioramenti rispetto al caso precedente, tuttavia se esse vengono aggiunte all'AST stesso è possibile discriminare il tipo di nodo per mezzo del polimorfismo.

Mettiamo ora a confronto la due metodologie appena presentate: una metodologia funzionale:

- Permette di separare nettamente la sintassi dall'interpretazione poiché si basa su di una funzione di interpretazione esterna alle classi.
- Facilita l'introduzione di nuove interpretazioni, perché basta scrivere una nuova funzione senza toccare le classi già esistenti
- Rende però più oneroso introdurre una nuova produzione, perché occorre modificare il codice di tutte le funzioni di interpretazione per tenere conto della nuova struttura

Una metodologia basata su oggetti invece:

- Utilizzando un metodo di interpretazione specializzato all'interno di ogni classe facente parte dell'AST, è sufficiente invocare il metodo perché venga eseguita l'azione corretta
- Facilita l'aggiunta di nuove produzioni, perché basta definire una nuova sottoclasse con il relativo metodo di valutazione senza la necessità di modificare le classi già esistenti
- Rende più oneroso introdurre nuove interpretazione, perché occorre definire il nuovo metodo in ogni classe

La scelta tra i due stili non è neutra, n quanto influenza modularità ed estensibilità del codice.

### Implementazione di un compilatore

Una volta ottenuto l'interprete ottenere un compilatore è relativamente semplice, in quanto se l'interprete restituisce una valutazione della frase letta un compilatore restituisce una meta-valutazione della stessa frase, ovvero una riscrittura in un codice intermedio.

Nella realtà però le interpretazioni da effettuare su di uno stesso codice sono molteplici (analisi semantica, type checking, generazione de codice, ...). Ad una prima vista l'approccio funzionale sembra più opportuno, sebbene vada contro i principi dell'ingegneria del software. Perciò sovente si utilizza il **pattern visitor** per descrivere le varie interpretazioni da effettuare sul codice. Questo pattern permette di ottenere i vantaggi descritti dalla seconda modalità sopra descritta separando però il codice applicativo relativo all'interpretazione da quello relativo al modello dei dati. Tutte le classi derivanti da `Exp` implementeranno il metodo `accept()` per accettare la visita da parte delle classi che implementano l'interfaccia `IExpVisitor`.

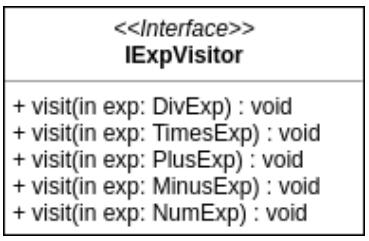



Figura 6.2: Esempio di visitor per le espressioni numeriche

**Esempio 6.8: visitor per le stampe di un'espressione**

Il visitor necessario a stampare le espressioni in forma prefissa

```
class ParExpVisitor implements IExpVisitor {
  String curs = "";

  public String getResult() { return curs; }
```

```
    protected void visitOpExp(OpExp e) {
      e.left().accept(this);
      String sleft = getResult();
      e.right().accept(this);
      String sright = getResult();
      curs = "(" + e.myOp() + " " + sleft + " " + sright + ")";
    }

    public void visit( PlusExp e ) { visitOpExp(e); }
    public void visit( MinusExp e ) { visitOpExp(e); }
    public void visit( TimesExp e ) { visitOpExp(e); }
    public void visit( DivExp e ) { visitOpExp(e); }
    public void visit( NumExp e ) { curs = "" + e.getValue(); }
  }
```

Il metodo `getResult()` risulta utile per leggere il risultato alla fine di ogni valutazione in quanto il valore della variabile `curs` viene riscritto da ogni analisi

### 6.2.3 Assegnamento e variabili

Ogni linguaggio di programmazione prevede l'introduzione del concetto di operatore di assegnamento e di variabile. Sebbene spesso si utilizzi l'operatore "=" per gli assegnamenti, esso non è né riflessivo né simmetrico, al contrario dell'operatore matematico "=" rappresentante l'uguaglianza matematica. A questo proposito alcuni linguaggi come il Pascal utilizzano un diverso operatore per gestire l'assegnamento (":=") che evidenzia la non riflessività dell'operatore stesso.

La peculiarità dell'assegnamento, soprattutto quello in stile C, è che il simbolo di variabile ha un significato diverso a seconda del fatto che si trovi a sinistra o a destra dell'uguale. Infatti una variabile che si trovi a sinistra dell'uguale rappresenta in realtà l'area di memoria in cui andare ad inserire i dati, mentre la variabile che si trovi a destra dell'uguale rappresenta il valore contenuto all'interno dell'area di memoria (si parla dunque di L-value, e di R-value), e dunque il simbolo di variabile è *overloaded*.

Inoltre si possono effettuare altre scelte per quanto riguarda le caratteristiche dell'assegnamento. L'assegnamento può essere *distruttivo* o *non distruttivo* a seconda del fatto che una variabile possa cambiare valore durante l'esecuzione del programma o meno. Tipicamente i linguaggi imperativi sono dotati di assegnamento distruttivo in quanto partono dall'astrazione della macchina di Von Neumann e dal concetto di cella di memoria riscrivibile, mentre i linguaggi logici, che si basano sulla logica matematica, tipicamente sono dotati di assegnamento non distruttivo (si creano sempre nuove variabili).

Associato al concetto di variabile vi è quello di *environment*, inteso come elenco di coppie simbolo valore. Nel caso la semantica dell'assegnamento sia quella dell'assegnamento distruttivo è consentita la modifica dei valori associati a variabili già esistenti all'interno dell'environment, in caso contrario tale modifica non è consentita, ma è consentito solamente l'inserimento di nuove variabili al suo interno.
Spesso risulta comodo dotarsi di environment multipli, soprattutto nel caso in cui si sia in presenza di un linguaggio di programmazione dotato di assegnamento distruttivo. In questo caso l'enviroment è suddiviso in vari sottoambienti separati tra loro e dotati di un ben specifico tempo di vita. Di conseguenza ogni modello computazionale deve specificare quale sia il campo di visibilità (*scope*), ovvero quali environment siano visibili in un dato punto della struttura fisica del programma. L'environment globale contiene il valore delle variabili il cui tempo di vita è l'intero programma, mentre gli environment locali contengono le coppie il cui tempo di vita

non coincide con quello del programma. Questi ultimi sono tipicamente legati all'attivazione di funzioni o ad altre strutture presenti a runtime.

Di base lo scopo dell'assegnamento è quello di produrre una modifica dell'environment, non quello di produrre un valore; di conseguenza l'assegnamento, almeno in linea teorica, sarebbe un'istruzione e non un'espressione. Tuttavia interpretando l'assegnamento come istruzione si rende impossibile la realizzazione dell'assegnamento multiplo come `x=y=z=2` (che rende palese **l'associatività a destra** dell'assegnamento).

**Esempio 6.9: assegnamento multiplo**

L'espressione `x=y=z=2` può essere interpretata come `x=(y=(z=2))`. Si può notare che il valore 2 fluisce da destra a sinistra

Ipotizzando di voler introdurre l'assegnamento all'interno di un ipotetico linguaggio di programmazione occorre anche decidere se diversificare o meno la sintassi di L-value e di R-value. Nel caso in cui si scelga di *non* diversificare la sintassi per L-value e R-value la grammatica ottenuta, limitatamente agli assegnamenti, diviene però $LL(2)$, e dunque richiede una bufferizzazione maggiore da parte del parser[1]. Nei linguaggi reali spesso si accetta questo compromesso, ma alcuni linguaggi che hanno necessità di essere particolarmente efficienti, come quelli interpretati in tempo reale, scelgono di diversificare le due sintassi (vedi Bash).

**Esempio 6.10: espressioni aritmetiche (continua)**

Si supponga di riesaminare la grammatica delle espressioni aritmetiche precedente e di voler aggiungere il concetto di assegnamento[a]. La grammatica assume la forma:

$$
\begin{aligned}
EXP &::= TERM \\
EXP &::= EXP + TERM \\
EXP &::= EXP - TERM \\
EXP &::= ASSIGN \\
ASSIGN &::= IDENT = EXP \\
TERM &::= FACTOR \\
TERM &::= TERM * FACTOR \\
TERM &::= TERM/FACTOR \\
FACTOR &::= num \\
FACTOR &::= \$IDENT \\
FACTOR &::= (EXP)
\end{aligned}
$$

Si nota perciò che saranno necessarie tre nuove classi `AssignExp`, `RIdentExp` (che rappresenta il nome di una chiave all'interno dell'environment per ricercare una variabile) e `LIdentExp` (che è dotato di un nome e di un valore all'interno dell'environment). La valutazione dunque di un token del tipo `RIdentExp` ha lo scopo di inserire (o modificare a seconda della semantica) all'interno dell'environment un valore, mentre la valutazione di un token del tipo `LIdentExp` ha lo scopo di cerca e restituire un valore che deve essere presente all'interno dell'environment

[a] Per l'esempio completo si veda il pacco c08 di slide

[1] Infatti, se non si differenzia sintatticamente l'identificatore "sinistro" da quello "destro", per distinguerli bisogna per forza verificare se il token successivo è l'operatore =

# Capitolo 7

# Modelli computazionali

## 7.1 Paradigma imperativo e paradigma dichiarativo

Solitamente siamo abituati ad utilizzare il **paradigma imperativo**, il quale prevede che un programma non sia altro che una **sequenza di ordini**. Ciò è intuitivo poiché comandare è facile ed è immediato mappare su una macchina mini-azioni che eseguono un determinato codice. Tuttavia un elenco di ordini è intrinsecamente non invertibile, ovvero per esprimere la situazione "inversa" occorre un elenco di ordini completamente diverso anche se in origine la relazione fra i dati era simmetrica.

**Esempio 7.1**

L'equazione $x = y - 2$ in matematica è simmetrica poiché dato $y$ si ricava $x$ e viceversa, mentre un programma imperativo pensato per ricavare $x$ non può essere usato per ricavare $y$

### 7.1.1 Il paradigma dichiarativo

Il paradigma imperativo non è l'unico possibile. Esiste infatti il **paradigma dichiarativo** in cui non si esprimono ordini, ma **relazioni fra entità**. Esso è sicuramente meno intuitivo ma ha il vantaggio di essere invertibile poiché si limita ad affermare *ciò che è vero*. Perciò in un programma dichiarativo si possono esprimere relazioni così come sono, demandando a runtime la scelta di usarla in un verso o nell'altro.

### 7.1.2 Basi di Prolog

Un programma Prolog è un inseme di regole espresse secondo la seguente notazione:

```
testa :- corpo.
```

L'operatore `:-` esprime l'implicazione logica $\leftarrow$, ovvero se il **corpo è vero**, allora anche la **testa è vera** (non viceversa). Se il corpo manca esso viene assunto come vero, dunque la testa di tali istruzioni risulta sempre vera. In tal caso l'istruzione viene chiamata **fatto** e l'operatore `:-` viene omesso. La testa ha la forma

```
funtore(lista_argomenti)
```

nel caso la lista degli argomenti sia assente le parentesi possono essere omesse. Il corpo invece è una congiunzione di termini separati da virgole, dove ogni termine ha anch'esso la forma

```
funtore(lista_argomenti)
```

Inoltre sia testa che corpo possono contenere variabili, le quali devono necessariamente iniziare con la maiuscola.

**Esempio 7.2**

```
p(a,12X) :- q(a), r(13), s(X,1).
s(Y,X) :- q(X), r(Y).
```

Si noti come in questo caso la testa sia vera solo se tutti i termini del corpo lo sono. Inoltre lo scope di una variabile è la singola regola, perciò la variabile `X` nella prima regola non ha nulla a che vedere con la variabile `X` della seconda regola

Infine se occorrono termini che iniziano per la maiuscola (ma che non siano variabili) o che contengano caratteri non standard, occorre racchiuderli tra apici.

**Esempio 7.3: genitori e figli**

Fatti (assiomi):

```
uomo(adam).
uomo(peter).
donna(mary).
donna(eve).
genitore(adam, peter).
genitore(eve, peter).
genitore(adam, paul).
genitore(mary, paul).
```

Regole:

```
padre(Y,C) :- uomo(Y), genitore(Y,C).
madre(X,C) :- donna(X), genitore(X,C).
```

Innanzitutto stabiliamo che i fatti `genitore(a,b)` si leggano come "`a` genitore di `b`". Conseguentemente le due regole sono di facile interpretazione, ovvero "`Y` è padre di C se `Y` è un uomo ed è anche genitore di `C`". Analogamente "`X` è madre di `C` se `X` è una donna ed è anche genitore di `C`". Quindi come in un database si ha una conoscenza di base (fatti), ma non solo poiché si hanno anche regole che permettono di dedurre nuove conoscenze. Infatti le possibili query possono:

- Recuperare tuple dal "database":

  ```
  ?- donna(X)
  > X=mary ? ;
  > X=eve ? ;
  > no
  ```

  Il motore Prolog fornisce un nome di donna alla volta e si ferma in attesa di un OK o di uno skip. In quest'ultimo caso ne elenca un altro e così via. Se non vi sono più risultati restituisce `no`.

- Recuperare tuple da sintetizzare al volo:

  ```
  ?- padre(X,paul)
  > X=adam
  > yes

  ?- padre(eve, paul)
  > no
  ```

**Esempio 7.4: equazioni**

Per i numeri naturali non è opportuno adottare la rappresentazione classica[a] poiché i simboli da essa utilizzati hanno quel significato solo nella nostra mente, ma non sono una *rappresentazione esplicita*. Possiamo quindi indicare con `s(N)` il successore di un numero naturale `N` e con `eq(X,Y)` la relazione fra `X` e `Y`. Dunque l'equazione `x+2=y` si può esprimere tramite la relazione:

```
eq(X, s(s(X))).
```

Ora si può interrogare il sistema Prolog fornendo coppie di elementi (ottenendo risposte del tipo `yes/no` in base al fatto che soddisfino o meno l'equazione) o uno solo elemento ottenendo l'altro.
Si può sfruttare la relazione anche per far generare tutte le possibili soluzioni fornendo all'equazione numeri naturali via via diversi. Si può utilizzare a tal proposito un generatore di numeri naturali, ad esempio:

```
num(1).
num(s(N)) :- num(N).
```

Ovvero 1 è un naturale e se `N` è un naturale anche il suo successore lo è. Invocando `num(X)` si otterranno tutte le soluzioni, ovvero `1`, `s(1)`, `s(s(1))`, ...
Tuttavia nella pratica esprimere i numeri con la relazione `s(N)` risulta molto scomodo. Il linguaggio Prolog accetta quindi anche numeri reali nell'usuale notazione decimale. Per calcolare il valore di un'espressione numerica tuttavia non si può utilizzare il classico = poiché in Prolog esso indica unificazione sintattica e non semantica. `Value = 13-4` dà come risultato la struttura `13-4` ossia `'-'(13,4)` e non 9. Per ottenere 9 bisogna usare il predicato (non invertibile) `is`, ovvero `Value is 13-4`. In questo modo prima di assegnare un valore alla variabile `Value` viene fatta una valutazione semantica del lato destro nel dominio dei numeri reali

[a] ovvero 1, 2, 3, ...

**Esempio 7.5: append su liste**

Siano `l1` e `l2` due liste di cui si vuole fare l'append. Imperativamente occorre stabilire *a priori* gli argomenti di input e di output oppure distinguere i tre casi. Ad esempio in Java si può utilizzare il metodo `addAll`.

```
l3 = l1.addAll(l2)
```

Andando però ad esaminare il codice della `addAll` ci si accorge di come esso sia relativamente lungo, error prone, difficile da debuggare e con una "ossessione del controllo" che tratta la macchina come un mero esecutore incapace di autonomia.
Con l'approccio dichiarativo di Prolog, invece, basta esprimere solo due regole:

```
append([],L,L).
append([T|C],L,[T|Cr]) :- append(C,L,Cr).
```

La prima regola stabilisce che appendendo una lista `L` ad una lista vuota `[]` si riottiene sempre `L`. La seconda stabilisce che appendendo `L` a una lista la cui testa sia `T` e la cui coda sia `C`, si ottiene una nuova lista avente per testa `T` e per coda il concatenamento `Cr` di `C` e `L`.
Un esempio di utilizzo è trovare i valori delle variabili tali che:

```
?- append([a,b], [c,4,d], R).
> Solution: R / [a,b,c,4,d]
```

```
?- append([a,b], X, [a,b,c,d]).
> Solution: X / [c,d]
```

## 7.2 Processi computazionali iterativo e ricorsivo

È bene non confondere un processo computazionale con il costrutto che lo rappresenta all'interno di un linguaggio di programmazione, pertanto occorre chiedersi cosa distingua un processo computazionale iterativo da uno ricorsivo.

Nei linguaggi imperativi, il costrutto linguistico che esprime un processo computazionale iterativo è tipicamente il ciclo. Al di là della sintassi specifica di ogni linguaggio, ogni ciclo possiede una variabile che funge da *accumulatore* (pertanto l'assegnamento è necessariamente distruttivo) che deve essere inizializzata prima del ciclo e che deve essere modificata durante il ciclo. Tale variabile, al termine del ciclo, contiene il risultato finale dell'intero processo. Di conseguenza, immaginando di congelare l'esecuzione al passo $k$ del ciclo, l'accumulatore conterrebbe il risultato parziale $k$-esimo, in questo senso un processo iterativo computa **in avanti**.

**Esempio 7.6: calcolo iterativo del fattoriale**

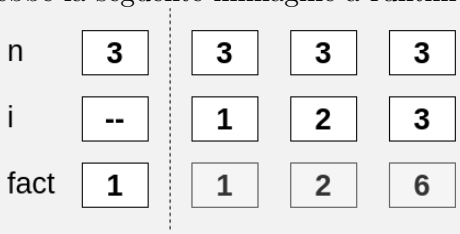
```
int fact = 1;

for (int i = 1; i <= n; i++) {
  fact = fact * i;
}

printf(fact)
```

Come si può notare nel caso ci si fermi a metà del processo la variabile `fact` conterrebbe un valore parziale

Nel caso in cui `n=3` si avrebbe la seguente immagine a runtime




Poiché i processi iterativi dispongono dell'assegnamento distruttivo sono estremamente efficienti nell'uso della memoria, tuttavia non permettono un debug semplice in quanto per capire cosa sia successo è necessario rieseguire linea per linea il codice.

Al contrario un processo ricorsivo è tipicamente espresso tramite una **funzione ricorsiva**. Al di là della sintassi specifica tale schema non prevede alcun accumulatore, infatti ogni chiamata si preoccupa di ottenere il risultato $(k-1)$-esimo e di sintetizzare il risultato $k$-esimo a partire da esso. Durante le chiamate non vi è alcun risultato parziale in quanto solamente una volta raggiunto il caso base del processo è possibile sintetizzare il primo risultato, poi utilizzato per sintetizzare via via gli altri risultati, pertanto il processo in questo caso computa **all'indietro**.

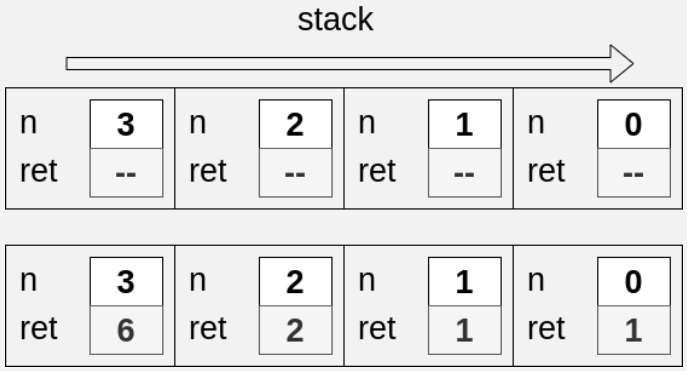
**Esempio 7.7: calcolo ricorsivo del fattoriale**

```
int fact (int n) {
  return (n == 0) ? 1 : fact(n - 1) * n;
}

printf(fact(3))
```

Il primo risultato ottenibile è quello della chiamata `fact(0)`, ogni chiamata si occupa di sintetizzare poi nuovi valori aumentando il consumo di memoria ma permettendo un debug più agevole.

Nel caso in cui `n=3` si avrebbe la seguente immagine a runtime




Si nota in questo caso una fase ascendente in cui il problema viene sgranato, ma non vi è alcuna computazione ed una fase discendente in cui viene sintetizzato ogni risultato a partire dal precedente

In alcuni casi è possibile esprimere un processo computazionale iterativo mediante un costrutto sintatticamente ricorsivo, ciò accade nel caso in cui l'istruzione ricorsiva sia l'ultima della funzione. Si parla in tal caso di *funzione ricorsiva in coda*.

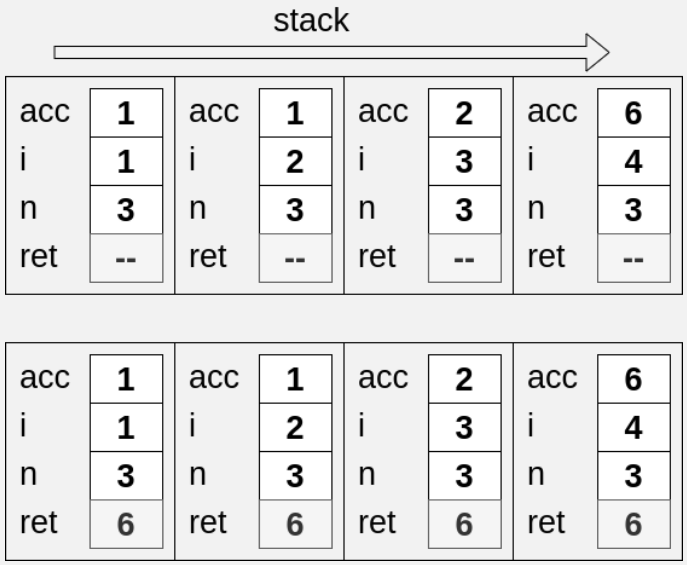
**Esempio 7.8: calcolo ricorsivo in coda del fattoriale**

```
int factIt(int acc, int i, int n) {
  return i > n ? acc : factIt(i * acc, i + 1, n - 1);
}

printf(factIt(1,n))
```

Si noti come la variabile `acc` porti in avanti il risultato parziale. Come nel caso del ciclo la computazione avviene in avanti, infatti interrompendo l'iterazione al passo $k$ si otterrebbe il risultato $k$-esimo. Questo processo però si basa sulla sintesi di nuovi valori che però possono sovrascrivere i precedenti perché computando in avanti questi ultimi non sono più necessari quando si fa una nuova chiamata.
In questo caso non sono necessari tutti i record di attivazione, ma ne basta solamente uno, facendo sì che mano a mano i nuovi record vadano a sovrascrivere il precedente. In questo modo si riesce a risparmiare memoria, ma si perde la traccia dell'esecuzione

Nel caso in cui `n=3` si avrebbe la seguente immagine a runtime




Si noti che l'immagine a runtime è identica a quella di un processo iterativo.

Poiché la ricorsione in coda permette di esprimere un processo computazionale iterativo mediante costrutti ricorsivi può essere usata in alternativa ai cicli per esprimere il concetto di iterazione. Tipicamente i linguaggi funzionali ed i linguaggi logici tendono a seguire questo approccio ottimizzando il runtime, mentre i linguaggi imperativi, possedendo i cicli e l'assegnamento distruttivo, non ottimizzano la ricorsione in coda.
Questa ottimizzazione può essere effettuata allocando il nuovo record di attivazione al di sopra del record precedente, in tal modo l'occupazione delle risorse è identica a quella del caso ciclico. Di conseguenza ricorsione in coda e iterazione mediante cicli sono solo forme diverse dello stesso processo computazionale.

**Esempio 7.9**

Questa caratteristica può essere ottenuta in linguaggi come Scala o Kotlin su richiesta. In Scala è ottimizzata di default (disattivabile tramite un'opzione di compilazione).

```
def factIt(acc:Int, i:Int, n:Int):Int = {
  if (i>n) acc else factIt(i * acc, i + 1, n)
}
```

In Kotlin invece viene ottimizzata su richiesta mediante l'utilizzo della keyword `tailrec`

```
tailrec fun factIt(acc:Int, i:Int, n:Int):Int = {
  return if (i > n) {
    acc
  } else {
    factIt(i * acc, i + 1, n)
  }
}
```

Empiricamente si può dimostrare che con l'attivazione della Tail Recurtion Optimization si ha un risparmio all'incirca del 60% delle risorse stack utilizzate per realizzare la funzione fattoriale

# Capitolo 8

# Programmazione funzionale

Storicamente, i paradigmi di programmazione e i linguaggi funzionali hanno dato un notevole contributo di idee forti, ma sono stati penalizzati dalla sintassi poco user-friendly per il grande pubblico facendo sì che questi linguaggi rimanessero confinati all'interno dell'ambiente accademico. Nell'ultimo decennio invece tali linguaggi hanno guadagnato popolarità, spesso anche grazie alla nascita di linguaggi blended, che uniscono idee del mondo funzionale ad idee del mondo imperativo.

Alcune idee chiave del paradigma funzionale sono:

- La distinzione tra variabili e valori: se da un lato i linguaggi imperativi si basano sul cambio di valore dei simboli, il paradigma funzionale si basa sul fatto che i simboli mantengano il loro significato. I linguaggi blended uniscono i due mondi dando la possibilità di fare entrambe le cose con parole chiave diverse ad esempio `val` e `var`, perciò supportando uno stile di programmazione imperativo offrono (ed in certi casi impongono[1]) l'uso di uno stile più funzionale
- Costrutti considerati come espressioni (*everything is an expression*): i linguaggi imperativi spesso fanno distinzione tra istruzioni (le quali non ritornano alcun valore) ed espressioni. Questo induce ad esprimere sequenze di operazioni facendo uso di variabili di appoggio, le quali riducono la leggibilità dei programmi. I linguaggi blended invece, tipicamente, posseggono solamente espressioni permettendo di porle in cascata senza il bisogno di variabili di appoggio
- Collezioni di oggetti immodificabili (*compute by synthesis*): le collezioni modificabili presentano problemi in caso di concorrenza e promuovono uno stile imperativo basato sulla modifica delle stesse. I linguaggi funzionali invece suggeriscono di computare per sintesi di nuovi oggetti, aiutando la gestione delle situazioni di concorrenza. Tipicamente i linguaggi blended forniscono sia collezioni modificabili che collezioni immodificabili, facilitando l'uso di queste ultime
- Funzioni come "first class entities"
- Concisione ed operatori come funzioni

Queste caratteristiche, inserite all'interno di un linguaggio blended, permettono di rafforzarne gli aspetti ad oggetti abolendo:

[1] Ad esempio le proprietà introdotte da `val` possono avere solamente metodi getter e non metodi setter in quanto il loro valore non può cambiare nel tempo, rafforzando a livello sintattico quello che altrimenti sarebbe stato espresso al più con un commento

- I tipi primitivi (*everything is an object*): molti linguaggi ad oggetti tradizionali si basano su tipi primitivi. Questo causa una mancanza di uniformità e comporta la necessità di un trattamento specifico per queste entità, ad esempio i tipi primitivi sono spesso passati per valore mentre gli oggetti sono passati per riferimento. I linguaggi blended risolvono queste differenze rendendo tutto un oggetto, spesso distinguendo però tra classi-valore (immodificabili e non estendibili) e classi-riferimento (le classiche classi)
- Le parti statiche (*don't be static*): i linguaggi ad oggetti tradizionali si basano sul concetto di classe, che differisce enormemente da quello di oggetto. Questo porta alla necessità dei design pattern. I linguaggi blended introducono il concetto di **companion object** che contiene le parti "statiche" di un altro oggetto sebbene non sia statico

## 8.1 Funzioni come first-class entities

Tutti i linguaggi di programmazione supportano il concetto di funzione ed i relativi costrutti sintattici, ma di rado esse sono supportate come entità aventi "pieno diritto di cittadinanza". Una funzione che gode di queste caratteristiche può essere manipolata come un qualsiasi altro tipo di dato, quindi:

- poter essere assegnata a variabili (di tipo funzione)
- poter essere passata come argomento ad un'altra funzione
- poter essere restituita come risultato da un'altra funzione
- poter essere definita al volo mediante una sintassi literal come altri tipi di dato
- poter essere definita eventualmente senza un nome

Nei linguaggi imperativi invece le funzioni sono dei puri contenitori di codice e dunque non possono essere manipolate come gli altri tipi di dato. I linguaggi ad oggetti moderni stanno incorporando questi concetti, sebbene sotto varie forme. Ad esempio javascript ha da sempre il costruttore `Function` e la parola chiave `function` che consente di specificare funzioni anonime, C# possiede delegati e costruttori `Func<>`, Java possiede le lambda expression (abbastanza limitate).
I linguaggi blended come Scala e Kotlin, d'altro canto, propongono nativamente questi concetti.

La possibilità di poter passare una funzione come argomento ad un'altra funzione o di ritornare funzioni permette la generazione di funzioni di ordine superiore, ovvero *funzioni che manipolano funzioni*. Per poter eseguire queste operazioni però è necessario possedere un tipo funzione che consente alle istanze di essere valori, ma al contempo di essere eseguibili.

**Esempio 8.1**

```
//definire una funzione anonima al volo
var f = function (z) { return z*z; }

//creare al volo una funzione che ne riceve un'altra come parametro
   e che computa il risultato sulla base di essa
var ff = function (f, x) { return f(x); }

//una funzione puo' restituire una nuova funzione come risultato
function ff(){return function(r){return r+10}}
```

## 8.2 Variabili libere e chiusure

Se un linguaggio ammette variabili libere, ovvero non definite localmente, tali funzioni dipendono dal contesto circostante. Di conseguenza sono necessari dei criteri con cui darvi significato, ovvero **chiusura lessicale** o **chiusura dinamica**.

Nei linguaggi tradizionali l'esistenza di variabili libere non crea particolari problemi. In C, ad esempio, le uniche variabili libere possibili sono quelle globali, mentre in Pascal (che offre la possibilità di creare funzioni innestate) c'è la necessità di distinguere le catene di chiamate.

Nel caso in cui il linguaggio supporti le funzioni come first-class entities la presenza di variabili libere comporta la nascita del concetto di **chiusura**, ovvero di un oggetto funzione ottenuto chiudendo rispetto ad un certo contesto una definizione di funzione che aveva variabili libere.

**Esempio 8.2**

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

var f1 = ff( Math.sin, 0.8)
var f2 = ff( function(q){return q*q}, 0.8)

var r1 = f1(3) // 3.717356091
var r2 = f2(0.36) // 1.000000000 [0.64 + 0.36]
```

Invocando `ff` si ottiene dunque un oggetto funzione che al suo interno incorpora i riferimenti alle variabili `f` e di `x` e dunque tale funzione *anonima* deve mantenere traccia dei loro valori. Una seconda esecuzione di `ff` con parametri differenti, infatti, comporta la creazione di una seconda funzione anonima *differente* dalla prima. I parametri `f` e `x` sono variabili libere per la funzione anonima, poiché non definite localmente ad essa

Questo concetto è da sempre disponibile nei linguaggi che contengono le funzioni come first-class entities, ma recentemente è stato aggiunto anche nei linguaggi OO nella forma delle lambda expression (spesso però depotenziate) con forti conseguenze sul modello computazionale.

In presenza di chiusure il tempo di vita delle variabili di una funzione non coincide più necessariamente con quello della funzione che le contiene, dunque alcune variabili (non tutte, solo quelle usate nelle chiusure) devono essere allocate sullo heap e non più semplicemente sullo stack. Infatti una variabile locale ad una funzione esterna può essere indispensabile al funzionamento della funzione interna, dunque il tempo di vita di tale variabile non può coincidere con quello della funzione esterna, ma deve necessariamente coincidere con quello della chiusura, anche se la funzione che la definisce termina prima.
Nell'esempio precedente i parametri della funzione `ff` devono essere allocati sullo heap in quanto devono sopravvivere alla terminazione della funzione in modo da poter essere usati dalla funzione anonima generata in qualsiasi momento.

Poiché tuttavia queste variabili possono essere condivise possono presentarsi problemi in ambienti concorrenti a memoria comune. Per questo motivo in linguaggi come Java le variabili presenti all'interno della chiusura possono essere accedute solamente in lettura e mai in scrittura (devono essere dunque `final` o `effectively final`).

In generale le chiusure possono essere utilizzate per:

- Rappresentare uno stato interno privato[2] (in assenza di atri strumenti) perché le variabili contenute nella chiusura non sono visibili da fuori

[2]Tecnica utilizzata in linguaggi come Javascript

- Creare un canale di comunicazione nascosto, in quanto definendo più funzioni nella stessa chiusura esse condividono uno stato interno, che può essere usato anche per comunicare privatamente
- Definire nuove strutture di controllo, in quanto la funzione esterna esprime il controllo mentre quella ottenuta come parametro esprime il corpo da eseguire
- Riprogettare e semplificare le API ed i framework di largo uso permettendo di esprimere funzioni parametriche che ricevano il comportamento desiderato come parametro

### 8.2.1 Chiusure nei linguaggi mainstream

Le chiusure sono presenti da anni in linguaggi come Javascript, Scala e Ruby. Ora sono presenti anche in Java (>1.8), C#, C++ (>11).

**Chiusure in Java 8 e C#** Questi due linguaggi offrono le chiusure sotto forma di lambda expression con una notazione leggermente differente. In C# vengono introdotte tramite:

```
{ params => body }
```

mentre in Java

```
(params) -> {body}
```

In entrambi i casi `params` è una lista di parametri tipati separati da virgole, mentre `body` è una lista di istruzioni. Il valore di ritorno, se presente, è dato dal valore dell'ultima espressione.

**Esempio 8.3: chiusure in Java 8**

```
(int x) -> {x + 1}
(String s, int y) -> System.out.println(s + " " + y)
```

In entrambi i casi le variabili di chiusura sono solamente in lettura (effectively final), perciò un tentativo di modifica risulta in un errore di compilazione. Lo scopo è cercare di evitare effetti collaterali difficilmente prevedibili in presenza di multi-threading

**Chiusure in C++** Sono state introdotte le labda expression con la sintassi

```
auto f = [ext_var_ref] (args)
```

Dove la keyword `auto` lascia dedurre al compilatore il tipo dell'oggetto funzione. Questo viene tipicamente fatto poiché i tipi per le lambda expression possono diventare estremamente complessi molto rapidamente. La sezione tra parentesi quadre consente di specificare quali siano (e come accedervi) le variabili libere, ovvero le variabili di chiusura.

**Esempio 8.4: chiusura in C++**

```
#include <iostream>

int main() {
  int start = 20;
  //definizione di una lambda che considera tutte le variabili
      locali come variabili di chiusura in lettura e scrittura
  auto myfunc = [&] (int x) {
    std::cout << x << std::endl;
    int y = x * start;
    std::cout << y << std::endl;
    start = y;
```

```
    };

    myfunc(10);
    myfunc(5);
  }
```

La prima chiamata a `myfunc` stampa il valore `10`, modifica il valore `start` assegnandogli il nuovo valore `200`. La seconda chiamata invece stampa il valore `5` ed assegna a `start` il nuovo valore `1000`

**Chiusure in Scala, Kotlin e Javascript** Questi tre linguaggi offrono le funzioni come first class entities e rendono possibile definire dei literal di tipo funzione. Nel caso in cui la definizione di un literal includa delle variabili libere nasce una chiusura.
La sintassi Scala:

```
(params) => body
```

permette di definire un oggetto di tipo `(A,B,C,...) => R`.
La sintassi Kotlin:

```
(params) -> body
```

permette di definire un oggetto di tipo `(A,B,C,...) -> R`.
Mentre la sintassi javascript:

```
function() { body }
```

permette di definire un oggetto di tipo `function`.

### 8.2.2 Criteri di chiusura

A questo punto occorre definire un criterio riguardante il come ed il dove cercare le variabili libere da chiudere. In particolare, in presenza di funzioni innestate ci si pone il problema se debba prevalere la catena di ambienti di definizione innestati secondo il codice, ovvero se debba prevalere la definizione più prossima della variabile (**catena lessicale**), oppure se debba prevalere la catena degli ambienti attivi, ovvero se debba prevalere la definizione più vicina secondo l'ordine delle chiamate (**catena dinamica**). Queste due catene sono generalmente differenti.

**Esempio 8.5: criteri di chiusura**

```
var x = 20;

function provaEnv(z) {
  return z + x;
}

function testEnv() {
  var x = -1;
  return provaEnv(18);
}
```

Nel caso prevalga la catena lessicale il risultato ottenuto alla fine sarebbe $20 + 18 = 38$, tuttavia nel caso prevalga la catena dinamica il risultato sarebbe $18 - 1 = 17$ in quanto la funzione `testEnv`, che chiama `provaEnv` ridefinisce il valore della variabile `x`

Nel caso venga scelta la catena lessicale il modello computazionale segue la **chiusura lessicale**. Questa da un lato vincola a priori il valore di una variabile, ma proprio per questo motivo

permette di eseguire un debug efficace e di leggere il testo del programma stesso (sarebbe una follia debuggare un programma che si comporta in maniera differente a seconda dell'ordine di sequenza delle chiamate).
Se invece prevale la catena dinamica si parla di **chiusura dinamica**. Essa è molto più potente della chiusura lessicale permettendo una grande dinamicità del codice in quanto i simboli vengono legati ai valori sul momento, tuttavia questa potenza riduce le possibilità di debug.

Poiché tipicamente la comprensione del comportamento del codice è cruciale praticamente tutti i linguaggi di programmazione adottano la **chiusura lessicale**.

### 8.2.3 Modelli per la valutazione delle funzioni

Ogni linguaggio che introduca funzioni deve prevedere un modello computazionale per la loro valutazione, il quale deve stabilire **quando** vengano valutati i parametri, **cosa** venga passato alla funzione e **come** si attivi la funzione.

Tradizionalmente il modello più usato è il cosiddetto **modello applicativo**. Esso prevede che tutti i parametri vengano valutati all'atto della chiamata, che alla funzione vengano passati i valori dei parametri (o l'indirizzo in memoria[3]) e che la chiamata a funzione segua il modello sincrono.
Questo modello è semplice, pratico ed efficiente in quanto valuta i parametri una sola volta, rende facile seguire il flusso di esecuzione e passando dei valori viene ridotto l'impatto per il trasferimento dei dati. Tuttavia ha alcuni svantaggi, infatti valutando tutti i parametri sempre e comunque potrebbe venir fatto del lavoro inutile (si pensi ai casi in cui sulla base di una condizione interna servano i primi due parametri e non gli altri oppure viceversa), ma soprattutto nei casi in cui tale valutazione produca un errore questo modello porta ad un fallimento che si sarebbe potuto evitare.

**Esempio 8.6: modello applicativo in Java**

```
class Esempio {
  static void printOne(boolean cond, double a, double b) {
    if (cond) {
      System.out.println("result = " + a);
    } else {
      System.out.println("result = " + b);
    }
  }

  public static void main(String[] args) {
    int x=5, y=4;
    printOne(x>y, 3+1, 3/0); //ArithmeticException: div by zero
  }
}
```

In questo caso la valutazione immediata porta ad una eccezione che si sarebbe potuta evitare

**Esempio 8.7: modello applicativo in Javascript**

```
var f = function(flag, x) {
  return (flag<10) ? 5 : x;
}
```

[3]Anche gli indirizzi costituiscono dei valori

```
var b = f(3, abort() ); // Errore!!
document.write("result =" + b);
```

Anche in questo caso il modello computazionale applicativo causa un fallimento non necessario valutando tutti i parametri a prescindere dalla loro reale utilità all'interno della chiamata corrente. In questa specifica chiamata poiché `flag<10 == true` la funzione non avrebbe mai utilizzato il parametro `x`

Esistono vari altri modelli per la chiamata a funzione, tra cui il modello **call by name**. Questo modello prevede che i parametri vengano valutati al momento del loro utilizzo e non all'atto della chiamata. A tal fine i parametri passati non sono dei valori ma degli oggetti computazionali (eseguibili a richiesta), di conseguenza, a seconda del flusso di esecuzione della funzione, un parametro potrebbe non essere mai valutato. Utilizzando queste modalità l'insieme delle funzioni che terminano con successo aumenta sensibilmente. Riprendendo gli esempi precedenti, se Java o Javascript utilizzassero questo modello, l'esecuzione non avrebbe dato problemi, in quanto i parametri problematici non sarebbero mai stati valutati.

Sebbene sia utile e potente questo modello è raramente utilizzato all'interno dei linguaggi di largo uso perché:

- La valutazione di uno stesso parametro potrebbe avvenire più volte, riducendo l'efficienza. Questo problema sarebbe risolvibile mediante una cache (modello **call by need**)
- Sono richieste più risorse a runtime in quanto è necessario gestire degli oggetti computazionali e non dei semplici valori
- È richiesta una macchina virtuale capace di effettuare una valutazione pigra dei parametri (*lazy evaluation*). In questa maniera non si valuta tutto a priori, ma volta per volta. Chiaramente il tempo di esecuzione aumenta perché non si hanno i valori "già pronti".
- Permette di catturare pochi casi che spesso sono dei veri e propri errori di programmazione da risolvere (quindi meglio che esploda tutto subito piuttosto che in faccia all'utente). Inoltre molti casi di fallimento aritmetici sarebbero risolvibili arricchendo l'aritmetica mediante i concetti di `NaN` ed infinito (come avviene in Javascript)

Se un linguaggio supporta questa estensione della matematica anche il modello applicativo risulta particolarmente potente e sicuro.

**Esempio 8.8: modello applicativo in Javascript (matematica arricchita)**

```
printOne = function (cond, a, b) {
  if (cond) {
    println("result = " + a);
  } else {
    println("result = " + b);
  }
}

var x=5, y=4;
printOne(x>y, 3+1, 3/0);
```

Questo esempio non fallisce in quanto l'espressione `3/0` restituisce il valore `Infinity`

Sebbene il modello call by name non sia molto diffuso esso può essere *simulato*.

**Esempio 8.9: call by name simulata in Javascript**

```
var f = function(flag, x) { //i parametri sono due funzioni
  return (flag()<10) ? 5 : x(); //invoca le due funzioni
}

var b = f( function(){ return 3 }, function(){ abort() } ); //ok!
document.write("result =" +b);
```

In C possono essere utilizzate delle macro per ritardare la valutazione dei parametri[4].

**Esempio 8.10: call by name simulata in C**

```
#define f(FLAG, X) (((FLAG)<10) ? 5 : (X))

main()
{
  float b = f( 3, abort()); // non da' errore!
}
```

Questo esempio funziona grazie alla valutazione in corto circuito dell'operatore ternario

Più interessante è il caso dei linguaggi che possiedono le funzioni come first-class entities. In questo caso è sufficiente passare alle funzioni altre funzioni come argomento e non dei semplici valori, in modo che la valutazione di queste funzioni possa avvenire solo al momento più opportuno.

**Esempio 8.11: call by name simulata in Scala**

```
object CallByName0 {
  def printTwo(cond: Boolean, a: Unit=>Int, b: Unit=>Int):Unit = {
    if (cond) {
      println("result = " + a() );
    } else {
      println("result = " + b() );
    }
  }

  def main(args: Array[String]) {
    val x=5;
    val y=4;
    printTwo(x>y, (Unit => 3+1), (Unit => 3/0) );
  }
}
```

In Scala è possibile passare delle funzioni come parametro ed eseguirle solo al momento

Fino a Java 7 questo risultava molto complesso, in quanto non erano presenti nemmeno le lambda expression, da Java 8 in poi la situazione è migliorata lato cliente ma è rimasta comunque complessa per chi implementa le funzioni. In particolare la nomenclatura delle lambda expression in Java segue un modello basato sul nome di interfacce[5] e non sulla firma della funzione stessa (che renderebbe le cose molto più semplici come in Scala o Go). In C# invece

[4] Anche se poi non ci si capisce più niente

[5] I nomi delle interfacce vanno saputi tutti a memoria

la soluzione proposta è più completa ed elegante permettendo di definire delegati e di invocarli come vere funzioni e non come metodi di una interfaccia.

**Esempio 8.12: call by name simulata in Java**

```
static void printTwo(boolean cond, IntSupplier a, IntSupplier b) {
  if (cond) {
    System.out.println("result = " + a.getAsInt() );
  } else {
    System.out.println("result = " + b.getAsInt() );
  }
}

public static void main(String[] args) {
  int x=5, y=4;
  printTwo(x>y, () -> 3+1, () -> 3/0 );
}
```

Poiché non esiste un tipo funzione autonomo ogni lambda è vista in realtà come un'istanza di un'opportuna interfaccia funzionale. In questo caso `IntSupplier` è una lambda che ritorna un `int` ed è dotata di un metodo `getAsInt()`

In linguaggi come Scala è tuttavia data la possibilità di attivare la call by name su richiesta mediante l'operatore =>. Sotto banco il runtime provvederà a gestire le lambda per la JVM, ma il tutto viene mascherato per l'utente rendendone più semplice l'utilizzo.

**Esempio 8.13: call by name nativa in Scala**

```
object CallByName0ByName {
  def printTwo(cond: Boolean, a: =>Int, b: =>Int):Unit = {
    if (cond) {
      println("result = " + a );
    } else {
      println("result = " + b );
    }
  }

  def main(args: Array[String]) {
    val x=5;
    val y=4;

    printTwo(x>y, 3+1, 3/0 );
  }
}
```

Scala supporta nativamente il modello call by name permettendo un salto espressivo. In questo caso l'operatore => non è legato ad una lambda expression ma è l'espressione della keyword *by name*

# Capitolo 9

# Javascript

Javascript è un linguaggio interpretato object-based non dotato del concetto di classe che adotta un approccio funzionale con una sintassi leggera. Per questi motivi è adatto a creare applicazioni multi paradigma. Tuttavia, poiché il linguaggio si è diffuso estremamente velocemente, il linguaggio incorpora anche alcune caratteristiche che lo rendono difficile da utilizzare correttamente (come un ambiente globale mal definito, alcuni operatori contorti, loose typing[1] ed ereditarietà *prototype based*[2]).

## 9.1 Le basi del linguaggio

Javascript, inventato da Netscape, viene inizialmente denominato Livescript e poi rinominato Javascript dopo un accordo con Sun. È un linguaggio object based ma non object oriented, infatti, sebbene esistano gli oggetti, essi non sono definiti da classi.

I motori Javascript possono essere trovati all'interno dei browser, come macchine virtuali a sé stanti oppure integrati all'interno di API incluse in altri linguaggi.

### 9.1.1 Elementi linguistici

Il tipo `string` denota stringhe **immutabili** di caratteri Unicode ed è dotato di proprietà, come `length`, e metodi, come `substring()`. Come in Java tramite l'operatore `+` è possibile sintetizzare delle nuove stringhe. Esse sono definite da un'apposita sintassi literal che vede l'uso di apici singoli o doppi.

Il tipo `number` denota dei reali a 64 bit. Questo da un lato consente di evitare le conversioni e di effettuare sempre operazioni tra i reali evitando problemi di overflow, dall'altro implica che l'utilizzo di operatori bitwise necessiti di una conversione al volo del valore in un intero[3]. Le costanti numeriche sono sequenze di caratteri numerici non racchiuse tra apici (singoli o doppi). Inoltre sono definite le costanti `Nan` e `Infinity` le quali rappresentano rispettivamente un valore che non è un numero (derivante ad esempio da una conversione errata da stringa) e il valore infinito (derivante ad esempio da una divisione per 0).

Le costanti `boolean` sono definite tramite apposite stringhe `true` e `false`.

Inoltre sono definite anche le costanti `null` e `undefined` (la quale è restituita dalle procedure e rappresenta il valore iniziale delle variabili non inizializzate).

Le variabili in Javascript sono **loosely typed**, ovvero il linguaggio permette di assegnare alla stessa variabile prima un valore di un tipo e poi un valore avente un tipo differente. La dichiarazione delle variabili può essere implicita, nel caso la si utilizzi e basta (`x = 3`), od

[1] Da un lato consente di evitare cast dall'altro rende molto più difficile la ricerca di errori durante il debug

[2] Concettualmente è uno strumento molto potente, ma non è semplice da utilizzare per chi proviene da un linguaggi class based

[3] Questo provoca un drastico calo delle prestazioni, dunque è bene non abusare di tali operatori

esplicita, nel caso in cui si utilizzi la parola chiave `var` (es. `var x = 3`). Le variabili sono dotate di scope locale nel caso in cui la dichiarazione sia esplicita all'interno di una funzione, e globali in tutti gli altri casi. A differenza di Java, in Javascript un blocco non delimita uno scope, questo consente di riferire ad esempio variabili definite all'interno di uno scope interno.

**Esempio 9.1: variabili in Javascript**

```
x = '3' + 2   //contitene la stringa '32'
{
  {
    x = 5     //ora x contiene il numero 5
  }
  y = x + 3   //contiene il numero 8, x qui denota il valore 5 e non
      la stringa '32'
}
```

Il tipo delle variabili non è definito a priori, come anticipato. Mediante l'operatore `typeof` è possibile ottenere tale tipo dinamicamente.

**Esempio 9.2: tipi delle variabili in Javascript**

```
a=18;
typeof(a)                //number
a="ciao";
typeof(a)                //string
typeof(18/4)             //number
typeof("aaa")            //string
typeof(false)            //boolean
typeof(document)         //object
typeof(document.write)   //function
typeof([1,2,3])          //object, non array
```

Infine Javascript è dotato dei classici operatori relazionali e logici. A questo proposito occorre fare attenzione a come Javascript interpreta il valore falso ed i confronti tra variabili. Per Javascript sono considerati falsi: la costante `false` così come tutti i valori `falsy` (`null`, `undefined`, la stringa vuota `''`, il valore `0`, il valore `Nan`). Tutti gli altri oggetti (inclusa la stringa `'false'`) sono considerati veri. Gli operatori == e != in Javascript applicano la type coercion[4] secondo regole innaturali, portando talvolta a risultati inaspettati. Per questo motivo sono stati introdotti gli operatori === e !==, che offrono un'alternativa più naturale.

**Esempio 9.3: operatori di uguaglianza in Javascript**

```
0 == ''              //true, sono entrambi falsy values
0 == '0'             //true, 0 e' coercibile a '0'
false == 'false'     //false, come e' giusto che sia
false == '0'         //true, perche' sono due falsy
false == undefined   //false
false == null        //false
null == undefined    //true
' \t\r\n' == 0       //true...
```

[4]Ovvero cercano di convertire automaticamente un tipo in un altro

## 9.2 Lato funzionale

In Javascript le funzioni sono introdotte dalla keyword `function` e possono essere dotate di nome oppure no (andando di fatto a creare delle funzioni anonime analoghe alle lambda).

Le funzioni possono essere chiamate con l'operatore di chiamata `()` come in Java tuttavia, poiché la tipizzazione è dinamica, non viene fatto alcun controllo circa la compatibilità dei tipi all'atto della chiamata. Di conseguenza è possibile che a runtime le operazioni contenute all'interno della funzione non abbiano senso per i tipi specificati generando dunque un errore. Inoltre, contrariamente ad altri linguaggi, non è necessario che i parametri attuali corrispondano in numero ai parametri formali; in caso essi siano di più i parametri extra vengono ignorati, in caso contrario quelli mancanti sono `undefined`.

Le funzioni sono first class entities, dunque sono entità oggetto manipolabili come qualsiasi altro oggetto.

**Esempio 9.4**

```
//possono essere assegnate
var f = function(x){ return x / 10; }

//possono essere definite ed invocate al volo
var z = function(y){return y + 1;}(8); // z=9

//possono essere usate come argomento di un'altra funzione
var p = ff(f);

//possono essere restituite da una funzione
var f = fgen(...);
var z = f(3);
```

Javascript distingue tra **function expression** e **function declaration**. Nel primo caso il nome della funzione è superfluo, nel caso in cui venga specificato esso ha come scope il corpo della funzione stessa (utile per chiamate ricorsive). Nel secondo caso il nome è essenziale per poter chiamare la funzione, in quanto rappresenta l'unico modo per riferirsi ad essa. Lo scope di tale nome è lo scope della funzione stessa.

**Esempio 9.5: function expression vs function declaration**

```
//function expression
var f1 = function g(x){ return x / 10; }
g(32) // ERRORE: il nome "g" e' qui indefinito
f1(32) //OK

//il nome risulta utile per le chiamate ricorsive
var f2 = function h(x){ return (x == 0) ? 1 : h(x - 1); }

//function declaration
function g(x){ return x/10; }
g(32) // il nome g e' noto nell'ambiente
```

### 9.2.1 Chiusure

A differenza di quanto si può fare in linguaggi come C e Java in Javascript è possibile definire una funzione dentro un'altra funzione, generando di fatto delle chiusure nel caso in cui la

funzione innestata faccia riferimento a delle variabili contenute all'interno della funzione esterna. L'entità chiusura nasce a runtime all'atto della funzione "generatrice" esterna, ma sopravvive alla chiamata stessa in quanto occorre mantenere "vive" (accessibili) alcune variabili necessarie all'esecuzione della funzione generata.

**Esempio 9.6**

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

//ogni nuova invocazione di ff genera una chiusura
var r1 = ff( Math.sin, .8)(3)
var r2 = ff( function(q){return q * q}, .8 )(0.36)
```

La funzione `ff` è come una factory di funzioni, ad ogni invocazione crea una nuova funzione sulla base degli argomenti passatigli

Le chiusure in Javascript possono essere utilizzate per vari scopi.

- **Rappresentare uno stato privato interno**. In Javascript tutte le proprietà degli oggetti sono pubbliche, si può ottenere una proprietà privata tramite una chiusura, mappando lo stato interno su un argomento della funzione generatrice

  **Esempio 9.7**

  ```
  function genContatore(){
    var contati=0;
    function tick() { return contati++; }
    function num() { return contati;}
    //crea al volo un nuovo oggetto contenente quelle due funzioni
    return { num, tick };
  }

  var c = genContatore();
  document.writeln(c.num());   //0
  document.writeln(c.tick());  //0 poi 1
  document.writeln(c.num());   //1
  document.writeln(c.tick());  //1 poi 2
  document.writeln(c.tick());  //2 poi 3
  document.writeln(c.num());   //3
  ```

- **Realizzare un canale di comunicazione privato**. Si può ottenere un canale di comunicazione privato mettendo in una chiusura sia lo stato sia i due metodi accessor per poi restituire entrambi

  **Esempio 9.8**

  ```
  function myChannel() {
    var msg = "";
    function set(m) { msg = m;}
    function get() { return msg; }
    return { set, get };
  }
  ```

```
var ch = myChannel()
document.writeln(ch.set("hello")); //stampa undefined
document.writeln(ch.set("world")); //stampa undefined
document.writeln(ch.get());        //stampa world
```

Ad ogni invocazione di `myChannel` nasce un nuovo canale privato accessibile solamente ai due metodi restituiti dalla chiamata della funzione `myChannel` stessa

- **Realizzare nuove strutture di controllo**. Una funzione del secondo ordine incapsula il controllo mentre gli argomenti di tipo funzione rappresentano l'azione da svolgere.

**Esempio 9.9: costruzione di una struttura per i cicli**

```
function loop(statement, n) {
  var k = 0;
  return function iter() {
    if (k<n) { k++; statement(); iter(); }
  }
}

var i=1;

loop(function(){i++}, 10)();
```

In questo caso il nome della funzione (`iter`) risulta necessario per la chiamata ricorsiva in coda contenuta al suo interno. Ogni chiamata della funzione `loop` genera un nuovo ciclo che ripete lo specifico statement. Tuttavia si nota l'utilizzo delle parentesi tonde alla fine dell'invocazione di `loop` che rivelano che questo non è un costrutto built in

**Esempio 9.10: variante con currying**

```
function loop(n){
  return function statement(action) {
    if(n>0) {
      action(); n--; statement(action);
    }
  }
}

loop5 = loop(5);
loop5( function(){ document.writeln("ciao"); })

//oppure
//loop(5)( function(){ document.writeln("ciao"); })
```

Anche in questo caso l'uso della keyword `function` rivela però il fatto che `loop5` non è un vero costrutto nativo

### Chiusure e binding delle variabili

Va tenuto a mente che una chiusura corrisponde ad una istanza delle variabili. Questo può presentare problemi nel momento in cui, ad esempio, si sfrutta una variabile di chiusura per generare e restituire ciclicamente più funzioni, in quanto si rischia di mal interpretare il significato delle variabili.

**Esempio 9.11**

```
function fillFunctionsArray(myarray) {
  for (var i=4; i<7; i++) {
    myarray[i-4] = function(){ return i;};
  }
}

var myFunctions = [];
fillFunctionsArray(myFunctions);
for (j=0; j<myFunctions.length; j++) {
  write(myFunctions[j]() + ", ")
}
```

In questo caso l'idea iniziale è quella di creare un array di funzioni che restituiscano i valori `4,5,6`. Tuttavia, poiché c'è una sola variabile `i` nella chiusura, tutte le funzioni restituiscono il valore finale di `i`, ovvero `6`. Per catturare il valore di una variabile che cambia occorre fotografarlo in una variabile ausiliaria. Per farlo ci si può appoggiare ad una funzione intermedia il cui argomento faccia da variabile tampone per la variabile di chiusura che cambia.

```
function fillFunctionsArray(myarray){
  function aux(j){ return function(){ return j;}}
  for (var i=4; i<7; i++) {
    myarray[i-4] = aux(i);
  }
}

var myFunctions = [];
fillFunctionsArray(myFunctions);
for (j=0; j<myFunctions.length; j++) {
  write(myFunctions[j]() + ", ")
}
```

In questo caso, pur essendoci, un'unica variabile `i` all'interno della chiusura ci sono tante variabili `j` quante sono le funzioni ausiliarie. Ogni variabile `j` fotografa il valore che `i` assume in un dato momento

## 9.3 Lato ad oggetti

Javascript adotta un modello object based senza classi ma dotato della nozione di prototipo. In questo approccio un oggetto non è istanza di una classe, ma solo una collezione di proprietà pubbliche accessibili tramite dot notation. Gli oggetti sono costruiti tramite l'operatore `new` da un costruttore[5] che ne specifica la struttura iniziale e stabilisce l'associazione ad un oggetto padre, chiamato prototipo, da cui l'oggetto eredità le proprietà (si parla di **prototype based inheritance**).

Oltre all'utilizzo di un costruttore è possibile utilizzare una sintassi literal elencando le proprietà dell'oggetto.

**Esempio 9.12: costruzione degli oggetti**

```
//utilizzo di un costruttore
Point = function(i,j) {
```

[5]Il costruttore è tutto ciò che resta delle classi

```
  this.x = i; this.y = j;
  this.getX = function(){ return this.x; }
  this.getY = function(){ return this.y; }
}
//uso dell'operatore new
p1 = new Point(3,4)

//utilizzo della sintassi literal
p3 = { x: 10, y: 7,
       getX: function(){return this.x;}
       getY: function(){return this.y;}
     };
```

La parola chiave `this` usata all'interno del costruttore permette di specificare le proprietà ed i metodi dell'oggetto da creare e dunque di distinguerle dalle proprietà e dai metodi della funzione stessa.
Inoltre un costruttore si distingue da una funzione normale per la sua visibilità

```
function test1() {
  Point1 = function(i,j) {
    this.x = i; this.y = j;
  }
}

function test2() {
  function Point2(i, j) {
    this.x = i; this.y = j;
  }

p1 = new Point1(3,4)
println(p1)  //[object Object]

p2 = new Point2(3,4) //Exception: "Point2" not defined
```

Da questo secondo listato si nota che il costruttore `Point1` è globale, mentre la funzione `Point2` è locale alla funzione `test2`

### 9.3.1 Proprietà

Poiché tutte le proprietà sono pubbliche esse sono anche liberamente modificabili[6].

Le proprietà specificate dal costruttore non sono le uniche che un oggetto può avere. Infatti, a runtime, è possibile aggiungere proprietà dinamicamente semplicemente nominandole e usandole, oppure eliminarle dinamicamente tramite l'operatore `delete`.

**Esempio 9.13**

Riprendendo i `Point` discussi sopra

```
//da {x: 10, y: 4} diventa {x: 10, y: 4, z: -3}
p1.z = -3
//da {x: 10, y: 4, z: -3} diventa {y: 4, z: -3}
delete p1.x
```

[6] In realtà esistono delle proprietà di sistema non visibili né enumerabili dai normali costrutti. Questo viene fatto perché tali proprietà sono molto delicate e dunque un uso incauto potrebbe portare a dei problemi

Una delle proprietà condivise da tutti gli oggetti è `constructor`, la quale contiene il codice del costruttore. Va notato però che, poiché l'oggetto potrebbe essere cambiato nel frattempo, tale codice non è detto che rifletta lo stato attuale della risorsa, né che tutte le proprietà elencate siano realmente presenti o che siano le sole presenti.

I costruttori Javascript riflettono le "categorie" che in altri linguaggi sarebbero state rappresentate con classi, dunque le proprietà di classe possono essere modellate come proprietà del costruttore. Infatti poiché una funzione è allo stesso tempo un oggetto può essere dotata di proprietà. Dunque le proprietà comuni a tutti i `Point` possono essere contenute nell'unico oggetto che condividono, ossia proprio la funzione- costruttore omonima.

**Esempio 9.14**

Riprendendo i `Point` discussi sopra

```
p1 = new Point(3,4);
Point.color = "black";
Point.commonMethod = function(...){...}

p1.construcotr.color //"black"
```

In generale le proprietà degli oggetti sono pubbliche, l'unico modo per modellare delle proprietà private è l'utilizzo di chiusure con apposite funzioni accessor

**Esempio 9.15**

Riprendendo i `Point` discussi sopra

```
Point = function(i,j){
  this.getX = function(){ return i; }
  this.getY = function(){ return j; }
}

p1 = new Point(3,4);
x = p1.getX();    //restituisce 3
u = p1.x;         //undefined
```

Si sono difatti "chiuse" le variabili `i` e `j` all'interno delle funzioni.

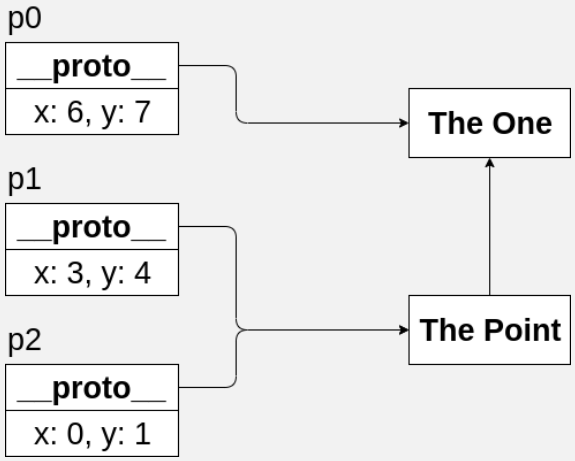
### 9.3.2 Prototipi di oggetti

Ogni oggetto è associato ad un oggetto padre, chiamato **prototipo**, da cui eredita le proprietà. Ciò dà luogo ad una forma particolare di ereditarietà, senza classi, detta *prototype-based inheritance*. In questo tipo di ereditarietà ogni oggetto referenzia il proprio prototipo mediante una proprietà nascosta chiamata `__proto__`.
Chi sia esattamente l'oggetto padre dipende da chi abbia costruito l'oggetto, tuttavia esiste un antenato comune a tutti, le cui proprietà sono comuni a tutti gli oggetti, lo chiameremo informalmente *The One*. Esso è un antenato diretto di tutti gli oggetti literal ed antenato indiretto di tutti gli altri oggetti.

**Esempio 9.16**

```
p0 = { x:6, y:7 };
p1 = new Point(3,4);
p2 = new Point(0,1);
```

Questo codice dà origine alle seguenti relazioni



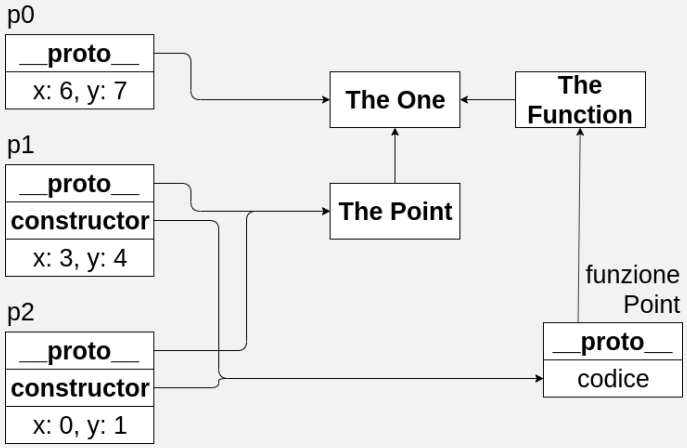

I prototipi sono il mezzo offerto da Javascript per esprimere le relazioni e le parentele tra oggetti. Esiste un prototipo predefinito per ogni categoria di oggetti (funzioni, array, numeri, Point, Persona, ...), tuttavia questi prototipi sono a loro volta degli oggetti, dunque hanno a loro volta un prototipo in "The One".

**Esempio 9.17**

Riprendendo l'esempio precedente

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= ... }
p0 = { x:6, y:7 };
p1 = new Point(3,4);
p2 = new Point(0,1);
```

Questo codice dà origine alle seguenti relazioni




In questo schema il padre delle funzioni (The function) è la funzione `function Empty(){}`. Mediante gli operatori `typeof` e `isPrototypeOf` è possibile risalire alla catena dei prototipi visualizzata nell'immagine

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

typeof(funz)                    //function
protfunz = funz.__proto__
typeof(protfunz)                //function
typeof(protfunz.__proto__)      //object
```

```
protfunz.isPrototypeOf(funz)    //true

pr0 = p0.__proto__
pr1 = p1.__proto__
pr2 = p2.__proto__
typeof(p1)                      //object
typeof(pr1)                     //object
pr1.isPrototypeOf(p1)           //true
p1.constructor                  //Point
pr1 == pr2                      //true
pr0 == pr2                      //false
pr1.__proto__ === pr0           //true
```

Il prototipo di un oggetto dipende da chi lo costruisce, ovvero gli oggetti costruiti da uno stesso costruttore condividono lo stesso prototipo. Il prototipo che un costruttore attribuisce agli oggetti costruiti è espresso dalla proprietà `prototype`, chiamata spesso *prototipo di costruzione*. Si noti che `__proto__` è una proprietà nascosta di tutti gli oggetti, mentre `prototype` è una proprietà pubblica dei soli costruttori.
È dunque grazie alla proprietà `prototype` che è possibile referenziare indirettamente gli oggetti predefiniti, infatti se `p` è un oggetto creato dal costruttore `Point` per definizione si ha

```
p.__proto__ == Point.prototype
```

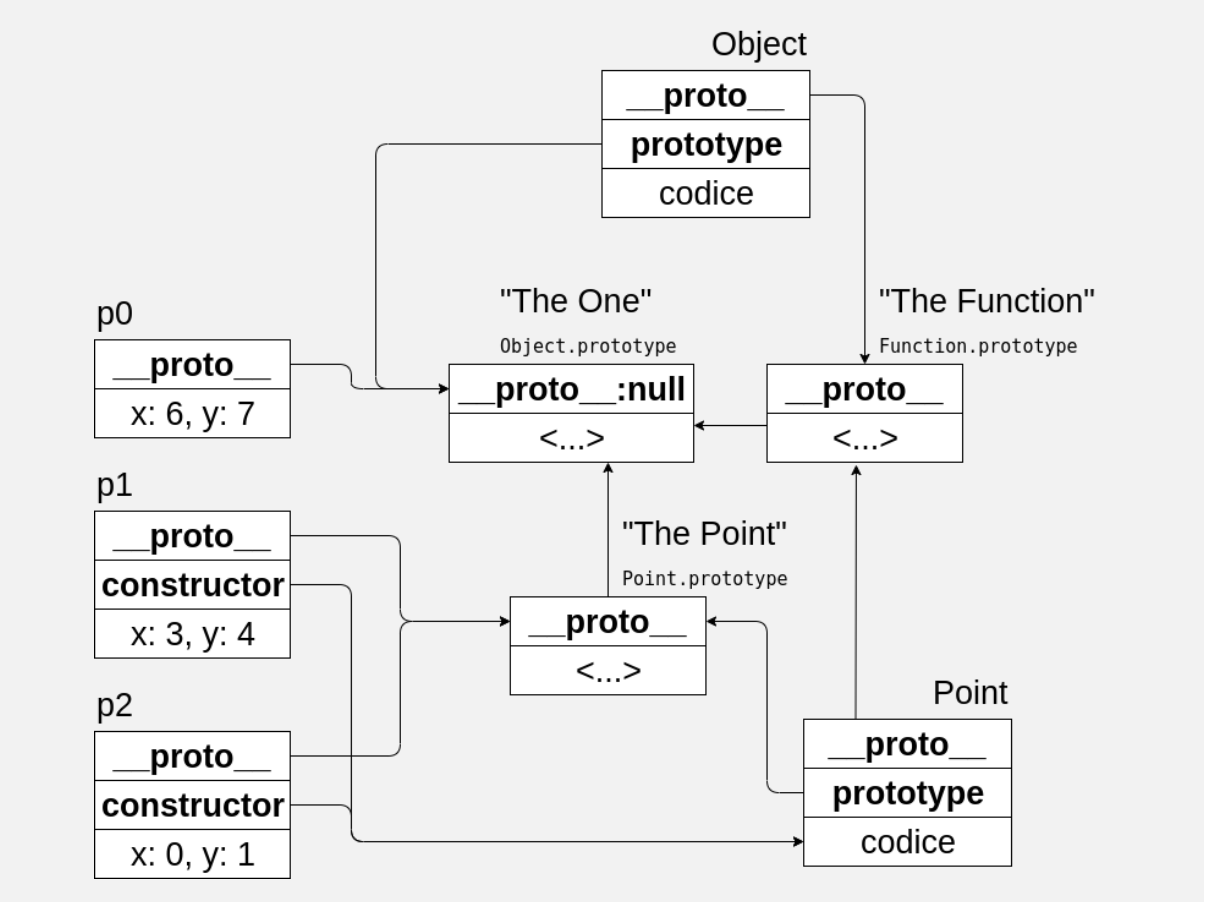
Detto `Object` il costruttore di tutti gli oggetti si ha che `Object.prototype` è "The One", ovvero l'antenato comune di tutta la tassonomia. Tuttavia, poiché `Object` in sé è un costruttore esso è in realtà una funzione, il cui valore `__proto__` punta a "The Function".

**Esempio 9.18**

Riprendendo l'esempio precedente

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= ... }
p0 = { x:6, y:7 };
p1 = new Point(3,4);
p2 = new Point(0,1);
```

Questo codice dà origine alle seguenti relazioni




### 9.3.3 Prototipi a runtime

Le relazioni tra oggetti espresse dalle catene di prototipi sono mantenute e reificate a runtime, di conseguenza sono dinamiche e possono essere manipolate a piacimento permettendo di propagare le modifiche all'intero sistema immediatamente. In particolare è possibile aggiungere o rimuovere proprietà ad un prototipo in uso o sostituire un prototipo con un altro.

Con **Type augmenting** si intende la possibilità di aggiungere proprietà al volo ad un prototipo esistente avendo un impatto immediato anche sugli oggetti già esistenti cambiando la struttura del sistema a runtime. Per farlo non è opportuno utilizzare la proprietà `__proto__` degli oggetti perché in generale essa non è pubblica e richiederebbe l'esistenza di un tale oggetto, ma è meglio utilizzare la proprietà `prototype` del costruttore (che punta al `_proto_` dei "suoi" oggetti), andando così ad impattare automaticamente su tutti gli oggetti costruiti.

**Esempio 9.19**

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= ... }
p0 = { x:6, y:7 };
showProps(p0);
p1 = new Point(3,4);
showProps(p1);
p2 = new Point(0,1);
showProps(p2);

Point.prototype.color = "black"
showProps(p1);  // ha anche la nuova proprieta' color!
showProps(p0);  // e' rimasto com'era
```

Aggiungendo la proprietà a `Point.prototype` la modifica si riverbera immediatamente ed automaticamente anche sugli oggetti già costruiti. Non impatta invece su `p0` in quanto fa riferimento ad `Object.prototype`

In alternativa è possibile sostituire il prototipo di costruzione con un altro. Questo non avrà effetto sugli oggetti già esistenti, ma solamente su quelli che verranno costruiti da quel momento in poi. È un modo per realizzare una differente versione dell'ereditarietà prototipale. In sostanza si procede ad agganciare la proprietà `prototype` del costruttore agganciandola ad un apposito oggetto/prototipo.

**Esempio 9.20**

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= ... }
p1 = new Point(3,4);
showProps(p1);

//ridefinizione del prototipo
Point.prototype = {
  color: "black", setX: function(x){ this.x = x; } };
}

showProps(p1); // e' rimasto tutto com'era
p3 = new Point(2,6);
showProps(p3); // ha anche le nuove proprieta'
p3.setX(9); // funziona e cambia l'ascissa a 9
```

Sostituendo il prototipo di default di `Point` tutti i `Point` futuri risulteranno differenti. Inoltre essi non risulteranno più imparentati con i vecchi `Point` in quanto il valore del loro campo `__proto__` sarà differente. Infatti il prototipo di `Point` non punterà più a "The Point", bensì al nuovo prototipo creato.

### 9.3.4 Ereditarietà prototype based

L'approccio precedentemente esposto può essere usato per definire relazioni di ereditarietà fra tipi propri. Di default ogni costruttore pone un diverso oggetto come prototipo degli oggetti costruiti, ma ciò porta a tante linee di parentela differenti e scollegate. Per avere relazioni del tipo padre-figlio occorre reimpostare i prototipi di costruzione in modo da riflettere la tassonomia desiderata[7].

**Esempio 9.21**

Tentiamo di esprimere il fatto che la categoria `Studente` erediti dalla categoria `Persona`. In primo luogo occorre definire costruttori per entrambe le categorie che contengano le proprietà desiderate, tuttavia, limitandosi a questo, le due categorie saranno comunque separate. Per ottenere il risultato desiderato occorre dunque che gli oggetti studenti abbiano come prototipo una `Persona` e non "The Student".
Quindi procediamo a:

1. Scegliere un oggetto-persona che faccia da prototipo per tutti gli studenti (scelto accuratamente)
2. Impostare su esso `Studente.prototype`, in modo che tutti gli studenti che verranno creati facciano riferimento ad esso

[7] L'ereditarietà fra classi dei linguaggi class- based diviene dunque un'ereditarietà prototipale espressa da `prototype`

3. Reimpostare la proprietà `constructor` dell'oggetto-persona prototipo (che di suo punterebbe al costruttore `Persona`) in modo che punti al costruttore `Studente`. Questo passo non è strettamente necessario, ma aiuta a mantenere la coerenza del sistema, infatti chiedendo ad uno studente chi sia il suo costruttore esso risponde così `Studente` e non genericamente `Persona`

```
//costruttore di Persona
Persona = function(nome, annoNascita) {
  this.nome = nome; this.anno = annoNascita;
  this.toString = function() {
    return this.nome + " e' nata nel " + this.anno
  }
}

//costruttore di Studente
Studente = function(nome, annoNascita, matricola) {
  this.nome = nome; this.anno = annoNascita;
  this.matricola = matricola;
  this.toString = function() {
    return this.nome + " e' nata nel " + this.anno + " e la sua
       matricola e' " + this.matricola
  }
}

//ora procuriamoci una Persona destinata a fungere da prototipo
protoStudente = new Persona("zio", 1900);

//impostiamo il prototipo di costruzione
Studente.prototype = protoStudente;

//per coerenza impostiamo la proprieta' constructor
Studente.prototype.constructor = Studente
```

Poiché Javascript non ha il concetto di classe non è tuttavia possibile riferirsi direttamente a quella che in linguaggi come Java sarebbe la "classe base". Tuttavia, utilizzando il metodo `call`, è possibile chiamare un oggetto funzione (svolgendo circa lo stesso compito) utilizzando il pattern:

```
funzione.call(target,...argomenti...)
```

```
Studente = function(nome, annoNascita, matricola) {
  Persona.call(this, nome, annoNascita);
  this.matricola = matricola;
  this.toString = function() {
    return Studente.prototype.toString.call(this) + " e la sua
       matricola e' " + this.matricola }
}
```

### 9.3.5 Ambiente globale e funzione `eval`

Javascript prevede la presenza di un ambiente globale in cui sono ospitate tutte le funzioni e tutte le variabili definite fuori da ogni altro scope. Anch'esso è un oggetto avente metodi e proprietà. I metodi di cui è dotato sono tutte le funzioni predefinite (inclusi i costruttori predefiniti, `Math`, `RegExp`, ...) e tutte le funzioni e le variabili definite dall'utente.

Va notato che non è prestabilito chi o cosa sia tale oggetto globale. Tipicamente in un browser esso corrisponde all'oggetto `window` (la finestra del browser). Solitamente questa informazione non è rilevante, tuttavia, nel caso in cui vengano utilizzate funzioni sensibili al contesto in cui vengono eseguite (come `eval`), questa informazione diventa vitale.

La funzione `eval` valuta una stringa ricevuta come parametro interpretandola come codice Javascript. Proprio a causa di questo comportamento il suo utilizzo è sconsigliato in modo da aumentare la sicurezza del codice. Tale funzione ha accesso in lettura e scrittura a tutti gli ambienti esterni ad essa, pertanto potrebbe essere utilizzata per portare a termine attacchi alla sicurezza.

## 9.4 Dove i due lati si incontrano

### 9.4.1 Costruttore `Function`

Come già detto, ogni funzione in Javascript è un oggetto invocabile che incapsula un comportamento. Negli esempi finora descritti sono sempre stati utilizzati dei *function literal*, ma in realtà è possibile costruire dinamicamente degli oggetti funzione tramite l'apposito costruttore `Function`[8] i cui parametri sono tutte stringhe (l'ultima il corpo della funzione, le altre i nomi dei parametri). Ciò permette di non cablare il comportamento delle funzioni all'interno del codice, permettendo dunque di creare nuove funzioni il cui comportamento è un parametro di ingresso.

**Esempio 9.22**

```
//function literal
square = function(x){ return x*x }

//uso del costruttore
square = new Function("x", "return x*x")
```

Questo approccio consente di rompere la tradizionale barriera che sussiste fra codice e dati, facendo sì che anche il codice possa essere un dato manipolabile[9].

**Esempio 9.23**

```
var funz = prompt("Scrivere f(x): ")
//si noti che qui viene forzato il tipo di x, nel caso generale x
   sarebbe una stringa e sarebbe trattata come una stringa anche all
   'interno della funzione
var x = parseInt("Calcolare per x = ? "))
var f = new Function("x", "return " + funz)
confirm("Risultato: " + f(x) )
```

Questo codice produce la seguente interazione:

```
Scrivere f(x):
> x * x - 1
Calcolare per x = ?
> 4
Risultato: 15
```

[8] È un oggetto funzione destinato alla creazione di altri oggetti funzione

[9] Ad esempio l'utente potrebbe scrivere il corpo della funzione da tastiera e la nuova funzione diventerebbe parte integrante del sistema come se fosse sempre esistita

Come tutte le funzioni anche `Function` possiede varie proprietà (tra cui `arguments` che contiene i parametri attuali, `arguments.length` che contiene il numero di parametri attuali, `caller` che contiene il chiamante o `null`) e metodi (come `valueOf()` che contiene la funzione stessa, `call` ed `apply` che supportano pattern di chiamata indiretta).

### 9.4.2 Pattern di chiamata indiretta

Una funzione tipicamente viene invocata per nome, in modo diretto, tramite l'operatore di chiamata (con i metodi assume la forma `object.method(args)`). Esistono però forme di chiamata indiretta in cui il target apparente è l'oggetto funzione stesso, mentre l'oggetto-target della chiamata è il primo argomento.

```
methodname.call(object, argsList)
methodname.apply(object, argsArray)
```

La differenza tra le due primitive sta solo nella forma con cui vengono specificati gli argomenti `args`, ma poiché in Javascript i parametri sono tutti opzionali, nel caso non ve ne siano, le due chiamate assumono la medesima forma.

**Esempio 9.24**

```
test = function(x, y, z){ return x + y + z }

test(3,4,5)
test.apply(obj, [3,4,5] ) //argomenti passati come array
test.call(obj, 3, 4, 5 )  //argomenti passati come lista
```

**Esempio 9.25**

```
prova = function(v){ return v + this.x }

//caso 1
x = 88
prova.call(this,3) //restituisce 3 + 88 = 91

//caso 2
x = 88
function Obj(u){
  this.x = u
}
obj = new Obj(-4)
prova.call(obj,3) //resituisce 3 + -4 = -1
```

### 9.4.3 Array Javascript

Un array Javascript è una di fatto una lista numerata i cui elementi si numerano a partire da 0. Al contrario di Java gli array possono contenere anche oggetti di tipo non omogeneo ed hanno lunghezza variabile[10]. Essi sono costruiti per mezzo del costruttore `Array` o per mezzo della sintassi literal `[...]`.

[10] Javascript consente di aggiungere dinamicamente elementi all'array

**Esempio 9.26: costruzione di array**

```
colori = new Array("rosso", "verde", "blu")
varie = ["ciao", 13, Math.sin]

//Aggiunta dinamica di un nuovo colore
colori[3] = "giallo"
```

Gli array costituiscono il supporto per gli oggetti in quanto, poiché ogni oggetto è caratterizzato da un insieme variabile di proprietà, un array è un modo pratico per supportarlo. In più gli oggetti sono array, di conseguenza la sintassi array-like (`obj[propname]`) è utilizzabile per accedere per nome alle proprietà degli oggetti. Questo risulta utile nel momento in cui si voglia accedere ad una proprietà senza cablarne il nome nel codice.

In particolare questo dà la possibilità di selezionare per nome delle funzioni da un elenco:

**Esempio 9.27**

```
var fname = prompt("Nome funzione?")
var f = listOfFunctions[fname]
```

La possibilità di aggiungere o togliere proprietà a un oggetto pone il problema di scoprire quali proprietà esso abbia in un dato istante. Una soluzione consiste nell'**introspezione**. Questo è realizzato, ad esempio, tramite il costrutto

```
for (variabile in oggetto) {...}
```

il quale itera sui nomi (e non sui valori) di tutte le proprietà[11].

**Esempio 9.28**

```
function show(ogg){
  for (var pName in ogg) {
    document.write("proprieta': " + pName +"<BR>")
  }
}
```

Per accedere alle proprietà per nome (ed eventualmente modificarle) ci si serve della notazione array-like, utilizzando dunque l'**intercessione**

**Esempio 9.29**

```
function show(ogg){
  for (var pName in ogg) {
    document.write("proprieta': " + pName + ", tipo " +
      typeof(ogg[pName]) + "<BR>")
  }
}
```

L'invocazione su due oggetti `Point` produce:

```
//p1
```

[11] Per gli array itera sui nomi degli indici

```
proprieta': x, tipo number
proprieta': y, tipo number
proprieta': z, tipo number
proprieta': getX, tipo function
proprieta': getY, tipo function

//p2
proprieta': x, tipo number
proprieta': y, tipo number
proprieta': getX, tipo function
proprieta': getY, tipo function
```

## 9.5 Applicazioni multiparadigma

Obiettivo di questa sezione è descrivere come si possano rendere interoperabili Java e Javascript, in modo da poter fornire supporto alla programmazione multi-linguaggio e multi-paradigma.

Storicamente si hanno varie alternative:

- 1997-2014 Rhino. Originariamente una costola del browser Netscape, in un primo tempo compilava il codice Javascript in bytecode, poi è passato ad interpretarlo direttamente. Il grosso punto debole erano le prestazioni
- 2014-2018 Nashrn (Java 8-11). Totalmente reingegnerizzato, permette di far girare nativamente il codice Javascript su JVM permettendo prestazioni nettamente migliori (comparabili con quelle del motore V8 di Google)
- 2018 Nashorn diviene deprecato con l'uscita di Java 11. La motivazione principale è che il team di sviluppo, molto piccolo, non riusciva a star dietro ai rapidi cambiamenti dello standard dietro a Javascript
- 2018+ GraalVM (`www.graalvm.org`). VM universale per l'esecuzione di più linguaggi, consente di eseguire in maniera efficiente altri linguaggi oltre a Javascript (come Ruby, R e Python). È in grado di girare sia come applicativo stand alone sia come pacchetto aggiuntivo in Java

### 9.5.1 Rhino

Rende possibile sia usare classi e package Java da codice Javascript che il viceversa.

La shell di Rhino definisce la variabile `Packages` per accedere ai pacchetti `java` e `com`, una variabile `java` (che equivale all'uso di `Packages.java`) e di una funzione `importPackages` analoga alla direttiva `import` di Java. Inoltre questo approccio consente di utilizzare anche classi Java custom all'interno del codice Javascript.

**Esempio 9.30**

```
js> fileName = "data.txt";
js> fw = new java.io.FileWriter(fileName);
js> fw.write("Hello World!");
js> fw.close();
js>
js> importPackage(java.io);
js> f = new BufferedReader(new FileReader(fileName));
js> while((line = f.readLine()) != null) line;
Hello World!
```

```
js>
```

È anche possibile che oggetti Javascript implementino interfacce Java.

**Esempio 9.31**

```
js> obj = { run: function(){ print("running...") }};
js> r = new java.lang.Runnable(obj);
js> t = new java.lang.Thread(r);
Thread[Thread-0,5,main]
js> t.start();
running..
```

L'istruzione alla seconda riga sarebbe errata in Java, tuttavia Rhino la interpreta correttamente come creazione di un wrapper per l'oggetto Javascript `obj`. Questo approccio consente però di implementare una singola interfaccia (od eventualmente di estendere una classe astratta). Pertanto si può procedere con un approccio differente.

```
JavaAdapter( javaInterfaceOrClass,
  [javaInterfaceOrClass, ...],
  javascriptObject)
```

In questo caso `javascriptObject` è l'oggetto Javascript che implementerà (o estenderà a seconda dei casi) le interfacce (o classi) con gli stessi meccanismi previsti da Java (l'ereditarietà multipla si ha solo per le interfacce)

È possibile inoltre includere Rhino all'interno di un programma Java. Per fare questo occorre un oggetto `Context` per mantenere le informazioni specifiche per quel thread, uno scope per mantenere le informazione specifiche per lo script (oggetto `Scope`), uno script (sotto forma di stringa o di `Reader`) e di un dominio di sicurezza in cui eseguire. L'esecuzione verrà poi lanciata dai metodi `evaluateString` o `evaluateReader` a seconda dei casi secondo la firma (identica per le due funzioni)

```
Object evaluateString(Scriptable scope,
   String source, String sourceName,
   int lineNumber, Object securityDomain )
```

**Esempio 9.32**

```
//associa al thread corrente un contesto
Context ctx = Context.enter();

//crea lo scope in cui eseguire lo script, necessario a mantenere
   variabili e funzioni
ScriptableObject scope = ctx.initStandardObjects();

//creazione dello script come stringa
String script = "java.lang.System.out.println('hello world!')";

//esegui lo script
Object result = ctx.evaluateString(scope, script, "Mia Prova", 1,
   null);

//stampa il risultato
```

```
System.out.println( ctx.toString(result) );

//rilascia il contesto
Context.exit();
```

### 9.5.2 Nashorn

Con Nashorm viene riprogettato il motore Javascript in modo da sfruttare le nuove istruzioni dinamiche aggiunte alla JVM. In più viene migliorata l'aderenza allo standard, e l'aderenza alle **Scripting API**. Viene infine migliorata l'interoperabilità con Java grazie all'introduzione di oggetti Javascript predefiniti, della funzione `Java.type` per accedere a qualunque tipo Java, una sintassi shortcut per manipolare liste e mappe Java e la possibilità di usare funzioni Javascript come lambda expression (sono convertite sottobanco in interfacce funzionali).

Tramite Nashorn risulta semplice accedere ai pacchetti Java, è possibile infatti utilizzare gli oggetti predefiniti (come `java.net.URL`) oppure utilizzare la funzione `Java.type('MyClass')` per ottenere il medesimo scopo[12]. Risulta inoltre semplice

- creare oggetti Java in Javascript tramite l'operatore `new` di Javascript
- accedere a metodi statici invocandoli direttamente
- accedere a proprietà di oggetti Java usando la notazione Javascript[13]

**Esempio 9.33**

```
jjs> var Thermo = Java.type('Thermo')
jjs>
jjs> var th1 = new Thermo()
jjs> th1
Thermo operating at [20.0-21.0]
jjs>
```

L'operatore `Java.type` fornisce un riferimento Javascript alla classe Java, che permette dunque alla `new` di istanziare il nuovo oggetto. La valutazione di `th1` viene tradotta nella chiamata alla `toString` sottostante

#### Integrazione tra Nashorn e Java

A proprosito dell'integrazione vanno tenuti a mente alcuni dettagli:

- Le stringhe Nashorn sono stringhe JavaScript (non Java), tuttavia esse vengono convertite automaticamente in stringhe Java se non c'è un metodo omonimo in Javascript. Più in generale, qualunque oggetto Javascript è convertito in una stringa Java se passato a un metodo Java che si aspetta una `String`
- I numeri Nashorn sono numeri JavaScript (non Java), dunque sono dei reali (non interi). La parte frazionaria è rimossa automaticamente se il valore è passato a un metodo Java che si aspetta un intero. Per motivi di efficienza Nashorn cerca comunque di mantenere la computazione fra interi quando possibile[14]

[12] Purché la classe indicata sia nel classpath
[13] Gli accessi vengono poi tradotti in opportune chiamate ai metodi setter e getter
[14] Anche se la differenza comunque spesso trascurabile

- Gli array Nashorn sono array JavaScript (non Java), quindi possono essere sparsi (possono esistere le celle 0 e 2, ma non la 1). Le conversioni tra i due tipi di array sono semplici da/verso array Java tramite `Java.from` e `Java.to`
- Nashorn fornisce una sintassi agevolata per liste e mappe Java. L'operatore JavaScript `[...]` può operare anche su liste e mappe Java, sia in lettura, la quale viene mappata sul metodo `get`, che in scrittura, la quale viene mappata sul metodo `set`. Risulta inoltre possibile iterare su esse: nel caso delle mappe si può iterare sia sulle chiavi (tramite il ciclo for) che sui valori (tramite il nuovo ciclo for each)
- Risulta semplice manipolare facilmente le proprietà di oggetti Java. È possibile utilizzare la notazione JavaScript per le proprietà (`obj.property`) anche su oggetti Java (vengono effettuate le opportune chiamate ai metodi setter/getter), oppure utilizzare la notazione JavaScript `obj["propname"]`
- Nashorn permette di intercettare e gestire eccezioni Java. Un metodo Java che possa lanciare eccezione può comparire in un try/catch Javascript (dove comunque è ammessa una sola clausola catch)
- Nashorn si integra bene col nuovo framework JavaFX, è infatti possibile lanciare direttamente uno script che contenga istruzioni JavaFX con `-fx`, inoltre la variabile d'ambiente `$STAGE` fornisce un riferimento allo Stage

**Esempio 9.34**

```
jjs> var myList = java.util.Arrays.asList("Enrico", "Giulio", "Ambra
   ")
jjs> myList[2]
Ambra
jjs> myList[1] = "Massimo"
jjs> myList
[Enrico, Massimo, Ambra]
jjs> myList[3] = "Roberta"
java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
jjs>
jjs> for (var x in myList) print(x)
0
1
2
jjs> for each (var x in myList) print(x)
Enrico
Massimo
Ambra
```

### Ereditarietà

Nashorn permette di estendere in JavaScript una classe Java e di implementare in JavaScript un'interfaccia Java tramite la funzione speciale `Java.extend`. Il tipo restituito è un'istanza del tipo passato come argomento, dunque le proprietà espsoste sono solamente quelle definite da questo tipo (eventuali proprietà aggiuntive sono usabili solo internamente).

**Esempio 9.35**

```
jjs> var it = Java.extend(java.util.Iterator,{
           next: function() Math.random(),
           hasNext: function() true
         }
     )
```

**Java e le SAM** Se l'interfaccia o la classe astratta da estendere hanno un singolo metodo (*Single Abstract Method*) è possibile specificarne l'implementazione inline senza dover precisare il nome del metodo.

**Esempio 9.36**

```
jjs> var MyRunnable = new java.lang.Runnable(
         function() {
           print('ciao')
         })
```

La creazione di un'istanza tramite la `new` è opportuna tuttavia solamente se si sta estendendo una classe astratta oppure un'interfaccia, in quanto in caso contrario si potrebbe avere ambiguità circa l'oggetto creato.

Di fatto quelle utilizzate sono delle lambda expression.

### Lambda Expression

Se un metodo Java si aspetta un'interfaccia funzionale è possibile passare una implementazione Javascript inline.

**Esempio 9.37**

```
java.util.Arrays.sort(array, function(x,y){ return ... } )
```

Nel caso particolare in cui il corpo della funzione sia una singola espressione, è possibile omettere le parentesi graffe e la keyword `return`, ottenendo così del codice particolarmente compatto e leggibile.

**Esempio 9.38**

```
var tF = Java.type('java.util.function.Function')
var f = new tF() { apply: function(x){ print(x+6) }}
f(5) // stampa 11
```

### Accedere a Nashorn da Java

Nashorn può essere invocato da Java tramite il meccanismo generale per gli scripting engine, JSR-223. Questo meccanismo (introdotto a partire da Java 6) consente di create un opportuno

`ScriptEngine` (interfaccia di `java.script`) e di invocare su di esso il metodo `eval` con gli opportuni argomenti [15] [16] [17].

Inoltre è possibile passare oggetti Java a Javascript tramite il metodo `put`. Di conseguenza è importante tenere a mente il mapping che sussiste tra gli oggetti Java e gli oggetti Javascript, ed è importante che le relative classi siano pubbliche (altrimenti le chiamate risulteranno `undefined`).

Si veda l'esempio seguente[18]

**Esempio 9.39**

```
import java.util.ArrayList;
import java.util.List;

import javax.script.*;

public class Main {
  public static void main(String[] args) {
    ScriptEngineManager myengineManager = new ScriptEngineManager();
    ScriptEngine engine =
      myengineManager.getEngineByName("nashorn");

    try {
      Object result = engine.eval("'Questa e' una stringa lunga ' +
        'Questa e' una stringa lunga lunga'.length");
      System.out.println(result);
    } catch (ScriptException e) {
      System.err.println(e);
    }

    try {
      List<Integer> list = new ArrayList<>();
      list.add(12);
      list.add(34);
      list.add(56);
      engine.put("myList", list);
      Object result = engine.eval("'La lista passata contiene ' +
        myList.length + ' elementi' ");
      System.out.println(result);
      // result == null
      Object result2 = engine.eval("for each (elm in myList)
                                      print(elm)");

      try {
        Invocable iengine = (Invocable) engine;
        // result3 == null
        Object result3 = iengine.invokeFunction("print",
                                                list.size());
      } catch (NoSuchMethodException e) {
        System.err.println(e);
      }
    } catch (ScriptException e) {
      System.err.println(e);
```

[15] Lo script può essere fornito sia come stringa sia come `Reader`

[16] Il risultato, se fornito, è un `Object`

[17] Il metodo può lanciare eccezioni

[18] I restanti esempi possono essere trovati nelle slide 142-145 del pacco di c19

```
      }
    }
  }
```

### 9.5.3 GraalVm

GraalVm è un progetto Oracle orientato alla *polyglot programming*. Fornisce un'apposita API, aderendo sempre alla Scripting Api.

Support l'interoperabilità con Java in maniera simile a Nashorn (spesso con una sintassi identica), tuttavia non permette l'accesso diretto alle classi Java per motivi di sicurezza, obbligando i programmatori a passare da `toJava.type()`. Inoltre è in grado di prevenire le conversioni di tipo che possono causare perdita di informazioni lanciando apposite eccezioni nel caso si verifichi questa evenienza.

**Esempio 9.40: Invocazione di Javascript da Java tramite Polyglot API**

```
import org.graalvm.polyglot.*;
...
Context context = Context.create();
Value array = context.eval("js", "[1,2,42,4]");
int result = array.getArrayElement(2).asInt();
```

**Esempio 9.41: Invocazione di Javascript da Java tramite Scripting API**

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine engine = m.getEngineByName("graal.js");
Object fn = engine.eval("(function() { return this; })");
Invocable inv = (Invocable) engine;
Object result = inv.invokeMethod(fn, "call", fn);
```

**Esempio 9.42: Utilizzo di classi Java da Javascript**

```
$ node --jvm
> var BigInteger = Java.type('java.math.BigInteger');
> console.log(BigInteger.valueOf(2).pow(100).toString(16));
10000000000000000000000000
```

**Esempio 9.43: Utilizzo di altri linguaggi da Javascript**

```
$ node --jvm --polyglot
> console.log(Polyglot.eval('R', 'runif(100)')[0]);
0.819835068437427
```

# Capitolo 10

# Lambda calcolo

Il lambda calcolo è un formalismo sviluppato da Church attorno agli anni trenta in grado di descrivere qualunque algoritmo. Tale formalismo è Turing- equivalente, e possiede come unico concetto quello di **funzione** e come unico paradigma quello di **applicazione di funzioni**.

Il lambda calcolo ha fornito le basi per tutti i linguaggi funzionali ed orientati agli oggetti del XX secolo.

Per comprendere cosa sia il lambda calcolo possiamo rifarci su di un esempio.

**Esempio 10.1**

Per definire matematicamente una funzione potremmo usare la seguente notazione:

$$cube : Int \to Int \text{ dove } cube(n) = n^3$$

A questo punto possiamo porci alcune domande:

- Qual è il valore dell'identificatore *cube*?
- Come possiamo rappresentare l'oggetto associato a *cube*?
- Può questa funzione essere definita senza un nome?

La stessa funzione può essere scritta usando la sintassi data da Church, tuttavia il lambda calcolo permette di definire solamente funzioni anonime:

$$\lambda n.n^3$$

Questa è una funzione anonima che mappa ogni $n$ del dominio in $n^3$.
Diciamo che l'espressione rappresentata da $\lambda n.n^3$ è il valore legato all'identificatore *cube*

**Lambda calcolo e linguaggi di programmazione** Si può notare che le funzioni utilizzate dal lambda calcolo sono simili alle funzioni anonime di Javascript. In questo linguaggio infatti abbiamo la sintassi:

```
f = function(x){return expr;}
        f(arg)
```

che costituiscono la sintassi per dichiarare una funzione anonima e per applicarla. Nel lambda calcolo il medesimo risultato è raggiunto tramite la sintassi:

$$\lambda x. < expr >$$
$$(\lambda x. < expr >)(arg)$$

In linguaggi come Java (8+) e C# la forma $\lambda$ è sostituita da operatori infissi come $\rightarrow$ (o $\Rightarrow$) ed i tipi degli argomenti possono essere sia espliciti che inferiti.

È inoltre possibile interpretare il lambda calcolo in chiave "object oriented".

- Definizione di metodi
  - $\lambda x.x$ può essere letta come `Object m1 (Object x) {return x;}`
  - $\lambda x.L$ può essere letta come `Object m2 (Object x) {return expr;}` dove $L$ è il lambda termine corrispondente ad `expr`
- Invocazione di metodi
  - $(\lambda x.x)y$ può essere letto come `m1(y)`, dove `m1` è il nome del metodo che contiene il codice corrispondente a $\lambda x.x$

## 10.1 Sintassi

La potenza del lambda calcolo sta nell'avere una sintassi con pochi elementi e dotata di una "semplice" semantica che tuttavia ha sufficiente espressività da poter rappresentare tutte le funzioni computabili (vedi tesi di Church- Turing).

Un lambda termine può essere:

- $x$, ovvero una variabile. Tipicamente questi identificatori vengono scritti con un singolo carattere minuscolo
- $\lambda x.L$, ovvero l'applicazione di una funzione anonima con un unico parametro (la variabile $x$) ed un corpo $L$ costituito da un lambda termine
- $LM$, ovvero l'applicazione di un termine $L$ ad un altro termine $M$

La sintassi del lambda calcolo può essere così rappresentata mediante sintassi BNF:

$$\begin{aligned} <termine> ::= & <variabile> \\ & | (<termine><termine>) \\ & | \lambda <variabile> . <termine> \\ & | (<termine>) \end{aligned}$$

La grammatica così definita è però ambigua. Prendiamo come esempio l'elemento $\lambda x.xy$. Occorre chiedersi come esso vada interpretato:

- come $(\lambda x.x)y$, ovvero come l'applicazione di $y$ alla funzione $\lambda x.x$
- come $\lambda x.(xy)$, ovvero come la definizione di una funzione con un singolo parametro $x$ e corpo $xy$, a sua volta interpretabile come applicazione di $y$ alla funzione $x$

Per evitare ambiguità è possibile utilizzare parentesi ed associatività degli operatori, tuttavia (esattamente come in matematica) vi sono delle regole per comprendere correttamente il significato di un'espressione.

- L'applicazione delle funzioni è associativa a sinistra
- I corpi delle funzioni sono estesi il più possibile, ovvero lo scope di una variabile si estende a destra quanto più possibile

Date queste regole si può vedere come l'applicazione di una funzione abbia precedenza sulle abstraction

**Esempio 10.2**

$yLxM$ va letto come $((yL)x)M$ poiché l'applicazione delle funzioni è associativa a sinistra

**Esempio 10.3**

$\lambda x.\lambda y.xyx$ va letto come $\lambda x.(\lambda y.xyx)$ (ovvero come $\lambda x.(\lambda y.(xy)x)$ considerando l'associatività dell'applicazione delle funzioni)

**Esempio 10.4**

Per l'espressione $(\lambda x.LM)N$ sono strettamente necessarie le parentesi nel caso in cui $N$ sia da intendere come argomento per la funzione $\lambda x.LM$ e non come parte del corpo di tale funzione

Data la semplicità della sintassi in alcuni casi può essere difficile interpretare correttamente un'espressione e inserire correttamente le parentesi.

**Esempio 10.5**

Consideriamo il termine

$$(\lambda n.\lambda f.\lambda x.f(nfx))(\lambda g.\lambda y.gy)$$

Per prima cosa occorre identificare le lambda abstraction che lo compongono:

- $(\lambda x.f(nfx))$
- $(\lambda f.(\lambda x.f(nfx)))$
- $(\lambda n.(\lambda f.(\lambda x.f(nfx))))$
- $(\lambda y.gy)$
- $(\lambda g.(\lambda y.gy))$

Infine occorre ricomporre i pezzi ricordando l'associatività sinistra

$$((\lambda n.(\lambda f.(\lambda x.(f((nf)x)))))(\lambda g.(\lambda y.(gy))))$$

## 10.2 Semantica

La semantica è definita tramite delle regole di trasformazione del tipo

$$(\lambda x.L)M \to L[M/x]$$

Questa computazione (chiamata **riduzione**) può essere scomposta in alcuni passi fondamentali:

1. Identificazione di un pattern del tipo $(\lambda x.L)M$ (ovvero l'applicazione al parametro $M$ della funzione $\lambda x.L$)
2. Sua trasformazione nel lambda termine $L[M/x]$ sostituendo nel corpo della funzione $L$ ogni occorrenza di $x$ con $M$

**Esempio 10.6**

L'espressione

$$x$$

non è chiaramente riducibile

**Esempio 10.7**

L'espressione

$$\lambda x.x$$

non è riducibile e rappresenta la funzione identità

**Esempio 10.8**

L'espressione

$$(\lambda x.x)y$$

si riduce ad $y$, infatti l'applicazione di questa funzione può essere vista come la sostituzione di tutte le occorrenze della variabile $x$ all'interno del corpo della funzione con il valore $y$

**Esempio 10.9**

L'espressione

$$(\lambda x.xx)(\lambda y.(\lambda z.yz))$$

si riduce ad $(\lambda y.(\lambda z.yz))(\lambda y.(\lambda z.yz))$, infatti la funzione $\lambda x.xx$ si occupa di duplicare l'argomento passatogli ottenendo il risultato già descritto

A questo punto si può riformulare il concetto di applicazione di una funzione ad un argomento come "valutarne" il corpo dopo aver sostituito al parametro formale il parametro attuale

## 10.3 Funzioni con più argomenti

Il lambda calcolo, come già detto, possiede come unico concetto quello delle funzioni ad un argomento. Di conseguenza ci si pone il problema di determinare come rappresentare funzioni aventi più argomenti. Una soluzione al problema è data dal *currying*, che permette di esprimere una funzione ad $n$ argomenti tramite $n$ funzioni ad un argomento innestate l'una dentro l'altra, in modo che ogni funzione ne restituisca un'altra parametrizzata come risultato. Mostriamo ora un caso con due argomenti (ma si possono fare osservazioni analoghe per il caso con tre o più argomenti).

**Esempio 10.10: currying con due argomenti**

La funzione

$$\lambda x.\lambda y.xy$$

si legge come

$$\lambda x.(\lambda y.xy)$$

ovvero una funzione in $x$ ($\lambda x.U$) il cui corpo $U$ è una funzione in $y$ ($\lambda y.xy$). Pertanto

$$(\lambda x.\lambda y.xy)LM \rightarrow (\lambda y.xy)[L/x]M = (\lambda y.Ly)M$$

A sua volta $(\lambda y.Ly)M$ è una funzione in $y$ in cui $y$ vale $M$. Pertanto

$$(\lambda y.Ly)M \to (Ly)[M/y] = LM$$

In definitiva

$$(\lambda x.\lambda y.xy)LM \to xy[L/x][M/y]$$

## 10.4 Funzioni notevoli

Alcune funzioni hanno un nome standard:

- $I ::= \lambda x.x$ è la funzione identità
- $K ::= \lambda x.\lambda y.x$ è un operatore di proiezione che dati due argomenti restituisce sempre il primo
- $K_I ::= \lambda x.\lambda y.y$ è un operatore di proiezione che dati due argomenti restituisce sempre il secondo
- $succ ::= \lambda n.\lambda z.\lambda s.s(nzs)$ permette di esprimere il successore di un numero naturale
- $\omega ::= \lambda x.xx$ è la funzione duplicatrice
- $\Omega ::= \omega\omega = (\lambda x.xx)(\lambda x.xx)$ è la funzione di loop

Si può osservare che la funzione $\Omega$ riproduce sempre se stessa.

## 10.5 Forme normalizzate

**Definizione 10.1: forma normalizzata**

Un lambda termine si dice in forma normale se non si possono applicare altre riduzioni

Si noti che questo fatto può dipendere dall'ordine con cui vengono applicate le riduzioni. A seconda delle sue caratteristiche un lambda termine si dice:

- **Fortemente normalizzabile** se qualunque sequenza di riduzioni porta ad una forma normale
- **Debolmente normalizzabile** se esiste almeno una sequenza di riduzioni che porta ad una forma normale
- **Non normalizzabile** se non si arriva mai ad una forma normale

**Esempio 10.11: forma fortemente normalizzabile**

Il termine

$$K\ 0\ (K\ 0\ 1)$$

è fortemente normalizzabile. Infatti nel caso si scegliesse di applicare prima la funzione esterna si otterrebbe il termine 0 non più riducibile. Nel caso si scegliesse di applicare prima la seconda funzione si otterrebbe prima la forma $K\ 0\ 0$ e poi applicando nuovamente la funzione $K$ si otterrebbe nuovamente il termine non riducibile 0

**Esempio 10.12: forma debolmente normalizzabile**

Il termine

$$K\ 0\ \Omega$$

è debolmente normalizzabile. Infatti nel caso si scegliesse di applicare prima la funzione $\Omega$ si entrerebbe in un loop infinito, mentre se si scegliesse di applicare la funzione $K$ per prima si otterrebbe il termine non riducibile 0

**Esempio 10.13: forma non normalizzabile**

Il termine

$$\Omega$$

non è normalizzabile. Infatti applicando la funzione $\Omega$ si otterrebbe una nuova funzione $\Omega$ in un ciclo infinito.

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx) = (\lambda x.xx)(\lambda x.xx) = \omega\omega = \Omega$$

**Teorema 10.1: di Church-Rosser**

Ogni lambda ha al più una forma normale. Essa può essere dunque interpretata come un risultato e dunque il lambda calcolo è **deterministico**

Il teorema appena enunciato afferma semplicemente che se esiste una forma normale essa è unica. Pertanto

- Nel caso il termine sia fortemente normalizzabile essa verrà sempre raggiunta. Si pensi a

```
int f(int x) { return x + 2; })
f(2)
```

- Nel caso sia debolmente normalizzabile il suo raggiungimento dipende dall'ordine delle riduzioni (è importante dunque scegliere la giusta strategia di valutazione)

```
int f(int x){ return x == 0 ? 0 : f(x); }
f(2)
```

- Nel caso il termine non sia riducibile tale forma non esiste e dunque non può essere raggiunta

```
int f(int x){ return f(x); }
f(2)
```

## 10.6 Computare con le lambda

Gli algoritmi descrivono funzioni che prendono in input una struttura dati e che, dopo un numero finito di passi, restituiscono un'altra struttura dati. Nel lambda-calcolo un primo lambda-termine $L$ descrive l'algoritmo, un altro lambda-termine $D$ descrive i dati di input. La computazione viene espressa mediante l'applicazione di $L$ a $D$ (ossia, scrivendo $LD$) e riducendo $LD$ "il più possibile". Il risultato è il nuovo lambda-termine $R$. I dati sono essi stessi dei termini-funzione opportunamente scelti (ossia, non riducibili).

**Esempio 10.14: i booleani di Church**

Supponiamo di voler rappresentare i valori booleani "true" e "false" e gli operatori logici per mezzo del lambda calcolo. Poiché l'unico tipo di dato è la funzione, sia i dati (i valori vero e falso) sia gli algoritmi (gli operatori logici) saranno delle funzioni. A questo proposito dobbiamo garantire che:

- $NOT(true) = false$
- $NOT(false) = true$
- $AND(true, x) = x$
- $AND(false, x) = false$
- $OR(true, x) = true$
- $OR(false, x) = x$

Diamo dunque la rappresentazione dei dati e degli operatori logici:

- Il valore "vero" può essere rappresentato mediante una funzione (chiamiamola $T$) di proiezione:
$$\lambda x.\lambda y.x$$
essa restituisce il primo dei due argomenti
- Il valore "falso" può essere rappresentato mediante una funzione (chiamiamola $F$) di proiezione:
$$\lambda x.\lambda y.y$$
essa restituisce il secondo dei due argomenti

Si può notare come il funzionamento dei valori $T$ ed $F$ ricordi molto da vicino il funzionamento di un operatore ternario. Nel caso il booleano sia vero viene restituito il primo valore, in caso contrario viene restituito il secondo.
Su questi booleani possiamo definire i seguenti operatori:

- L'operatore $NOT$ può essere definito come:
$$\lambda x.xFT$$
ovvero una funzione che restituisce il primo valore (ossia $F$) nel caso in cui l'ingresso sia $T$, mentre restituisce il secondo valore ($T$) nel caso in cui l'ingresso sia $F$. Vediamo le sequenze di riduzione
$$NOT\ T = (\lambda x.xFT)T \to TFT \to (\lambda x.\lambda y.x)FT \to (\lambda y.F)T \to F$$
$$NOT\ F = (\lambda x.xFT)F \to FFT \to (\lambda x.\lambda y.y)FT \to (\lambda y.y)T \to T$$
- L'operatore $AND$ può essere definito come:
$$\lambda x.\lambda y.xyF$$
ovvero una funzione che nel caso in cui $x$ sia $T$ restituisce il secondo valore (qualunque esso sia) e che restituisce $F$ nel caso contrario (valutando l'operatore in corto circuito). Vediamo le sequenze di riduzione
$$AND\ T\ X = (\lambda x.\lambda y.xyF)TX \to (\lambda y.TyF)X \to TXF \to X$$
$$AND\ F\ X = (\lambda x.\lambda y.xyF)FX \to (\lambda y.FyF)X \to FXF \to F$$

- L'operatore $OR$ può essere definito come:

  $$\lambda x.\lambda y.xTy$$

  ovvero una funzione che nel caso in cui $x$ sia $T$ restituisce $T$ (valutando l'operatore in corto circuito) e che restituisce il secondo valore (qualunque esso sia) nel caso contrario. Vediamo le sequenze di riduzione

$$OR\ T\ X = (\lambda x.\lambda y.xTy)TX \rightarrow (\lambda y.TTy)X \rightarrow TTX \rightarrow T$$

$$OR\ F\ X = (\lambda x.\lambda y.xTy)FX \rightarrow (\lambda y.FTy)X \rightarrow FTX \rightarrow X$$

È importante notare che questa non è l'unica rappresentazione possibile, ma che altre sono possibili. Inoltre occorre tenere a mente che questa è solamente una rappresentazione dei dati e delle operazioni, sebbene possa apparire strana nel caso in cui non si sia abituati ad essa (anche le costanti sono delle funzioni). Per quanto strana possa apparire però è sufficiente pensare che lo stesso tipo di operazione, ovvero la definizione dei tipi di dato, viene effettuata in un qualsiasi linguaggio di programmazione.
Utilizzando una notazione javascript si può adottare la seguente rappresentazione alternativa (forse più familiare):

- True: $\lambda x.\lambda y.x$

```
T = x => y => x
```

- False: $\lambda x.\lambda y.y$

```
F = x => y => y
```

- NOT: $\lambda x.xFT$

```
NOT = x => x(F)(T)
```

- AND: $\lambda x.\lambda y.xyF$

```
AND = x => y =>  x(y)(F)
```

- OR: $\lambda x.\lambda y.xTy$

```
OR = x => y =>  x(T)(y)
```

Si può notare anche qui che le lambda seguono la filosofia "tutto è funzione"

### 10.6.1 Strategie di riduzione

Come già detto (vedi 10.6) nel lambda calcolo alcuni termini ($L$) rappresentano gli algoritmi, altri ($D$) i dati di input. Il risultato $R$ viene ottenuto nel momento in cui non sono più possibili altre riduzioni dei vari lambda termini. Il concetto di "riduzione massima" introduce il concetto di **strategia di riduzione**, infatti a seconda della strategia scelta alcune computazioni potrebbero terminare (portando ad un risultato) o non terminare (non consegnando alcun risultato).

Vi sono due principi ispiratori alternativi:

- **Valutazione eager**: viene privilegiata la valutazione dell'argomento rispetto all'applicazione dell'argomento alla funzione, ovvero gli argomenti vengono valutati il prima possibile (*strict evaluation*). Questo tipo di valutazione è alla base di alcune strategie trovate nei linguaggi di programmazione:
  - **Applicative order**: *rightmost innermost*, prima riduce e valuta gli argomenti poi li applica alla funzione, pertanto potrebbe non trovare la forma normale sebbene essa esista (non è una strategia normalizzante).
  - **Call by value**: è la famiglia di strategie più utilizzata nei linguaggi di programmazione (i componenti della famiglia differiscono per l'ordine con cui sono valutati gli argomenti della funzione, ad esempio da destra a sinistra...). Questa soluzione è simile alla precedente ma non riduce i termini dentro la funzione prima di applicare gli argomenti e, poiché non passa funzioni ancora da calcolare, considera come corpo **solo** lambda termini della forma

    $$E ::= x \mid xE \mid (\lambda x.L)$$

    escludendo quindi quelli del tipo $\lambda z.z$
  - **Call by reference**: la funzione riceve un riferimento implicito all'argomento del chiamante anziché una copia dello stesso. È presente in vari linguaggi di programmazione, ma raramente è la forma di default. Si noti che questa strategia non equivale al passaggio di un puntatore in call by value, poiché in tal caso il riferimento è passato esplicitamente
  - **Call by copy-restore**: è un caso speciale della precedente adatta al multi-threading. In questo caso ogni funzione riceve una copia privata dell'argomento ed al termine dell'esecuzione tale argomento viene copiato indietro. Questo sistema evita le interferenze, tuttavia solo il valore computato dall'ultimo thread sopravvive come risultato finale
- **Valutazione lazy**: viene privilegiata l'applicazione dell'argomento alla funzione rispetto alla valutazione dell'argomento stesso, ovvero gli argomenti vengono valutati il più tardi possibile (solamente quando serve, *non strict evaluation*). Questo tipo di valutazione è alla base di alcune strategie trovate nei linguaggi di programmazione:
  - **Normal order**: *leftmost outermost*, applica gli argomenti alle funzioni prima di ridurli, riducendo per prima la lambda più a sinistra e considerando come corpo ciò che compare a destra. Trova sempre la forma normale nel caso esista (da ciò deriva il nome "normal"), tuttavia questo procedimento può causare la creazione di copie multiple dello stesso termine causando inefficienza (risolta dalla call by need)
  - **Call by name**: funziona similmente al "normal order", tuttavia considera irriducibili le "lambda abstraction" (come $\lambda x.x$). Ad esempio $\lambda x.(\lambda x.x)x$ è irriducibile con questa strategia, mentre in normal order verrebbe ridotto a $\lambda x.x$, di fatto dunque non porta le sostituzioni fin dentro le funzioni interne
  - **Call by need**: simile alla call by name, tuttavia evita valutazioni multiple dello stesso termine memorizzandone il risultato (è la forma più usata di valutazione lazy)
  - **Call by macro**: simile alla call by name ma basata su sostituzioni testuali, per questo motivo richiede particolare attenzione ai nomi delle macro in modo da evitare effetti spuri
  - **Call by future**: variante della call by need in cui gli argomenti sono valutati in parallelo alla funzione stessa (dunque non solo quando vengono usati) ed i thread si sincronizzano solo nel momento in cui l'argomento serve

– **Valutazione ottimistica**: variante della call by need in cui gli argomenti rimangono parzialmente valutati per un determinato periodo di tempo, allo scadere del quale scatta la call by need normale. Con questa variante è possibile ridurre il costo a runtime

**Esempio 10.15: riduzione**

Il termine

$$(\lambda x.\lambda y.x)p((\lambda u.v)u)$$

è fortemente normalizzabile. In normal order si ottiene la seguente sequenza di riduzioni:

$$\begin{aligned}(\lambda x.\lambda y.x)p((\lambda u.v)u) &\to (\lambda y.x)[p/x]((\lambda u.v)u)\\ &\to (\lambda y.p)((\lambda u.v)u) \to p[((\lambda u.v)u)/y] \to p\end{aligned}$$

In applicative order invece si ottiene la seguente sequenza di riduzioni:

$$\begin{aligned}(\lambda x.\lambda y.x)p((\lambda u.v)u) &\to (\lambda x.\lambda y.x)pv[u/u] \to (\lambda x.\lambda y.x)pv\\ &\to (\lambda y.x)[p/x]v \to (\lambda y.p)v \to p[v/y] \to p\end{aligned}$$

**Esempio 10.16: riduzione**

Il termine

$$(\lambda x.a)((\lambda x.xx)(\lambda y.yy)) = (\lambda x.a)\Omega$$

è debolmente normalizzabile. In normal order si ottiene la seguente sequenza di riduzioni:

$$(\lambda x.a)\Omega \to a[\Omega/x] \to a$$

In applicative order invece si ottiene la seguente sequenza di riduzioni:

$$\begin{aligned}(\lambda x.a)((\lambda x.xx)(\lambda y.yy)) &\to (\lambda x.a)(xx)[(\lambda y.yy)/x]\\ &\to (\lambda x.a)((\lambda y.yy)(\lambda y.yy)) = (\lambda x.a)\Omega\end{aligned}$$

In questo caso si ha una riproduzione infinita

**Esempio 10.17: riduzione**

Il termine

$$\lambda x.(\lambda x.x)x$$

è debolmente normalizzabile. In normal order e con la call by value si riduce a:

$$\lambda x.x$$

Con la call by name è invece irriducibile

## 10.7 Turing-equivalenza

Perché un formalismo sia Turing equivalente esso deve poter rappresentare:

- I numeri naturali
- Il successore di un numero naturale
- Il concetto di proiezione

- Il concetto di composizione
- La ricorsione

Poiché il lambda calcolo è in grado di rappresentare tutti questi concetti esso **è** Turing equivalente. Alcune rappresentazioni risultano semplici, mentre altre risultano particolarmente complesse. In particolare, poiché le funzioni sono anonime, risulta particolarmente complesso rappresentare il concetto di ricorsione.

**Numeri naturali e successore** È possibile dare la seguente rappresentazione dei numeri naturali (numeri naturali di Church):

- $0 = \lambda z.\lambda s.z$, ovvero una funzione a due argomenti che funge da base
- $1 = \lambda z.\lambda s.sz$, ovvero una funzione a due argomenti che aggiunge una $s$ in testa
- $2 = \lambda z.\lambda s.s(sz)$, ovvero una funzione a due argomenti che aggiunge un'altra $s$ in testa
- $3 = \lambda z.\lambda s.s(s(sz))$, ovvero una funzione a due argomenti che aggiunge una terza $s$ in testa

È importante notare che tali funzioni non vengono calcolate, ma sono esse stesse i risultati.

Mediante questa rappresentazione dei numeri naturali è possibile anche dare una definizione per quanto riguarda il concetto di successore di un numero naturale. Esso è la funzione

$$Succ\ n = \lambda n.\lambda z.\lambda s.s(nzs)$$

**Esempio 10.18: successore di $0$**

$$Succ\ 0 \to \lambda z.\lambda s.s(0zs) \to \lambda z.\lambda s.s((\lambda z.\lambda s.z)zs) \to$$
$$\to \lambda z.\lambda s.s((\lambda s.z)s) \to \lambda z.\lambda s.s(z) \to 1$$

**Esempio 10.19: successore di $1$**

$$Succ\ 1 \to \lambda z.\lambda s.s(1zs) \to \lambda z.\lambda s.s((\lambda z.\lambda s.sz)zs) \to$$
$$\to \lambda z.\lambda s.s((\lambda s.sz)s) \to \lambda z.\lambda s.s(sz) \to 2$$

Inoltre si possono definire anche altre funzioni da utilizzare con i numeri naturali. Qui è riportato l'esempio della somma tra numeri naturali data dalla funzione

$$Sum = \lambda m.\lambda n.\lambda z.\lambda s.n(mzs)s$$

**Esempio 10.20: somma**

$$Sum\ 1\ 1 \to \lambda z.\lambda s.1(1zs)s \to \lambda z.\lambda s.1(sz)s \to$$
$$\to \lambda z.\lambda s.(\lambda u.\lambda v.vu)(sz)s \to \lambda z.\lambda s.s(sz) \to 2$$

Si noti come, anche in questo caso, la funzione "2" non venga utilizzata per calcolare il risultato, ma essa stessa sia il risultato

**Proiezione** Come già anticipato la proiezione è facilmente ottenibile mediante un'apposita funzione

$$K ::= \lambda x.\lambda y.x$$

**Composizione** La composizione può essere ottenuta con molta semplicità tenendo conto dell'associatività dell'applicazione delle funzioni nel seguente modo:

$$\lambda f.\lambda g.\lambda x.f(gx)$$

**Ricorsione** Per esprimere la ricorsione servono due ingredienti principali: la capacità di definire funzioni (il lambda calcolo ha solamente funzioni come tipo di dato) e la capacità di chiamare funzioni per nome. Questo secondo ingrediente presenta un grosso problema, infatti le funzioni lambda sono anonime, dunque serve un metodo alternativo per eseguire il corpo di una funzione e, al contempo, per rigenerare il corpo della funzione stessa. A questo provvede il combinatore di punto fisso $Y$.

**Definizione 10.2: punto fisso**

In matematica $x$ è un punto fisso per una funzione $f(x)$ se

$$x = f(x)$$

ovvero se la funzione mappa il valore $x$ in se stesso

**Definizione 10.3: combinatore di punto fisso Y**

Il combinatore di punto fisso è definito come

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Applicando questo operatore ad una funzione $F$ si ottiene il seguente effetto

$$YF \to F(YF)$$

cioè si ottengono esattamente i due effetti richiesti precedentemente, ossia eseguire il corpo della funzione ($F(...)$) e rigenerare la funzione stessa ($YF$). L'entità $YF$ è un punto fisso per la funzione $F$, più nello specifico si ottiene la seguente sequenza di riduzioni[1]

$$\begin{aligned} YF =& \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F \\ & \to (\lambda x.F(xx))(\lambda x.F(xx)) \\ & \to F((\lambda x.F(xx))(\lambda x.F(xx))) \to F(YF) \end{aligned}$$

Per comprenderne meglio il funzionamento conviene tradurne in codice Javascript il comportamento.

```
function Y(f) {
  return (
    //la chiamata x(x) diverge se valutata con call by value in quanto
       per valutare la funzione occorrerebbe valutare prima il valore di
       x
    (function(x) {return f( x(x) ); })
    (function(x) {return f( x(x) ); })
  );
}
```

[1] il combinatore $Y$ presuppone la strategia call by name: se usato con strategia call by value diverge perché il lambda termine $(xx)$ presuppone di applicare la funzione $x$ a se stessa

Tuttavia per usare questo approccio in un modello computazionale basato su call by value occorre simulare la call by name nel solito modo, ovvero introducendo una funzione intermedia ausiliaria.

Il combinatore $Y$ modificato per funzionare con la call by value viene anche chiamato combinatore $Z$.

**Definizione 10.4: combinatore Z**

Il combinatore $Z$ è definito come

$$Z = \lambda f.(\lambda x.f(\lambda v.((xx)v)))(\lambda x.f(\lambda v.((xx)v)))$$

Utilizzando questa forma, esso può essere implementato in qualunque linguaggio che abbia gli oggetti come entità di prima classe (come Java, Javascript, C#...). Tradotto in Javascript esso diverrebbe:

```
function Z(f) {
  return (
    (function (x) { return f( function(v){ return x(x)(v); } ); })
    (function (x) { return f( function(v){ return x(x)(v); } ); })
  );
}
```

Il combinatore $Y$ esploderebbe subito poiché avrebbe solo `x(x)`.

A questo punto è possibile creare algoritmi ricorsivi mediante l'utilizzo di tali operatori. Per prima cosa occorre eliminare la ricorsione esplicita sostituendola con una implicita. Per farlo occorre:

1. Trovare una $F$ di ordine superiore il cui punto fisso sia la funzione ricorsiva $f$ desiderata
2. Estrarre dalla funzione ricorsiva $f$ il passo-chiave, incapsulandolo in una nuova funzione *step* non ricorsiva
3. Delegare al combinatore $Y$ la "composizione" dei passi di step, tramite la scrittura $Y$ *step*
4. La "chiamata" di funzione sarà quindi $(Y\ step)(args)$

Riportiamo qui l'esempio del calcolo del fattoriale di un numero mediante l'utilizzo dell'operatore di punto fisso[2]

**Esempio 10.21: calcolo del fattoriale**

Se il lambda calcolo permettesse di dare un nome alle funzioni tale calcolo sarebbe semplice, tuttavia le funzioni sono anonime, pertanto introduciamo una funzione $F$ di ordine superiore che riceva $f$ come argomento

$$F := \lambda f.\lambda x.((x\ ==\ 0)\ ?\ 1\ :\ x\ *\ f(\ x\ -\ 1\ ))$$

Con questa funzione di ordine superiore la computazione effettiva del fattoriale si esprime come

$$(YF)(args)$$

ossia in questo caso

$$(YF)(n)$$

[2]Per altri esempi consultare le slide 47-61 del pacco c20

Passiamo al calcolo di 3!

$$
\begin{aligned}
3! = (YF)(3) = F(YF)(3) = \\
\lambda f.\lambda x.((\ x == 0\ )\ ?\ 1\ :\ x\ *\ f(x-1))(YF)(3) = \\
\lambda x.((\ x == 0\ )\ ?\ 1\ :\ x\ *\ (YF)(x-1))(3) = \\
(\ 3 == 0\ )\ ?\ 1\ :\ 3\ *\ (YF)(2) = \\
3\ *\ (2 == 0)\ ?\ 1\ :\ 2\ *\ (YF)(1) = \\
3\ *\ 2\ *\ (1 == 0)\ ?\ 1\ :\ 1\ *\ (YF)(0) = \\
3\ *\ 2\ *\ 1\ *\ (0 == 0)\ ?\ 1\ :\ ... = \\
3\ *\ 2\ *\ 1\ *\ 1 = 6
\end{aligned}
$$

Questo esempio può essere tradotto in codice Javascript come segue

```
FactGen = function(f){
  return function(x) {
    return (x==0) ? 1: x * f(x - 1);
  }
}

//in questo caso utilizziamo il combinatore Z in quanto Javascript
   utilizza la strategia call by value
var fact = Z(FactGen)

3! = fact(3) = Z(FactGen)(3) = ... = 6
```

## 10.8 Utilizzi del lambda calcolo

Il lambda calcolo è stato utilizzato per definire le proprietà dei linguaggi di programmazione, come sorgente di ispirazione per i linguaggi funzionali e come base per studiare i concetti dei sistemi di typing. Naturalmente però non è destinato direttamente agli utenti finali. Gli utenti godono dei vantaggi che ha comportato senza saperlo.

# Capitolo 11

# Scala

Scala è un linguaggio scalabile (aggettivo da cui deriva il nome) Java based. Tale linguaggio è stato progettato per poter creare agilmente sia piccoli script che grandi sistemi e, poiché è compilato in bytecode, è completamente integrato con il mondo Java e può dunque usarne le librerie. Tuttavia Scala è anche un linguaggio blended che unisce aspetti object-oriented (con pieno supporto ad uno stile di programmazione imperativo) ad aspetti funzionali (possiede funzioni come first-class entities, chiusure, ricorsione in coda ottimizzata e netta distinzione tra valori e variabili). Lo stile di codifica tipicamente è più conciso di quello di Java.

Scala possiede varie caratteristiche, riassumibili come:

- Tutto è un oggetto (anche gli ex tipi primitivi)
- Ogni funzione è un oggetto, istanza di un certo tipo-funzione
- Ogni metodo è un operatore e ogni operatore è un metodo
- Supporto a valori immodificabili e operazioni senza side effect
- Strutture di controllo che denotano valori
- Mix-in composition fra oggetti e classi (risolve ereditarietà multipla)
- Type inference sofisticata
- Shortcut linguistici e sintassi infisse gestite automaticamente

Possiede forti influenze da linguaggi come Java e C# (modello di esecuzione, espressioni, istruzioni, blocchi, classi, package, tipi, librerie) e linguaggi come Smalltalk, Ruby e Haskell. Tuttavia ha portato anche alcuni contributi originali come l'introduzione dei tipi astratti (come alternativa OO ai tipi generici), tratti (come evoluzione delle interfacce) ed extractor (per estrarre proprietà e per il pattern matching).

## 11.1 Concetti di base

### Tipi

In primo luogo Scala non possiede tipi primitivi, essi infatti sono sostituiti da appositi tipi-valore rimappati a livello di bytecode sui tipi primitivi di Java per motivi di efficienza. Alla radice della gerarchia dei tipi vi è `Any` e non `Object`, fatto che permette di distinguere immediatamente tra tipi-valore (derivanti da `AnyVal`) e tipi- riferimento (derivanti da `AnyRef`).

**Esempio 11.1: assegnamento multiplo ed il tipo Unit**

Scala non supporta volutamente l'assegnamento multiplo. A tal fine l'assegnamento è un'espressione di tipo `Unit` che restituisce la costante `()` (che **non** è la chiamata di funzione). In Java si ha

```
(b=18) == 18          //true
                      //assegna 18 a b. L'espressione tra parentesi
                         ha come valore complessivo 18

(s=readLine())!=null // s != null
```

In Scala si ha

```
(b=18) == 18          //sempre false
                      //assegna 18 a b, ma l'espressione tra
                         parentesi ha come valore Unit ()

(s=readLine())!=null // sempre true
```

**Esempio 11.2: array**

In Java gli array sono dotati di una sintassi derivante dal C basata sull'uso dell'operatore `[]` sia per la definizione del tipo sia per l'accesso alle celle, inoltre non sono introdotti da alcuna parola chiave specifica, causando difformità rispetto alle altre collezioni. Inoltre gli array Java sono covarianti mentre tutte la altre collezioni sono invarianti.
In Scala invece la sintassi degli array è uniforme a quella delle altre collections (parola chiave `Array`). L'accesso agli elementi è effettuato tramite l'operatore `()`, mentre la parametrizzazione in tipo avviene tramite le parentesi quadre.

```
val v = new Array[String](3);
v(0) = "Paperino";
v(1) = "Pippo";
v(2) = "Pluto";
```

### Valori e variabili

Scala fa distinzione tra *valori* (immodificabili) e *variabili* (modificabili) introducendo l'utilizzo di due parole chiave distinte per identificarle (`val` e `var`). Nonostante `val` sia immodificabile, rimane pur sempre una variabile, perciò può cambiare valore se re-instanziata durante l'esecuzione. Se si vuole invece una vera e propria costante che mantenga sempre lo stesso valore nel tempo, la si può dichiarare con il nome in `CamelCase`. Inoltre la specifica del tipo è postfissa (come in Go) rendendo semplice ometterla nel caso in cui essa non risulti necessaria. Il compilatore Scala infatti è in grado di effettuare una type inference molto avanzata, consentendo al programmatore di non specificare il tipo delle variabili, snellendo il codice.

**Esempio 11.3**

Una variabile viene dichiarata mediante l'utilizzo di `var` se modificabile

```
var msg = "ciao"
print(msg) //stampa "ciao"

msg = "ciao mondo"
print(msg) //stampa "ciao mondo"
```

Una costante (ovvero un valore non modificabile) viene dichiarato tramite la parola chiave `val`

```
val msg = "ciao"
print(msg) //stampa "ciao"

msg = "ciao mondo" //ERRORE
```

Scala inoltre consente di definire all'interno di un *inner scope* una variabile omonima ad una già esistente in un outer scope, provocandone l'oscurazione (shadowing). Questo concetto è potente sebbene sia poco leggibile.

```
var x = 2
var x = 3   //ERRORE, ridefinizione vietata
{
  var x = 4 //CONSENTITO
  print(x)  //stampa 4
}

print(x)    //stampa 2
```

### Funzioni

Esse sono introdotte dalla parola chiave `def` e posseggono una specifica del tipo di ritorno postfissa (come in Go) consentendo nuovamente di non specificarlo nel caso in cui sia possibile inferirlo. Scala introduce vari shortcut evoluti come metodi senza argomenti volutamente indistinguibili dalle proprietà e metodi a singolo argomento utilizzabili come operatori infissi.

**Esempio 11.4: definizione di funzioni**

```
def abs(x:Float) : Float = {
  if (x<0) -x else x
}
```

Si noti la specifica di tipo postfissa e l'assenza della parola chiave `return` (l'ultimo valore computato è quello ritornato dalla funzione). Così come il costrutto `switch` di C e Java, il costrutto `if` di Scala è un'espressione, che pertanto denota un valore.
Nel caso in cui sia possibile dedurre il tipo di ritorno, inoltre, la specifica di tipo può essere omessa.

```
def abs(x:Float) = {
  if (x<0) -x else x
}
```

Un'altra considerazione importante da fare è che gli argomenti sono implicitamente dei `val`, ovvero non è possibile modificarli all'interno della funzione.

**Esempio 11.5: procedure**

```
def hello( s:String ) = {
  println("Hello" + s)
}
```

Una funzione che non restituisce nulla (una procedura, dunque) ha tipo di ritorno `Unit`. In questo caso inoltre è possibile omettere anche il carattere = dalla definizione della

funzione, mantenendo però le parentesi graffe. Bisogna dunque prestare attenzione all'interpretazione: senza simbolo = la funzione sarà sempre interpretata come procedura, anche in presenza di un valore di ritorno.

Una funzione non void priva di un'istruzione `return` esplicita restituisce comunque l'ultimo valore calcolato. Inoltre se una funzione calcola un solo valore, le parentesi graffe attorno al corpo sono opzionali. Anche il `;` si può omettere a fine riga perché viene dedotto dal compilatore.

**Esempio 11.6**

```
def increment(i:Int) : Int = tot += i
```

Scala non consente di definire funzioni con un numero variabile di argomenti in modo classico, ma permette di ripetere l'ultimo argomento con una sintassi analoga alle espressioni regolari. Bisogna però prestare attenzione ai tipi degli argomenti ripetuti: infatti *internamente* alla funzione, l'argomento ripetuto è mappato su un array, quindi il tipo dell'argomento ripetuto è `Array` di quel tipo. Tuttavia *esternamente* il tipo di argomento ripetuto è considerato diverso da un array e quindi incompatibile con esso.

**Esempio 11.7: numero variabile di argomenti**

```
def myprint(args: String*)
```

Infine Scala ottimizza di default la tail recurtion *diretta* (può comunque essere disabilitata).

### Funzioni come first class entities

Le funzioni sono dei veri e propri oggetti istanza di uno specifico tipo funzione e come tali sono manipolabili ed innestabili. Eventualmente possono essere anonime generando delle vere chiusure.

### Classi

Sono introdotte dalla keyword `class`. Ogni classe possiede un costruttore primario definito a partire dai parametri di classe. Questa scelta consente di avere un unico entry point per la creazione degli oggetti, consentendo di migliorare il controllo dei parametri per la creazione degli oggetti, inoltre consente di evitare di utilizzare la classica serie di assegnamenti tipo `this.x = x`. Il costruttore primario include tutto il codice scritto dentro una classe che non sia dentro a una qualche funzione. Esso è unico e viene chiamato automaticamente da tutti i costruttori ausiliari. I costruttori ausiliari invece sono definibili solo tramite `this` e per questo si appoggiano tutti al costruttore primario. Tuttavia i costruttori ausiliari non possono richiamare il costruttore della classe base.

**Esempio 11.8: costruttore primario**

```
class Counter(v:Int) {
  Counter.howMany += 1;
}
```

Il costruttore generato contiene le due righe:

```
this.v = v;
```

```
Counter.howMany += 1;
```

**Esempio 11.9: costruttore ausiliario**

```
class Counter(v:Int) {
  Counter.howMany += 1;
  def this() = this(1)
}
```

Il costruttore ausiliario in questo caso non possiede argomenti e richiama il costruttore primario passandogli l'argomento 1.

I parametri di classe hanno tempo di vita uguale a quello del costruttore primario, quindi se servono anche dopo la terminazione del costruttore occorre copiarli in variabili di istanza (come in Java).

Per ridefinire i metodi ereditati è richiesto l'uso esplicito del qualificatore `override`

**Esempio 11.10: override**

```
class Counter(v:Int) {
  ...
  private var values = v;
  def setValue(v:Int) = { value = v}
  def getValue() : Int = value

  override def toString() = "Valore " + value
}
```

Ogni membro di ogni classe è pubblico a default (in Java la visibilità di default era quella di package). Sono privati invece i parametri di classe. Infine le classi non possono avere parti statiche al loro interno.

### Oggetti singleton

Sono introdotti dalla parola chiave `object` e offrono un' alternativa pulita ai membri statici di una classe. Strutturalmente sono identici ad una classe, tuttavia non definiscono alcun tipo poiché non ne esisteranno altre istanze. Questo però non significa che un singleton non abbia tipo. Infatti un singleton estende una classe base del cui tipo, quindi, si considera istanza. Perciò può essere, per esempio, passato a metodi che accettino tale tipo. Esso non ammette né parametri né costruttori e per questo motivo tutti i suoi membri devono essere già inizializzati. Infine un oggetto singleton può essere usato: da solo (*standalone object*), insieme ad una classe (*companion object*) o nella composizione di tratti.

**Esempio 11.11: oggetto singleton**

```
object Boss {
  private val name = "The Boss";
  def getName() = name
}

object Main {
  def main(args: Array[String]) {
```

```
    println(Boss.getName())
  }
}
```

L'oggetto singleton è usabile senza doverlo creare esplicitamente (come nelle parti statiche di Java)

### Oggetti companion

Sono sostanzialmente dei singleton omonimi di una classe e definiti all'interno dello stesso file della classe. Fra i due vi è una speciale relazione: entrambi possono accedere ai membri privati dell'altro (come fra la parte statica e non statica di una classe Java).

**Esempio 11.12: oggetto companion**

```
class Counter(v:Int) {
  Counter.howMany += 1;
  ...
}
```

Ha l'accesso privilegiato al campo privato `howMany` dell'oggetto companion `Counter`

```
object Counter(v:Int) {
  private var howMany : Int = 0
  ...
}
```

### Classi dati

Scala offre una sintassi per la definizione di classi contenenti solamente dati. Esse godono di supporto automatico al pattern matching avanzato (costrutto `match`). Inoltre tali classi sono dotate di generazione automatica di metodi standard di supporto come metodi factory con lo stesso nome della classe (per evitare le `new`) ed implementazioni "naturali" per alcuni metodo classici (come `toString`, `equals`, ...).

### Pattern matching avanzato

Scala introduce un nuovo costrutto introdotto dalla parola chiave `match` che va a sostituire il vecchio `switch`. Esso abilita un pattern matching avanzato su oggetti e classi dalla grande potenza espressiva che semplifica notevolmente la scrittura di funzioni che lavorano per casi. Adotta un approccio funzionale, pertanto è un'espressione che restituisce un valore. Utilizza una sintassi a casi distinti, senza `break`. Il caso di default è espresso tramite il simbolo underscore.

**Esempio 11.13: match**

```
objectTestMatch {
  main(args: Array[String]) {
    val arg = if (args.length > 0) args(0) else ""
    println(
      arg match {
        case "Pippo"   => "Goofy"
        case "Paperino"  => "Donald Duck"
        case _       => arg //untouched
      }
```

```
      )
    }
  }

  //test
  TestMatch.main(Array("Pippo", "Pluto", "Paperino"))
  > Goofy
```

### Oltre all'ereditarietà multipla

Pur estendendo una singola super-classe ogni classe può ereditare da più tratti (vedi Sezione 11.3) con una nuova semantica di composizione chiamata mix-in. Classi e tratti vengono mixati per linearizzazione portando come risultato ad uno *stackable behaviour*.

### Package innestati

Scala supporta package semanticamente innestati (non come in Java), offrendo una doppia sintassi per la loro definizione (Java classica, o in stile C#). Il package di top level viene identificato da `__root__` in modo da poter scrivere anche riferimenti assoluti tra i package.

La direttiva `import` importa anche i singoli oggetti ed altri package (non solo classi ed interfacce) in maniera eventualmente ricorsiva sui sotto package.

### Strutture di controllo

Scala ha solamente cinque strutture di controllo predefinite (`for`, `while` e `do-while`, `try`, `match`), secondo il principio "più librerie, meno sintassi", viene lasciata inoltre la possibilità all'utente di definire nuove strutture di controllo.

Tutte le strutture di controllo sono espressioni e dunque restituiscono un valore del tipo giusto in un dato contesto. Questo approccio più funzionale consente di avere codice più semplice e leggibile, evitando l'utilizzo di variabili di appoggio (ed evitandone soprattutto la gestione).

Come già accennato:

- L'`if` di Scala è equivalente all'operatore ternario del C
- Viene preferito l'uso del `for` (all'uso del `while`) in quanto esso è più funzionale
- L'assegnamento è un'espressione di tipo `Unit`

### Programmi stand alone

In Scala il `main` non può stare in una classe (la quale non ammette membri statici), ma è contenuto in un oggetto **singleton**.

**Esempio 11.14: dichiarazione del main**

La dichiarazione del main cambia leggermente rispetto a Java e passa da

```
public static void main(String[] args)
```

a

```
def main(args: Array[String])
```

Si noti come non vi sia più necessità di utilizzare la keyword `static` poiché tutti i membri dell'`object` lo sono già

### Stili a confronto

Può essere utile vedere i vari stili di programmazione che è possibile adottare in Scala.

**Esempio 11.15: stampe**

**Stile imperativo** (`foreach` simile a quello di java). Esprime un controllo assoluto sull'algoritmo: per capire il significato del programma, occorre simularlo mentalmente.

```
for (s <- args){
  println("Argomento: " + s)
}
```

**Stile funzionale** (lambda expression). Fattorizza il controllo: per capire cosa fa il programma basta capire il significato dell'espressione passata come argomento.

```
args.foreach( s => println("Argomento: " +s) )
```

**Stile pienamente funzionale**. Astrae il controllo (stile dichiarativo): per capire cosa fa il programma basta conoscere la semantica della `println`

```
args.foreach(println)
```

**Approfondimento sul for** Osservando l'esempio precedente si potrebbe notare una somiglianza sintattica tra il `for` di Scala e quello di Java, tuttavia essi sono molto diversi. L'argomento `s` infatti è un `val` e non una variabile `var`. Il valore `s` sembra cambiare solo perché a ogni iterazione viene creato un nuovo valore: esso però non è modificabile dentro il ciclo. Inoltre l'espressione interna è un **generatore**, ovvero un'espressione che genera una serie di valori su cui iterare che può diventare anche molto complessa.

**Esempio 11.16: for**

```
val v = Array("Paperino", Pippo", "Pluto");
for (i <- 0 to 2) {
  println(v(i))
}
```

L'espressione `0 to 2` è un generatore che restituisce un oggetto di classe `Range` che in questo caso rappresenta i valori {0,1,2}. Questo particolare generatore utilizza la **sintassi infissa** che è semplicemente un altro modo di chiamare un metodo (equivale a `0.to(2)`). In generale in Scala un metodo con un solo argomento può essere sempre chiamato in questo modo. L'array invece è inizializzato tramite l'uso delle parentesi (vedi in seguito). L'espressione generatrice può anche contenere *condizioni di filtro*. Per esempio per stampare solo le stringhe con al più 6 caratteri e che contengono il carattere "i":

```
val v = Array("Paperino", Pippo", "Pluto");
for (i <- v if s.length()<=6 ; if s.contains("i")) {
  println(s) // stampa solo "Pippo"
}
```

Si possono facilmente implementare anche i for innestati semplicemente concatenando più generatori.

```
for (i <- i to 5; j <- 'A' to 'B') {
  println("Cella " + i + "." + j);
}

// OUTPUT:
```

```
// Cella 1.A
// Cella 1.B
// Cella 2.A
// Cella 2.B
// ...
```

Per accumulare i risultati parziali calcolati durante un'iterazione è possibile utilizzare la keyword `yield`

**Esempio 11.17: yield**

```
def grep(path: String, pattern: String) : Array[String] = {
for ( file <- new java.io.File("./" + path).listFiles
  if file.getName().endsWith(".scala");
  line <- scala.io.Source.fromFile(file).getLines.toList;
  temp = line.trim;
  if temp.matches(pattern)) {
    yield temp
  }

def main(args: Array[String]) {
for ( s <- grep("src", ".*main.*") )
  println(s)
}
```

La funzione `grep` permette di estrarre da più file tutte le righe che soddisfano un certo pattern. In `Array[String]` vengono accumulati tutti i valori yielded ("prodotti"), mentre `yeld temp` trattiene tutti i risultati parziali. In questo caso il for è sfruttato come espressione che ritorna un valore di tipo array di stringhe

### Parentesi come shortcut

In Scala una lista di valori fra parentesi usata come R-value sottintende sempre una chiamata implicita al metodo `apply` dell'oggetto ricevente. Analogamente una lista di valori fra parentesi usata come L-value sottintende una chiamata implicita al metodo `update` dell'oggetto ricevente.

**Esempio 11.18: parentesi come shortcut**

```
//shortcut per l'inizializzazione di un array
val v = Array("a", "b", "c");

//espansione
val v = Array.apply("a", "b", "c");

//shortcut per la scrittura della cella di un array
v(2) = "a"

//espansione
v.update(2, "a");
```

### Identificatori

Il backtick rende legale qualunque sequenza di caratteri accettata dal runtime.

**Esempio 11.19: identificatore con backtick valido**

```
`while`
```

Esso è considerato un identificatore lecito

Una sequenza di caratteri-operatore è un unico operatore. Sono caratteri operatore anche i caratteri Unicode math symbols e other symbols, nonché alcuni classici caratteri ASCII. Per questo è opportuno mettere spazi intorno all'operatore anche in quei casi in cui non sarebbe obbligatorio in Java.

**Esempio 11.20: interpretazione**

Consideriamo `a<-b`. In Java viene interpretato come:

```
a < -b
```

In Scala invece come:

```
a <- b
```

L'underscore assume un significato particolare e introduce gli **identificatori misti**. La loro sintassi è costituita da `parola_operatore`.

**Esempio 11.21: identificatori misti**

Operatori unari:

```
unary_-
```

Metodi setter di proprietà:

```
myprop_=
```

### Conversioni implicite

Come in C++, Scala consente di definire conversioni implicite di tipo che saranno applicate automaticamente, utili per permettere l'uso commutativo degli operatori.

**Esempio 11.22: frazioni che si sommano anche a interi**

Il metodo + somma un intero passato per argomento con una `Frazione`

```
def +(i:Int): Frazione = sum(new Frazione(i))

println(f + 3) // OK
println(3 + f) // NO, + e' un metodo di frazione non di Int
```

Per rendere simmetrico l'uso del + occorre definire una conversione implicita tramite la keyword `implicit`

**Esempio 11.23: conversione implicita**

Il metodo di conversione implicita deve essere definito fuori dalla classe `Frazione`, ad esempio nell'oggetto companion.

```
// Companion Object
object Frazione {
```

```
  implicit def conv(int:Int): Frazione = new Frazione(i)
  def main (args: Array[String]) {
    ...
    println(f1 + 3) // OK con operatore + overloaded
    println(3 + f1) // OK con conversione implcita
  }
}
```

Si noti che il nome del metodo `conv` è del tutto arbitrario

### Eccezioni

Le eccezioni in Scala sono simili a Java ma:

- Non esiste la clausola `throws`. Esiste però l'annotazione `throws` con analoga semantica
- L'espressione `throw` ha un tipo di ritorno `Nothing`. Essa tuttavia non viene mai valutata e quindi non produrrà mai un valore
- La clausola `catch` opera per pattern matching. Utilizza una sintassi senza parentesi tonde e con case interni al blocco
- Il costrutto `try/catch/finally` è un'espressione e restituisce:
  - Il valore del ramo `try` se esso ha avuto successo
  - Il valore di uno dei rami `catch` se l'eccezione è stata catturata
  - Nessun valore se l'eccezione è stata lanciata ma non catturata

Per ogni `case` si usa la sintassi:

```
case e: tipo eccezione => expr
```

in cui l'espressione è opzionale mentre la freccia obbligatoria. Infine a differenza di Java, l'eventuale valore calcolato nel `finally` non è considerato e va quindi perduto.

**Esempio 11.24: test eccezioni**

```
object TestExceptions {
  def main(args: Array[String]) {
    val myurl = getURL("http://enricodenti.disi.unibo.it")
    println(myurl)
  }
def getUrl(url: String) {
  try {
    new java.net.URL(url)
  } catch {
    case e: java.net.MalformedURLException => println("bleah")
  }
  finally println("done.")
}
```

L'output con un sito che inizia per "http" è:

```
> done.
> ()
```

Se il `try` ha successo infatti il tipo di ritorno di `getURL` è `Unit` (visto che manca = dopo la definizione della `getUrl`) e il suo valore è `()`. Dunque il vero valore calcolato dalla

funzione va perso.
L'output senza "http" all'inizio è:

```
> bleah
> done.
> ()
```

### Letterali-funzione

Un letterale funzione (lambda) ha la forma: `(args) => blocco`. Il blocco può essere costituito da un'unica espressione (e in quel caso possono essere omesse parentesi graffe), oppure da più espressioni. In quest'ultimo caso il tipo e il risultato del letterale sono quelli dell'espressione *più a destra*. La funzione può essere eseguita direttamente tramite l'operatore di chiamata `()` o indirettamente tramite il metodo `apply` della classe `FunctionN`. Inoltre sono ammesse alcune sintassi brevi per casi frequenti, ad esempio:

- Se il tipo degli argomenti può essere dedotto, allora può essere omesso. In tal caso si possono anche omettere le parentesi tonde
- Se l'argomento appare una sola volta, allora può anche essere omesso e al suo posto si mette un underscore. In questo caso anche l'operatore => viene omesso

**Esempio 11.25: letterali funzione**

```
val f = _+1 // equivale a x => x+1

f(2) // equivale a f.apply(2) e restituisce 3
```

I letterali funzione sono utili anche per passare un comportamento ad altri metodi. Ad esempio nel seguente esempio si passa a filter il comportamento del filtro, ovvero quello di filtrare i numeri pari.

```
val v = Array(2,4,7)

v.filter(_%2==0) // restituisce un Array contenente i valori 2 e 4
```

Infine bisogna prestare attenzione al fatto che ogni underscore sta per un diverso argomento e non è possibile utilizzarlo per più occorrenze dello stesso. Quindi se una funzione prevede due argomenti (ovvero due underscore) non è possibile passargliene solo uno.

**Esempio 11.26: funzione con più argomenti**

```
val f = (_ : Int) * (_ : Int)

f(3,3) // OK
f(3)   // NO!
```

### Funzioni parzialmente specificate

L'uso del underscore è un caso particolare di funzione parzialmente specificata. Tramite essa si definisce una nuova funzione, con meno argomenti, "sopra" un'altra che ne avrebbe di più.

**Esempio 11.27: funzioni parzialmente specificate**

```
// funzione base
def sum(a:Int, b:Int, c:Int) = a+b+c

//funzioni parzialmente specificate
val sum3 = sum_ // sostituisce l'intera lista di argomenti
val sum2 = sum(10, _:Int, _:Int)
val sum1 = sum2(20, _:Int) // sostituisce un singolo argomento
val sum0 = sum1(33)

sum(3,4,5)  // 12
sum3(3,4,5) // 12
sum2(3,4)   // 17 (10+3+4)
sum1(3)     // 33 (10+20+3)
sum0        // 63 (10+20+33)
```

L'underscore è in generale sempre richiesto per sostituire l'argomento mancante (o la lista di argomenti). L'unico caso in cui l'underscore si può omettere è quando l'argomento stesso è una funzione.

**Esempio 11.28: omissione dell'underscore**

```
Array(3,4,5).foreach(println _)
Array(3,4,5).foreach(println)
```

Entrambe le forme sono accettate poiché è implicito che l'argomento di `foreach` possa essere solo una funzione

### Chiusure

Il pieno supporto alle funzioni come first-class object comporta un altrettanto pieno supporto alle chiusure. Le variabili libere sono chiuse su riferimenti alle variabili esterne, quindi percepiscono i loro cambiamenti nel tempo. In caso vi siano istanze multiple di tali variabili, chiusure diverse ne possono fotografare istanze diverse.

**Esempio 11.29: chiusure**

Questa lambda expression contiene una variabile libera (`x`):

```
val f = (a:Int) => a+x
```

Se eseguita in un ambiente in cui `x` non è definita dà errore, mentre in un ambiente in cui è definita definisce un nuovo valore-funzione, ovvero la chiusura. Ovviamente se `x` cambia, cambia anche il valore della chiusura. Coerentemente se un ambiente più interno definisce una nuova variabile omonima (*shadowing*), la variabile libera rimane legata comunque a quella originale.

```
val x = 500
val f = (a:Int) => a+x
f(21) // restituisce 521

x = 44
f(21) // restituisce 65

val x = 600
```

```
f(21) // continua a restituire 65
```

Le chiusure inoltre sono un modo efficace per creare effetti permanenti.

**Esempio 11.30: somma elementi di un array**

```
val elenco = Array(2,3,5,7)
var somma = 0
elenco.foreach(x => somma += x) // somma = 17
// elenco.foreach(somma += _) con sintassi shortcut
```

All'interno del `foreach` vi è una lambda expression con una variabile libera (`somma`). Il valore di `somma`, modificato dalla funzione, sopravvive alla chiusura ed è dunque utilizzabile come accumulatore del risultato

### Currying

Se una funzione ha più argomenti, di norma si passano tutti assieme nella classica lista degli argomenti. Una funzione *curried* invece li applica in sequenza.

**Esempio 11.31: currying**

```
// diff non curried
def diff(a:Int, b:Int) = a-b

//chiamata
val result = diff(8,3)

// diff curried
def diffC(a:Int)(b:Int) = a-b

//chiamata
val result = diffC(8)(3)
```

Nel secondo caso si stanno in realtà definendo due funzioni: una esterna che dato `a` restituisce una funzione con argomento `b` e una funzione interna che dato `b` restituisce il risultato finale. La diff con currying equivale quindi a:

```
def fExt(a:Int) = (b:Int) => a-b
val fInt = fExt(8)
val result = fInt(3)
```

### Block-like syntax

Scala permette di usare parentesi graffe anziché tonde in presenza di un solo argomento affinché funzioni definite dall'utente sembrino anche visivamente built-in

**Esempio 11.32: block-like syntax**

Data la funzione:

```
def raddoppia(v:Double):Double = v*2
```

Invocazione standard:

```
raddoppia(5) // restituisce 10.0
```

Invocazione con block-like syntax:

```
raddoppia {5} // restituisce 10.0
```

Nel primo caso, `raddoppia` si riconosce essere una funzione definita dall'utente a causa delle parentesi tonde usate per delimitare l'argomento 5. Nel secondo caso invece la presenza delle graffe fa sembrare `raddoppia` una keyword che introduce un costrutto built-in del linguaggio

### Call by name

Come è noto la strategia *call by name* valuta un argomento solo al momento dell'uso, mai a priori. Solitamente questa strategia non è disponibile nativamente nei linguaggi imperativi che adottano il modello applicativo. Rimane comunque emulabile passando funzioni anziché valori (facile nei linguaggi che supportano funzioni come first-class entity). In Scala non è tuttavia necessario emularla, infatti un argomento passa automaticamente *by name* se preceduto da =>. Questo è di fatto un modo compatto per definire la lambda poiché il cliente passerà semplicemente `exp` anziché `() => arg`. La funzione può dunque usare direttamente l'argomento ricevuto `v` senza doverselo procurare invocando `v()`.

**Esempio 11.33: call by value vs call by name**

Call by value classica (modello applicativo):

```
def se(c:Boolean, a:Int, b:Int) : Int = {
  if(c) a else b
}

se(5>4, 3*4, 13/0) // Esplode! Anche se 13/0 non viene mai usato
```

Call by name:

```
def se(c:Boolean, a: =>Int, b: =>Int) : Int = {
  if(c) a else b
}

se(5>4, 3*4, 13/0) // OK, restituisce 12 (3*4)
```

Senza questa notazione shortcut avremmo dovuto gestire in esplicito il passaggio di letterali-funzioni e il loro uso, come in Javascript.

### Definire nuovi costrutti

Ora che abbiamo tutti gli ingredienti possiamo definire dei nuovi costrutti di controllo che sembrino built-in.

**Esempio 11.34: costrutto repeat**

Vogliamo definire un nuovo costrutto che riceva una intero `n` e una procedura `action` e che ripeta `n` volte tale procedura. Possiamo implementarlo in modo tale che se `n>1` chiami `action` e poi richiami se stesso con `n-1` e `action` (tenere presente che la *tail recurtion* è ottimizzata di default in Scala). Si utilizza la call by name per la procedura `action`, il currying per separare gli argomenti e infine la block-like syntax lato chiamante.

```
def repeat(n:Int)(action: =>Unit) : Unit = {
  action; if (n>1) repeat(n-1)(action)
}
```

```
// chiamata
repeat(3) { println("ciao" } // stampa 3 volte "ciao"

// se repeat(3) dovesse servire spesso:
val rep3 = repeat(3)_
rep3 { println("ciao"); }
```

Il nuovo costrutto funziona anche con più di un'istruzione nel blocco, perché quello è formalmente il corpo di un letterale-funzione.

```
repeat(3){
  val a=12+3;
  println(a);
} //stampa 3 volte 15
```

### Funzioni senza argomenti

A differenza di Java, Scala ammette funzioni *senza lista di argomenti*, da non confondere con le funzioni con lista di argomenti vuota.

**Esempio 11.35**

Funzione con lista di argomenti vuota:

```
def fun() : Int = 10

//invocabile con o senza l'operatore ()
fun() // restituisce 10
fun   // restituisce 10
```

Funzione senza lista di argomenti:

```
def fun : Int = 12

// invocabile solo senza l'operatore ()
fun   // restituisce 12
fun() // errore
```

Le funzioni senza lista di argomenti sono importanti perché supportano il **principio di accesso uniforme**.

**Definizione 11.1: principio di accesso uniforme**

Un cliente non deve sapere se una data proprietà sia implementata come funzione o come dato

Le funzioni con lista di argomenti vuota non lo rispettano completamente perché la possibilità di usare le parentesi rivela che si tratta di funzioni. Al contrario le funzioni senza lista di argomenti sono indistinguibili da un campo dati perché, come lui, non accettano le parentesi. Le funzioni senza argomenti possono essere usate per i puri *accessor*, ossia funzioni che leggono dati senza modificarli. Invece le funzioni con lista argomenti (che può essere eventualmente vuota) vanno usate quando è bene che si veda che una funzione sia tale e non un dato. Tecnicamente le due tipologie si possono mischiare in libertà nell'override dei metodi: un metodo senza lista di argomenti può sovrascriverne uno che aveva lista di argomenti vuoti, e viceversa.

## 11.2 Classi astratte e ereditarietà

Le classi astratte in Scala sono simili a quelle di Java. Si utilizza la keyword `abstact` all'inizio della classe, ma non nei singoli metodi. Una classe derivata si definisce come in Java con la parola chiave `extends` nella dichiarazione. Se la classe base è omessa si sottintende `AnyRef`. Tuttavia a differenza di Java, non si ereditano i membri privati e nemmeno quelli omonimi poiché vengono sovrascritti. Per il principio di accesso uniforme, l'indistinguibilità tra metodi e dati implica che un campo-dati possa ridefinire un metodo senza argomenti, e viceversa. Inoltre metodi e dati appartengono allo stesso namespace e quindi, a differenza di Java, non possono esistere metodi e dati omonimi fra loro.

**Esempio 11.36: cerchio che estende Forma**

```
abstact class Forma {
  def area: Double
  def perimetro : Double
  override def toString = "Una forma qualsiasi"
}

class Cerchio(val raggio:Double) extends Forma {
  // val introduce un campo dati pubblico omonimo del parametro di
     classe
  def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override def toString = "Un cerchio di raggio: " + raggio +
         ", area: " + area + " e perimetro: " + perimetro
}

// Utilizzo
def main(args: Array[String]) {
  val c1 = new Cerchio(1)
  println(c1)
}
  println(c1.raggio) // accessibile essendo pubblico

// OUTPUT
// Un cerchio di raggio: 1.0, area: 3.1415926 e perimetro: 6.2831853
1.0
```

Ovviamente se si usa `var` al posto di `val` nella dichiarazione dell'argomento del metodo, tale valore diventa modificabile. Ad esempio:

```
// Utilizzo
def main(args: Array[String]) {
  val c1 = new Cerchio(1)
  println(c1)
  c1.raggio = 2.0 // var, quindi modificabile
  println(c1)
}

// OUTPUT
// Un cerchio di raggio: 1.0, area: 3.141592 e perimetro: 6.283185
// Un cerchio di raggio: 2.0, area: 12.56637 e perimetro: 12.56637
```

Infine per il principio di accesso uniforme, volendo, è possibile ridefinire `toString` come `val` anziché `def`. Ovviamente se eseguiamo una stampa dopo aver modificato il raggio,

comparirà ancora quella con i valori precedenti. Questo poiché `val` rende la funzione una "costante", ovvero non viene più ricalcolata ogni volta

Infine in Java il costruttore di una sottoclasse si appoggia sempre a un costruttore della classe base (o a quello di default o a quello invocato tramite la `super`). Invece in Scala come si è già visto vi è un solo costruttore primario, mentre gli altri sono tutti ausiliari rimappati sul primario tramite `this`. Coerentemente una classe derivata può invocare solo il costruttore primario della sua classe base, senza usare la `super` ma indicandolo direttamente nella dichiarazione `extends`.

**Esempio 11.37: costruttori**

```
abstact class Forma(nome: String) {
  def area: Double
  def perimetro : Double
  override def toString = nome
}

class Cerchio(val raggio:Double) extends Forma {
  def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override def toString = nome " di raggio: " + raggio +
           ", area: " + area + " e perimetro: " + perimetro
  def this(v:Double) = this("cerchio", v)
}

// Utilizzo
def main(args: Array[String]) {
  println(new Cerchio(1));
  println(new Cerchio("Monetina", 1))
}

// OUTPUT
// cerchio di raggio: 1.0, area: 3.1415926 e perimetro: 6.2831853
// Monetina di raggio: 1.0, area: 3.1415926 e perimetro: 6.2831853
```

### 11.2.1 Tassonomia di Scala

La classe base `Any` definisce i metodi generali, mentre le due sottoclassi `AnyVal` e `AnyRef` fanno da base rispettivamente ai valori e ai riferimenti. I metodi generali agiscono in modo trasparente su entrambi, in particolare `==` e `!=` hanno la semantica di equals e not equals. Le sottoclassi di `AnyVal` sono tutte astratte e finali e non si possono né estendere né istanziare poiché i loro valori si scrivono solo come letterali (ad esempio l'intero 14 è `14` non `new Int(14)`). Scala definisce anche dei *bottom type* come sottotipi comuni a tutti i tipi per rendere coerente il type system nei casi limite. `Null` è il tipo della costante `null`, sottoclasse implicita di ogni tipo che derivi da `AnyRef` (incompatibile quindi con gli `AnyVal`). `Nothing` è invece il sottotipo di chiunque. Esso non ha istanze e serve a rendere coerente il type system nelle terminazioni anomale (ad esempio nelle funzioni di errore).

**Esempio 11.38: if e Nothing**

Se un `if` ha uno dei due rami a `Nothing`, allora il tipo dell'`if` è *quello dell'altro ramo* (come è giusto che sia).

## 11.3 Tratti e composizionalità

In Java classico (< Java 8) le interfacce non possono contenere codice né dati (al più, costanti). Scala supera le interfacce classiche introducendo i **tratti**. Sono più simili a classi che a interfacce e possono contenere codice. Come classi e interfacce essi definiscono un tipo, ma supportano una nuova forma di **composizionalità**: il *MIX-IN*. Si tratta di una classe che estende una superclasse ma che può anche "mixarsi" con un numero arbitrario di tratti che vengono composti assieme mediante *linearizzazione*. Più precisamente un classe può:

- **Estendere** un solo tratto (keyword `extends`), ereditandone la classe base. I riferimenti `super.x` a dati o metodi della superclasse sono risolti staticamente a compile time
- **Comporsi** con molti tratti (keyword `with`). In questo caso non ne eredita la classe base e i riferimenti `super.x` a dati o metodi di un tratto sono risolti dinamicamente a run time

Va notato che un'interfaccia Java 7/C# è un caso particolare di tratto senza codice e con sole dichiarazioni. In questo caso è implementato a livello di bytecode come `interface`. Tuttavia dovendo una classe implementare tutti i metodi di un'interfaccia, spesso si definiscono interfacce molto sottili (anche di un solo metodo) a scapito delle interfacce ricche, a causa della loro scomodità. In Scala i tratti consentono un diverso trade-off poiché è possibile definire un tratto con pochi metodi astratti e molti metodi concreti implementati sopra di essi.

Le classi hanno noti problemi con l'ereditarietà multipla. Le interfacce la supportano tranquillamente poiché, essendo senza codice, evitano alla radice i noti rischi di duplicazione dei dati e/o dei metodi. I tratti tuttavia permettono di scavalcare il problema poiché, pur potendo contenere dati o metodi, non avendo costruttori non possono inizializzare dati. C'è infatti sempre e solo un unico costruttore primario, ovvero quello dell'unica classe da cui un tratto eredità.

Sintatticamente un tratto è analogo a una classe: si utilizza la keyword `trait`, si ha la possibilità di definire dati e metodi, ridefinire metodi, etc. Tuttavia non si hanno parametri di classe perché, come detto, non ci sono costruttori.

**Esempio 11.39: animale terrestre modellato con un tratto**

```
abstract class Animale(nome:String) {
  def vive : String
  def siMuove : String
  override def toString = nome
}

trait AnimaleTerrestre {
  val vive = "sulla terraferma"
  override def toString = super.toString + " vive " + vive
}

trait Quadrupede {
  override def toString = super.toString + " ha 4 zampe"
}

class Cavallo(n:String) extends Animale(n) with AnimaleTerrestre
    with Quadrupede {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove " + siMuove
}

// Chiamata nel main
```

```
println(new Cavallo("Furia del West"));

// OUTPUT
// Furia del west vive sulla terraferma ha 4 zampe e si muove
    cavalcando
```

Come si può notare dal precedente esempio le chiamate a `super` sono risolte per *linearizzazione*, ovvero viene ripercorsa la sequenza di `extends` e `with` con un criterio prestabilito ottenendo una **lista di tratti**. Nei casi facili (come quello precedente) il risultato è un semplice stack di chiamate a funzione (es. `toString`), dall'ultima alla prima. Il meccanismo di linearizzazione implementa il mix-in di classi e tratti in modo trasparente ma controllato. Funzione bene poiché i tratti non hanno costruttori, perciò non potranno mai invocare costruttori della classe base, permettendo di linearizzare l'albero di generazione. Inoltre degli $n$ genitori di un tratto, uno (il primo) può prevalere sugli altri perché può essere una classe.

**Algoritmo 11.1: linearizzazione**

Data una classe `Q` che deriva da una classe base `A` e si mixa con un certo numero di tratti `B`, `C`, etc.

`class Q extends A with B with C ...`

- Si considera la classe base `A` e se ne copia la linearizzazione (se `A` non ha classe base esplicita si considera `AnyRef` che a sua volta deriva da `Any`)
- Si considerano i vari tratti dall'ultimo al primo (in questo caso prima `C` poi `B`) e per ciascuno si ripete il procedimento, ma escludendo le classi già considerate prima
- Si concatenano via via i vari contributi appendendoli in testa, ottenendo una lista in cui ogni classe compare una sola volta

Infine la risoluzione delle chiamate `super` segue tale lista.

Dunque una classe Scala eredita sempre da una sola classe ma al contempo la "mixa" con altri contributi che però rimangono su un diverso piano logico e concettuale.

**Esempio 11.40: ancora animali**

```
abstract class Animale(nome:String) {
  def vive : String    // astratto
  def siMuove : String // astratto
  override def toString = nome + " vive" + vive + ", si muove " +
     siMuove
  def arti : String    // astratto
}

trait AnimaleTerrestre {
  val vive = "sulla terraferma"
}

trait Gambuto extends Animale {
  override def arti = "gambe"
}
```

```
trait Zamputo extends Animale {
  override def arti = "zampe"
}

trait Bipede extends Animale {
  override def toString = super.toString + ", ha 2 " + arti
}

trait Altissimo extends Bipede {
  abstract override def arti = super.arti + " lunghissime"
  // la doppia qualifica abstract override rende legale la chiamata
      super.arti anche se tale proprieta' risulta astratta in Bipede
}

// Alcune composizioni corrette:
println(new Animale("Zombie") with AnimaleTerrestre with Bipede with
     Zamputo )

println(new Animale("Fauno") with AnimaleTerrestre with Bipede with
    Zamputo with Altissimo )

// Composizione non corretta: Altissimo specializza un arti che non
    esiste prima che sia definito in Gambuto
println(new Animale("Sgrodolo") with AnimaleTerrestre with Bipede
    with Altissimo with Gambuto )
```

Infine conviene chiedersi quando è meglio usare tratti e quando le classi. Innanzitutto se il comportamento non sarà riusato è meglio una classe concreta, altrimenti è meglio un tratto. Se una classe deve ereditare o estenderne un'altra, allora è meglio utilizzare una classe astratta. Se invece l'efficienza conta molto è meglio usare una classe concreta poiché la JVM è più veloce ad invocare una classe rispetto ad un'interfaccia. In tutti gli altri casi meglio un tratto.

## 11.4 Scala collections

Le scala collection sono fornite in due versioni: **mutable** (anche se è meglio non usarle) e **immutable** (favorite sotto ogni punto di vista) e consentono di scrivere meno "boilerplate code".[1] Infatti non c'è nessuna necessità di ricordare nomi di interfacce o classi, gli array vengono trattati in maniera uniforme ed è presente un comodo metodo `mkString` che permette di ottenere una stringa con tutti gli elementi di una collection divisi da un separatore configurabile. Si hanno ha disposizione 4 tipi di collezione: Liste, Tuple, Set e Mappe.

### Liste

`List` rappresenta una lista *immutabile*. Si tratta di una classe e non di un'interfaccia come in Java.

**Esempio 11.41: liste**

```
// costruzione array-like
val l1 = List("Paperino", "Pippo", "Pluto");
```

[1] sezioni di codice uguali che devono essere ripetute in varie parti del programma. In altre parole, è necessario scrivere numerose righe di codice anche per portare a termine compiti semplici.

```
// costruzione con operatore ::
val l2 = "Qui" :: "Quo" :: "Qua" :: Nil;

//concatenazione con operatore :::
val l3 = l1 ::: l2;
```

Da notare che gli operatori `::` e `:::` sono metodi dell'oggetto alla loro destra. C'è infatti la convenzione per cui gli operatori sono solitamente associativi a sinistra, *tranne* quelli il cui nome termina con `:`. Quindi le ultime due liste equivalgono a:

```
val l2 = Nil.::("Qua").::("Quo").::("Qui");
val l3 = l2.:::(l1);
```

Un esempio di utilizzo del metodo `mkString` può essere:

```
l2.mkString(", ") // restituisce: Qui, Quo, Qua
```

Tuttavia un fattore da tenere a mente è che l'append è troppo costosa nelle liste immutabili. Le possibili alternative sono:

- Usare `::` e `reverse`
- Usare una lista mutevole (`ListBuffer`) come appoggio, ottenendone una copia immutabile solo alla fine tramite la `toList`
- Convertire la lista immutabile in `Buffer` tramite la `toBuffer`, operare su essa tramite l'operatore append += e infine tornare a `List`

```
// Appendere Paperone alla lista l1 (primo metodo)
("Paperone" :: l1.reverse).reverse // () omesse (no argomenti)

// Appendere anche Paperoga (secondo metodo)
(l1.toBuffer += "Paperoga").toList
```

### Tuple

La classe `Tuple` rappresenta una tupla immutabile che può contenere elementi di tipi diversi. Si costruiscono con l'operatore `(...)` e vengono accedute tramite l'operatore `_index`. L'operatore di accesso non può essere `(index)` come nelle liste poiché, essendo in generale gli elementi di una tupla di tipo diverso, essi richiedono metodi diversi con tipi di ritorno differenziati. Infine gli elementi di una tupla sono numerati a partire da 1 (per motivazione storica, era così anche in Haskell e ML) e il loro tipo è una delle classi da `Tuple2` a `tuple22`.

**Esempio 11.42: tuple**

```
val p = ("Giovanni", 20)
// p._1 restituisce: Giovanni
// p._2 restituisce: 20
```

### Set

È un tratto che esiste in due versioni (mutable e immutable) ed è implementato dalla classe `Hashset`. La costruzione avviene mediante l'operatore `(...)`, mentre l'aggiunta di elementi con l'operatore +=. Bisogna però prestare attenzione alla semantica di += : nei set mutevoli esiste un vero operatore +=, invece nei set immutabili non esiste, quindi viene interpretato

come uno shortcut per la creazione di un nuovo set (come nelle stringhe di Java), che è corretto purché il riferimento sia dichiarato come `var` e quindi ammetta la riassegnazione di un nuovo set (pure lui immutabile). Per passare dalla versione mutable a quella immutable e viceversa basta semplicemente cambiare la `import`.

**Esempio 11.43: set**

```
val set1 = Set(1, 3, 5); // default: immutabile
set1 += 7 // errore! += non appartiene ai set immutabili

var set2 = Set(1, 3, 5);
set2 += 7 // OK, il riferimento e' un var

//scelta esplicita di un set mutevole
val set3 = scala.collection.mutable.Set(1, 3, 5);
set3 += 7 // OK anche se il riferimento e' un val. Infatti in questo
    caso il set e' modificabile "per natura"
```

A questo punto ci si potrebbe chiedere: come è possibile che `set` si possa istanziare se è definito come un tratto? In realtà non stiamo realmente "istanziando" un'interfaccia, bensì stiamo soltanto invocando un metodo factory dell'oggetto companion. Infatti la frase `Set(...)` è uno shortcut linguistico per l'invocazione del metodo `apply` dell'oggetto companion di Set.

**Esempio 11.44**

```
val set1 = Set(12, 13);

//equivale a

Set.apply(12, 13)
```

### Mappe

Anche `Map` è un tratto che esiste sia in versione mutable che immutable ed è implementato dalla classe `HashMap` (come prima per passare da una versione all'altra basta cambiare la `import`). Ha gli stessi operatori di `set` per quanto riguarda costruzione e aggiunta elementi. Quest'ultima può anche essere realizzata tramite tuple speciali `key -> value`. Anche in questo caso l'operatore `(...)` è uno shortcut per il metodo `apply` del companion object. Tuttavia per le `map` occorre prestare attenzione ai tipi. Infatti i `set` ospitano oggetti omogenei e dunque il tipo è deducibile, nelle mappe invece il tipo di chiavi e valori non è sempre deducibile e talvolta può essere necessario specificarlo esplicitamente.

**Esempio 11.45: map**

```
// Mappa mutevole
val myMap = Map(1 -> "Qui", 2 -> "Quo", 3 -> "Qua")
myMap += (0 -> "Goo")

// Mappa immutabile con riferimento var, += interpretato come
   shortcut
var map1 = Map[Int, String]()
map1 += (1 -> "Pippo")
```

```
map1 += (2 -> "Pluto")
map1 += (6 -> "Zio Paperone")
```

Infine poiché in Scala ogni operatore equivale ad un metodo, la sintassi `key -> value` equivale di fatto a `key.->(value)`.

```
myMap += 3 -> "Goo"

// equivale a

myMap += (3).->("Goo")

// Ovvero si invoca il metodo -> sull'oggetto 3 (Int). Le parentesi
    servono solo ad evitare che 3 sia inteso come Double
```

## 11.5 Varianza e covarianza

In Java, la varianza di un argomento è specificabile tramite le wildcard solo nel punto in cui viene usata. Scala offre invece una soluzione più sofisticata rispetto a Java: è possibile specificare anche la varianza "assoluta" di un tipo:

- **lower bounds** `[U >: T]` per tipi covarianti [+T]
- **upper bounds** `[U <: T]` per tipi controvarianti [-T]

Il compilatore naturalmente verifica che l'utilizzo dei bound sia coerente con la specifica di varianza del tipo.