LINGUAGGI E MODELLI COMPUTAZIONALI

(PROF. DENTI)

LAURA GRUPPIONI

Sommario

0 - OVERVIEW E INTRODUZIONE

0.1 - COS'È UN LINGUAGGIO DI PROGRAMMAZIONE

LINGUAGGIO DI PROGRAMMAZIONE DI ALTO LIVELLO: strumento per far eseguire ad un elaboratore le funzioni desiderate ad un livello concettuale più elevato rispetto a quello permesso/indotto dall'hardware. Servono **opportuni traduttori** (compilatori e interpreti) che colmino il divario con la macchina sottostante.

Storicamente l'informatica è sempre stata legata alla **computabilità**, oggi invece l'informatica non serve a nulla se non è anche **interattiva**: bisogna integrare il linguaggio di programmazione e i livelli sottostanti. **Ogni livello costituisce una MACCHINA VIRTUALE** che ha un suo **sistema di interpretazione di simboli** e richiede:

- **ANALISI SINTATTICA**: correttezza strutturale dei comandi
- **ANALISI SEMANTICA**: verifica del significato dei comandi

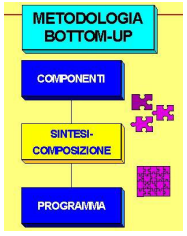
Esempi
"Il gatto mangia il topo": semanticamente e sintatticamente corretto.
"Il computer mangia il topo": sintatticamente corretto, semanticamente no.
"15" in numeri romani è "IV" se considero numeri singoli o "XV" se considero quindici → semantica.

I click del mouse sono comandi che fanno parte di un **linguaggio grafico**: i linguaggi grafici sono nati prima degli altri (umani hanno lasciato disegni rupestri, poi hanno cominciato ad utilizzare simboli per identificare un linguaggio che compone scritte leggibili). Nel **doppio click del mouse** c'è la **variabile tempo**.

0.2 - IL RUOLO DEL LINGUAGGIO e SPAZIO CONCETTUALE

La **CAPACITÀ ESPRESSIVA** di un linguaggio ne fa un potente suggeritore di **concetti e metodi di soluzione**: es. da Java è facile imparare C# perché i concetti sono gli stessi, hanno solo una sintassi diversa.

METODOLOGIE DI RISOLUZIONE DI UN PROBLEMA:



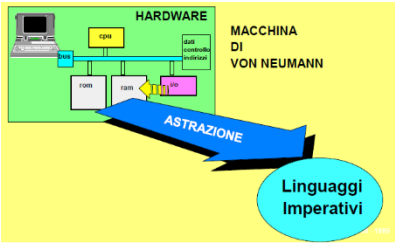
- **MODELLO A CASCATA**: procedimento lineare in cui analizzo il problema, progetto la soluzione, la implemento e infine la testo
- **TOP-DOWN**: scompongo problema in mini problemi finché non ho trovato un'unità risolvibile singolarmente
- **BOTTOM-UP**: compongo il mio progetto partendo dalla parte più piccola fino ad arrivare al programma completo

La soluzione generalmente viene pensata sui concetti di uno specifico linguaggio o categoria di linguaggi: non penserò ad una soluzione ad oggetti se ho sempre avuto a che fare con linguaggi come il C o il Pascal.

L'insieme di CONCETTI E METAFORE introdotti da un linguaggio costituisce il suo SPAZIO CONCETTUALE: avendo ogni linguaggio il suo spazio concettuale ha senso che si evolvano continuamente i linguaggi esistenti e che se ne creino di nuovi. In questo modo, in base al problema che devo risolvere, so se utilizzare un linguaggio o un altro.

0.3 - LE SORGENTI DEI LINGUAGGI

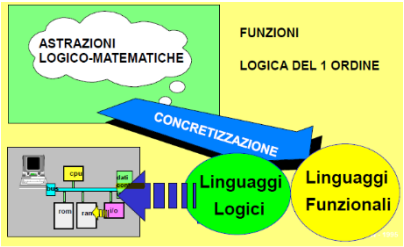
0.3.1 - LA MACCHINA HARDWARE



La **macchina hardware** è la base di tutti i linguaggi imperativi: **la CPU è un soldatino** che semplicemente esegue gli ordini che gli dai, lui da solo non decide nulla. Questo approccio va bene finché hai problemi che possono essere articolati in una serie di ordini: se ci sono compiti simbolici, non funziona più.

Es: per calcolare la derivata in un punto no problem. Ma se volessi calcolare la derivata simbolica di x^2? Mi deve venire fuori 2x, ma con la variabile x non è immediato.

0.3.2 - LE FUNZIONI



Prima di Von Neumann e dell'informatica, si è puntato sulla scienza e sulla matematica con le astrazioni basandosi sul **concetto di funzione**.
Esiste una famiglia di linguaggi che prende origine da questa idea, dimenticandoci di come è fatta la macchina al di sotto: i **LINGUAGGI FUNZIONALI**.
Si parte da un linguaggio funzionale, dopodiché componendo diverse funzioni si arrivano a comporre i teoremi.

0.3.3 - LA LOGICA

Su quella base si costruiscono i **LINGUAGGI LOGICI**, come i **DICHIARATIVI**: sono **regole che vengono specificate** e poi si immagina di avere sotto una macchina che deve scegliere quale regola usare per risolvere un problema con la possibilità di tornare indietro e ritentare; si delega il problema.

Esempio
Confronto tra approccio imperativo (Java) e approccio dichiarativo (Prolog) per fare append di due liste: nel caso imperativo ho un'ossessione del controllo, devo aver previsto TUTTI i casi di fallimenti e SOLO DOPO aver superato i controlli, posso fare le mie operazioni. Nel caso dichiarativo invece semplicemente indico delle regole e delego il controllo: mi concentro sull'obiettivo, non su come arrivarci.

0.4 - STILI DI PROGRAMMAZIONE

Ogni modello computazionale promuove uno specifico stile di programmazione: l'adozione di uno stile ha conseguenze sulle proprietà dei programmi, sulla metodologia di soluzione dei problemi, sulle caratteristiche dei sistemi software (struttura, efficienza, modificabilità, manutenibilità).
Ma perché usare solo un paradigma dentro ad un'applicazione? Perché non usare più linguaggi? Forse si lavora meglio utilizzando il meglio di più programmi → nasce il **MULTI-PARADIGM PROGRAMMING**: si fanno **coesistere modelli computazionali diversi in una stessa applicazione** prendendo il meglio di ognuno. Attenzione però, è importante **gestire bene i confini fra i diversi modelli**, altrimenti c'è rischio di inquinamento.

BLENDED LANGUAGES: oggi si inseriscono sempre più spesso elementi, approcci e concetti di derivazione funzionale in linguaggi di tradizione imperativa (a oggetti) ottenendo a volte miscele di linguaggi incoerenti.

L'obiettivo è prendere il meglio di entrambi i mondi:

- Strutture dati immodificabili
- Funzioni come FIRST-CLASS ENTITIES: chiusure, lambda…
- Meno controllo = minor accoppiamento, leggibilità, manutenibilità

1 - LINGUAGGI, MACCHINE ASTRATTE, TEORIA DELLA COMPUTABILITÀ

1.1 - ALGORITMI, PROGRAMMI, SISTEMI FORMALI

ALGORITMO: sequenza **finita** di mosse che risolve **in un tempo finito** una classe di problemi.
CODIFICA: descrizione dell'algoritmo tramite un **insieme ordinato di frasi** di un linguaggio di programmazione che specificano le **azioni** da svolgere.
PROGRAMMA: testo scritto in accordo alla **sintassi** e alla **semantica** di un linguaggio di programmazione. **Un programma NON è un algoritmo se non termina**: i programmi come i SO non devono terminare.

Si presuppone l'esistenza di un **AUTOMA ESECUTORE** (macchina astratta capace di eseguire le azioni dell'algoritmo): prende in ingresso dei dati che qualcuno gli fornisce, utilizza un algoritmo e dopo aver eseguito l'algoritmo sui dati ottiene determinati risultati.

L'automa esecutore:

- Deve comprendere l'algoritmo dato, quindi deve **interpretare il linguaggio macchina**
- Deve avere **VINCOLI DI REALIZZABILITÀ FISICI**, non si può barare e usare "infinite viti":
  - Deve essere costituito da un **numero finito di elementi**
  - Input/Output devono essere denotabili con un **insieme finito di simboli**

Concetto di **COMPUTABILITÀ:**

- **APPROCCIO A GERARCHIA DI MACCHINE ASTRATTE**: dagli ASF alla Macchina di Turing
- **APPROCCIO FUNZIONALE**: fondato sul concetto di funzione matematica
- **SISTEMI DI RISCRITTURA**: descrivono l'automa come un insieme di regole di riscrittura/inferenza

La nozione di **SISTEMA FORMALE** corrisponde ad una formalizzazione rigorosa e completa della nozione di sistema assiomatico. Gli esempi più comuni di sistemi formali sono le teorie del primo ordine.

1.2 – GERARCHIA DI MACCHINE ASTRATTE

Considereremo, nell'ordine:
MACCHINA COMBINATORIA < AUTOMI A STATI FINITI < ALTRE MACCHINE... < MACCHINA DI TURING

Si definisce una gerarchia perché non ha senso "sparare ad un topo con il bazooka", ha invece senso utilizzare uno strumento tanto potente quanto basta per risolvere il problema che abbiamo davanti. Se poi neanche la macchina più "potente" riesce a risolvere tale problema, allora possiamo dire che esso non è risolubile.

1.2.1 - LA MACCHINA BASE

$\langle I, O, mfn \rangle$

dove
- I = insieme finito dei **simboli di ingresso**
- O = insieme finito dei **simboli di uscita**
- $mfn: I \rightarrow O$ *(funzione di macchina)*

Essa è definita da una **tripla**. Una funzione su un numero di ingressi finito e un numero di uscite finito, ha finite soluzioni. Otteniamo quindi la **tabella della verità.**
Tutte le **porte logiche** sono fatte così e **non hanno memoria.**
La macchina base è un **DISPOSITIVO PURAMENTE COMBINATORIO**, non riconosce le stringhe né le somme di numeri forniti in successione. **Questa macchina si può usare per tutti i problemi che non richiedono l'utilizzo della memoria.**

1.2.2 - L'AUTOMA A STATI FINITI

È definito da una **quintupla** ed è l'evoluzione della macchina base in quanto ha degli **stati**, quindi può lavorare con algoritmi che richiedono l'uso della memoria.

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

dove
- I = insieme finito dei **simboli di ingresso**
- O = insieme finito dei **simboli di uscita**
- S = insieme finito degli **stati**
- $mfn: I \times S \rightarrow O$ *(funzione di macchina)*
- $sfn: I \times S \rightarrow S$ *(funzione di stato)*

La funzione di stato calcola lo stato futuro in cui la macchina va a finire. Lo stato rappresenta la storia: 10 stati sono 10 possibili "ricordi", configurazioni.
Limite computazionale: è un dispositivo a **MEMORIA FINITA**, inadatto a problemi che non consentano di limitare a priori la lunghezza delle sequenze da ricordare. Potrei barare mettendo un numero altissimo di stati, ma devo considerare se l'input, ad esempio, è dato da un umano o da una macchina. Nel secondo caso, la mia soluzione crollerebbe miseramente, perché il limite verrebbe superato sicuramente.

1.2.3 - LA MACCHINA DI TURING

La macchina più potente di tutte: con questa viene introdotto un **nastro** (carta, magnetico, RAM…) **come supporto di memorizzazione esterno**. È definita da una quintupla.

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

dove
- A = insieme finito dei **simboli di ingresso e uscita**
- S = insieme finito degli **stati** *(uno dei quali è **HALT**)*
- $mfn: A \times S \rightarrow A$ *(funzione di macchina)*
- $sfn: A \times S \rightarrow S$ *(funzione di stato)*
- $dfn: A \times S \rightarrow D = \{Left, Right, None\}$ *(funz. di direzione)*

Il nastro non è infinito, ma è ILLIMITATAMENTE ESPANDIBILE perché se finisce si può sostituire con uno nuovo: questa è la potenzialità incredibile della macchina.
La macchina ha una testina di lettura e scrittura che può spostarsi dove vuole nel nastro e sovrascrivere "celle" già occupate. Per risolvere un problema con questa macchina bisogna:

- Definire rappresentazione dati di partenza e di uscita
- Definire il comportamento tramite le funzioni mfn(), sfn(), dfn() in modo che scrivano la soluzione

Essa ha uno **stato particolare** detto **HALT**: quando arriva in questo stato, essa **si ferma**. L'importante è che sia stato scritto il risultato sul nastro prima dell'arrivo nello stato HALT.
Quello che potrebbe capitare è l'**infinito loop della macchina**: essa infatti potrebbe non arrivare mai nello stato di HALT. Supponendo perciò che l'algoritmo che le abbiamo dato sia corretto (altrimenti si tratta semplicemente di un errore del programmatore), allora vuol dire che questo **algoritmo non** è **computabile**.

Esempio: riconoscimento di frasi palindrome binarie
Per farlo, leggeremo il primo carattere, lo memorizzeremo e lo segneremo con un rombo. Poi leggeremo l'ultimo numero: se i due non coincidono, scriveremo un errore e termineremo, altrimenti metteremo un rombo e proseguiremo. Una volta consumati tutti i bit, la frase sarà palindroma e la macchina dovrà fermarsi.

Potremmo programmare la macchina con questa tabella, indicando gli stati e gli ingressi, guardando poi dove vanno a finire i prossimi stati e dove va a spostarsi la testina. In questo modo abbiamo espresso tutto ciò che deve fare la macchina e tutto ciò che ci aspettiamo che succeda.

Questo tipo di meccanismo è ciò che ha permesso la creazione della Random Access Memory (RAM): la testina può spostarsi avanti e indietro delle celle che vuole e sovrascrivere quelle che vuole.

mfn *iniziale*

	s0	s1	s2	s3	s4	s5
0	0	♦	0	0	♦	☹
1	1	♦	1	1	☹	♦
♦	♦	☺	♦	♦	☺	☺

sfn

	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

dfn

	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

Esistono **macchine multi-nastro** che possono risultare più veloci della MdT, ma sostanzialmente sotto hanno comunque la stessa macchina, quindi per ora non esistono macchine più potenti della MdT e ce lo dice anche la **TESI DI CHURCH-TURING**.

TESI DI CHURCH-TURING
Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella risolta dalla Macchina di Turing.

È una tesi e non un teorema per il semplice fatto che ciò che dice fino ad ora non è stato dimostrato, ma, nonostante gli svariati tentativi, non è nemmeno stato dimostrato il contrario.

1.2.4 - PUSH DOWN AUTOMATON (PDA)

Non sempre è necessaria una MdT, anzi, molto spesso basta un modello di memoria più limitato come il PDA o **MACCHINA A STACK**: è una macchina costruita come uno stack, molto efficiente e con un nastro espandibile da una parte sola. L'unico vincolo è quello di accedere solo alla cella al top.

1.2.5 - LA MACCHINA DI TURING UNIVERSALE (UTM)

Il problema principale della macchina di Turing è che una volta definita, essa è specifica per il suo scopo perché conosce il solo algoritmo che le viene fornito e quindi può essere una calcolatrice, una lavastoviglie, un orologio, ma non sarà tutti e tre insieme.

Per costruire una MACCHINA DI TURING UNIVERSALE bisognerebbe mettere l'algoritmo sul nastro: in questo modo la macchina può leggere qualsiasi algoritmo e quindi essere generica e svolgere qualsiasi compito. Diventa così un **LOADER**.

Per poter eseguire l'algoritmo dato nel nastro è necessario aver descritto l'algoritmo richiesto tramite un linguaggio, quindi occorre che la **UTM** sia un **interprete del linguaggio**: arriviamo così all'automa esecutore di cui è rappresentata l'immagine nel capitolo 1.1. La UTM modella quindi il concetto di elaboratore di uso generale le cui **funzioni principali** sono:

- **Fetch**: va a cercare le istruzioni sul nastro
- **Decode**: le interpreta
- **Execute**: le esegue

UTM versus MACCHINA DI VON NEUMANN

Macchina di Turing universale	Macchina di Von Neumann
Legge/scrive un simbolo dal/sul **nastro**	Legge/scrive da/su una **memoria** (RAM, ROM…)
Transita in un altro **stato interno**	Passa a diverse **configurazioni dei registri CPU**
Si sposta sul nastro di 1+ posizioni	**Può scegliere la cella di memoria su cui operare**
Ha solo **una memoria interna**, è pura computazione	Ha anche tutta la parte di **interazione col mondo esterno**

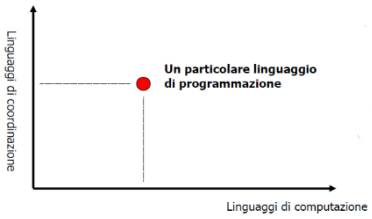
Fondamentalmente la differenza principale è la parte di I/O.

1.3 - COMPUTAZIONE E INTERAZIONE

La computazione e l'interazione sono **dimensioni ortogonali** espresse da due linguaggi distinti:

- **Il linguaggio di Computazione:** come elaborare un dato, quali primitive usare (es. for, while, if…)
- **Il linguaggio di Coordinazione:** come leggere/scrivere un dato (es. printf, scanf…) da/verso esterno
    - **Il linguaggio di Comunicazione:** come vengono formattati i dati, perché se parliamo due lingue diverse non riesco a leggerlo e non riesco ad elaborare i dati

Bisogna accoppiare le due dimensioni per poter capire il linguaggio con cui si esprimono le istruzioni e il linguaggio con cui si trasferiscono i dati. Qualunque linguaggio di programmazione occupa un punto in quel piano.



Tutti i linguaggi moderni non includono le librerie con le istruzioni di I/O, bisogna includerle. La comunicazione basata su post-it o su messaggi lasciati sulla lavagna ha un qualcosa di importantissimo e cioè il **disaccoppiamento temporale**; invece se usiamo una scanf/printf, è come una telefonata, abbiamo bisogno che l'altro sia presente in quel preciso momento.
Io potrei prendere il C e inventare una libreria chiamata "PostLibrary", in cui quando devo leggere o scrivere posso chiamare le funzioni "writePost" o "readPost".

1.4 - INTRODUZIONE ALLA TEORIA DELLA COMPUTABILITÀ

Secondo la Tesi di Church-Turing, **se neanche la macchina di Turing riesce a risolvere un problema, quel problema è irresolubile.** Supponendo che l'algoritmo sia corretto, la MdT non riesce a risolverlo se non risponde proprio o se non si ferma. Se la macchina invece si ferma e produce un risultato, allora è un problema risolubile.

Un **Problema Risolubile** è un problema la cui soluzione può essere espressa sa una MdT. Però **la MdT computa FUNZIONI e non PROBLEMI**, quindi è necessario associare ad un problema una funzione.

Si definisce **FUNZIONE CARATTERISTICA DI UN PROBLEMA** la funzione ***fp: X → Y*** con X insieme dei dati in ingresso, Y insieme delle risposte corrette e P il problema.
Un problema irresolubile diventa così sinonimo di funzione caratteristica non computabile.

Dato un **problema P** e detti
- **l'insieme X dei suoi dati di ingresso**
- **l'insieme Y delle risposte corrette**

si dice ***funzione caratteristica*** **del problema P**

$f_P: X \rightarrow Y$

la funzione f_P che associa a ogni dato d'ingresso $x \in X$ la corrispondente risposta corretta $y \in Y$.

FUNZIONE COMPUTABILE

Una funzione f: A→B è *computabile* se esiste una Macchina di Turing che
- data sul nastro una rappresentazione di $x \in A$

dopo un numero finito di passi
- produce sul nastro una rappresentazione di $f(x) \in B$

Una funzione caratteristica è **computabile se dopo un numero finito di passi produce sul nastro un risultato, quindi se la macchina si ferma.**

Non tutte le funzioni sono computabili, anche se sono definibili: bisogna allora confrontare le funzioni che possiamo definire con quelle che una MdT può computare.

Le **FUNZIONI COMPUTABILI SU N** sono **funzioni sui numeri naturali *f: N→N***. Da qui in poi considereremo solo questo tipo di funzioni, ma non è limitativo poiché tutte le informazioni sono finite quindi possono essere codificate da una collezione di numeri naturali che può essere espresse con un unico numero naturale tramite il **PROCEDIMENTO DI GÖDEL.**

1.4.1 – PROCEDIMENTO DI GÖDEL

- siano $N_1, N_2, \dots N_k$ i numeri naturali dati
- siano $P_1, P_2, \dots P_k$ i primi k *numeri primi*
- sia ***R*** un ***nuovo numero naturale*** così definito:

$$R ::= P_1^{N1} \cdot P_2^{N2} \cdot \dots \cdot P_k^{Nk}$$

Grazie all'unicità della scomposizione in fattori primi, R rappresenta univocamente, in modo compatto, la collezione originale $N_1, N_2, \dots N_k$.

Obiettivo: data una collezione di numeri naturali, esprimerla con un unico numero naturale.

L'insieme delle funzioni definibili su N non è enumerabile, mentre l'insieme delle MdT che corrispondono alle funzioni computabili lo è: la **gran parte delle funzioni definibili non è computabile**!
Di buono c'è che **molte di queste non ci interessano** perché le uniche che ci interessano sono quelle definibili da un linguaggio basato su un **alfabeto finito di simboli**, che sono un insieme enumerabile.
Purtroppo esistono comunque funzioni definibili con un alfabeto finito, ma non computabili, cioè esistono **PROBLEMI IRRISOLUBILI**. Per dimostrarlo, basta trovarne uno, come il problema dell'HALT della MdT.

1.4.2 - DIMOSTRAZIONE: PROBLEMA DELL'HALT DELLA MDT

Problema dell'HALT della MdT: stabilire se una MdT detta T con un ingresso generico X, si ferma oppure no. Questo problema è perfettamente definito, ma in genere non è computabile.

- Siano:
  - **M** l'insieme di tutte le Macchine di Turing
  - **X** l'insieme di tutti i possibili ingressi
- Siano inoltre:
  - $x \in X$ un generico dato di ingresso
  - $m \in M$ una generica Macchina di Turing
- Indichiamo infine con:
  - $\perp$ un *risultato indefinito*, dovuto a una Macchina di Turing che non risponde (entra in un ciclo infinito)

La funzione caratteristica f_{HALT} di questo problema può essere così definita:

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

La funzione f_{HALT} è ben definita, ma nel caso generale, non è computabile perché tentare di calcolarla conduce ad un assurdo. Supponiamo comunque che esista una funzione del genere, allora **se f_{HALT} è computabile, deve esistere una MdT capace di calcolarla**. Definiamo allora una nuova funzione g_{HALT} che sia funzione soltanto della generica MdT n:

- **Se g si ferma** e risponde 1
  → **la MdT** n con ingresso n **non si ferma**
- **Se g non si ferma** ed entra in un loop infinito
  → **la MdT** n con ingresso n **si ferma**

$$g_{HALT}(n) = \begin{cases} 1, \text{ se } f_{HALT}(n,n) = 0 & \text{La MdT } \mathbf{n} \text{ con ingresso } \mathbf{n} \textit{ non} \text{ si ferma} \\ \perp, \text{ se } f_{HALT}(n,n) = 1 & \text{La MdT } \mathbf{n} \text{ con ingresso } \mathbf{n} \text{ si ferma} \end{cases}$$

Nel caso particolare in cui $n=n_g$, cioè quando l'ingresso è proprio il numero che rappresenta la MdT detta TG che calcola la funzione g (che è come dare in ingresso alla macchina se stessa), si genera un assurdo: sostituendo, si ottiene la funzione $g_{HALT}(n_g)$.
In questa funzione, la prima condizione dice che TG si ferma (perché g dà come risultato 1) se e solo se TG non si ferma (perché f vale 0), nella seconda condizione viceversa → le due funzioni si contraddicono, è un loop.

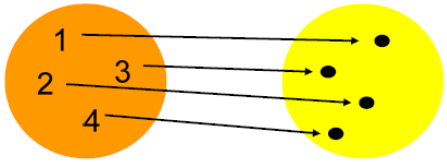
$$g_{HALT}(n_g) = \begin{cases} 1, \text{ se } f_{HALT}(n_g,n_g) = 0 \\ \perp, \text{ se } f_{HALT}(n_g,n_g) = 1 \end{cases}$$

Conclusione: la **funzione caratteristica di questo problema non è computabile nel caso generale**.
Decidere se **una data MdT con un generico ingresso si fermi oppure no è un PROBLEMA INDECIDIBILE.**

1.5 - GENERABILITÀ E DECIDIBILITÀ

Un linguaggio è un insieme di frasi, quindi ci interessa indagare il problema della generabilità vs. decidibilità di un insieme. L'analisi matematica introduce il concetto di **insieme numerabile** per cui esiste una funzione di corrispondenza con i naturali. Noi vogliamo che tale funzione sia **computabile**, affinché l'insieme sia effettivamente generabile da una MdT: ci servono **funzioni numerabili che lavorino con insiemi ricorsivamente enumerabili**.

INSIEME NUMERABILE: insieme i cui elementi possono essere contati, ha una funzione biettiva ***f: N → S*** che mette in corrispondenza i numeri naturali con gli elementi dell'insieme.



INSIEME RICORSIVAMENTE NUMERABILE se tale funzione è computabile, cioè se una MdT può computare una funzione generando uno ad uno gli elementi dell'insieme. Se chiedo perciò alla MdT di darmi l'elemento numero X, deve essere in grado di fornirmelo senza inlooparsi.

1.5.1 – APPARTENENZA ALL'INSIEME

Per **decidere se un dato elemento appartiene a un insieme** ricorsivamente enumerabile, la MdT potrebbe generare uno ad uno gli elementi dell'insieme, fermandosi se e quando lo trova. Ma se non lo trova non sapremo se l'elemento appartiene o no all'insieme: la MdT va in loop e non risponderà mai.
Questo è un **INSIEME SEMI-DECIDIBILE**: indica che **si riesce a decidere se appartiene (in positivo), ma non se non appartiene.**

Esempio - MdT che può generare tutti i numeri pari
Se chiedo il numero 32 va tutto bene, se le chiedo il numero 33, lei comincia a generare tutti i numeri pari possibili immaginabili non arrivando mai al 33. In questo caso abbiamo a che fare con un insieme semi-decidibile.

Un insieme **S** è **DECIDIBILE** (o ricorsivo) se la sua funzione caratteristica è computabile:

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

Deve quindi esistere una MdT capace di rispondere sì o no, senza entrare mai in un ciclo infinito.
Per avere invece un **insieme decidibile**, dobbiamo avere una **MdT** che non solo sappia fare la sua funzione, ma sappia fare anche la **complementare**.
Es. MdT di prima deve saper generare i pari e i dispari. Se le chiedo il 32 me lo dà. Se le chiedo il 33, non si blocca all'infinito, ma genera i dispari e vede che appartiene all'insieme non pari, cioè dispari.

TEOREMA 1
Se un insieme è *decidibile* è anche *semidecidibile* ma non viceversa

TEOREMA 2
Un insieme S è *decidibile* *se e solo se*
• sia *S*
• sia il suo complemento N-*S*
sono *semidecidibili.*

Questi teoremi sono importanti perché i **linguaggi di programmazione sono costruiti a partire da un alfabeto finito e da un insieme delle sue frasi lecite,** ma non basta che tale insieme sia semi-decidibile e cioè generato: è indispensabile poter anche decidere se una frase è giusta o sbagliata senza entrare in un loop infinito, quindi **occorre che l'insieme sia decidibile.**

2 - LINGUAGGI E GRAMMATICHE

2.1 - DEFINIZIONI

LINGUAGGIO: insieme di PAROLE e METODI di COMBINAZIONE delle parole usate e comprese da una comunità di persone.
Le parole sono elencate nel vocabolario: se lì dentro non c'è vuol dire che ciò che ho scritto non esiste in quel linguaggio. Non puoi mettere le parole nell'ordine che ti pare: servono regole grammaticali. Perché un linguaggio sia efficace, qualcuno deve usarlo, non come l'esperanto.

Per ciò che dobbiamo fare noi, **serve una nozione di linguaggio più precisa**. Pensiamo al linguaggio come ad un sistema formale: per progettarlo bene occorre tenere conto di diversi fattori altrimenti creeremmo un linguaggio con dei buchi e con dell'incoerenza. In particolare:

- Evitare ambiguità dei linguaggi naturali
- Deve prestarsi a descrivere processi computazionali meccanizzabili
- Deve aiutare a stabilire proprietà

2.1.1 - SINTASSI E SEMANTICA

SINTASSI: insieme di regole formali per la scrittura di programmi in un linguaggio che dettano le **modalità per costruire frasi corrette**. È espressa tramite notazioni formali come **BNF e diagrammi sintattici**.

SEMANTICA: insieme dei significati da attribuire alle frasi costruite nel linguaggio. **È esprimibile:**

- **A parole** (quando spieghi un concetto utilizzando sinonimi e altre frasi)
- **Mediante azioni → SEMANTICA OPERAZIONALE** (per spiegare la moltiplicazione, usi l'addizione)
- **Mediante funzioni matematiche → SEMANTICA DENOTAZIONALE** (per spiegare un'operazione, la definisci in modo astratto come in analisi)
- **Mediante formule logiche → SEMANTICA ASSIOMATICA**

Una frase può essere sintatticamente corretta ma può non avere significato.

2.1.2 – INTERPRETE vs COMPILATORE

INTERPRETE per un linguaggio L	COMPILATORE per un linguaggio L
INPUT: singole frasi di L	**INPUT**: intero programma scritto in L
ESECUZIONE una per volta	**RISCRITTURA** di un altro linguaggio
OUTPUT: valutazione della frase	**OUTPUT**: riscrittura della "macro-frase"

La **differenza tra interprete e compilatore** è che **l'interprete verifica se la parola è corretta o no**, mentre il **compilatore riscrive** un intero **programma in un altro linguaggio** a lui più comprensibile.

2.1.3 – ANALISI LESSICALE, SINTATTICA, SEMANTICA

ANALISI LESSICALE: **si individuano singole parole (TOKEN) di una frase tramite uno SCANNER/LEXER.**
Lo **scanner/lexer** prende le sequenze di caratteri e individua i token in base ai separatori tipici del linguaggio. Poi, in base ai caratteri che trova, usa delle categorie: es. una parola ha un ruolo diverso da un ";".

ANALISI SINTATTICA: **si verifica che una frase (SEQUENZA DI TOKEN) rispetti le REGOLE GRAMMATICALI utilizzando un PARSER** che genera una rappresentazione interna della frase, solitamente sotto forma di un opportuno **albero**.

ANALISI SEMANTICA: si determina il SIGNIFICATO DI UNA FRASE usando un ANALIZZATORE SEMANTICO che controlla la coerenza logica della frase. Avviene dopo l'analisi sintattica.

SIGNIFICATO DI UNA FRASE: **associazione tra la frase e un concetto nella nostra mente**.

Nella nostra mente deve esserci una funzione che associa ad ogni frase un concetto: es. con 2+3 noi diamo per scontato che nella nostra mente sia presente il concetto di somma e automaticamente diventa 5. La funzione di associazione deve dare significato a simboli, parole e frasi.

2.1.4 – DESCRIZIONE LINGUAGGIO

Formalmente, definendo:

- **Alfabeto** = insieme finito (e abbastanza ridotto) e non vuoto di simboli atomici, es. A = {a, b}
- **Stringa** = sequenza di simboli, i.e. elemento del prodotto cartesiano A^n (stringhe con n simboli)
- **Lunghezza di una stringa** = numero di simboli che la compongono
- **Stringa vuota ε** = stringa di lunghezza zero (con $A^0=\varepsilon$)

- **LINGUAGGIO L** su un alfabeto A = un insieme di stringhe su A
- **Frase di un linguaggio** = stringa appartenente al linguaggio
- **Cardinalità di un linguaggio** = numero delle frasi di un linguaggio, può essere finita o infinita
  - L1 = {aa, baa} → cardinalità finita
  - L2 = {a^n, n>0} → cardinalità infinita
- **CHIUSURA A* di un alfabeto A** = insieme **infinito** delle stringhe composte con i simboli di A
- **CHIUSURA POSITIVA A^+ di un alfabeto A** = insieme **infinito** delle stringhe **non nulle** (A* - {ε})

2.1.5 – GRAMMATICA FORMALE

Per esprimere la chiusura A* di un alfabeto A, specialmente se infinito, non si possono elencare tutte le frasi, ma serve una NOTAZIONE → si utilizza la **GRAMMATICA FORMALE.**
GRAMMATICA: notazione formale con cui esprimere in modo *rigoroso* la sintassi di un linguaggio.

Simboli terminali VT = caratteri o stringhe dell'alfabeto.
Simboli non terminali VN = meta-simboli che rappresentano categorie sintattiche.
VT ∩ VN = ∅ → VT e VN insiemi disgiunti.
VT ∪ VN = **Vocabolario** della grammatica.

Una grammatica è una *quadrupla* ⟨VT,VN,P,S⟩ dove:
- **VT è un *insieme finito di simboli terminali***
- **VN è un *insieme finito di simboli non terminali***
- **P è un *insieme finito di produzioni*, ossia di *regole di riscrittura*** $\alpha \rightarrow \beta$ *dove* α *e* β *sono stringhe:* $\alpha \in V^+$, $\beta \in V^*$
  - **ogni regola esprime una trasformazione lecita che permette di scrivere, *nel contesto di una frase data,* una stringa β al posto di un'altra stringa α.**
- **S è un particolare *simbolo non-terminale* detto *simbolo iniziale* o *scopo* della grammatica.**

Alcune convenzioni:

- **VT** si indicano con le **minuscole**
- **VN** si indicano con le **maiuscole**
- Le **lettere greche** indicano **stringhe mixed** di terminali e meta-simboli
- **Produzione α → β =** una stringa non nulla α viene riscritta sottoforma della nuova stringa β

FORMA DI FRASE / SENTENTIAL FORM: **qualsiasi stringa comprendente sia simboli terminali VT che meta-simboli VN, ottenibile dallo scopo applicando 1+ regole di produzione**. È un **prodotto intermedio**: alcune parti sono già finali, altre no.

FRASE: comprende solo simboli terminali, è un **prodotto finale** in cui non si può trasformare altro.

2.1.6 - DERIVAZIONE

Siano α, β due stringhe $\in (VN \cup VT)^*, \alpha \neq \varepsilon$.

DERIVAZIONE DIRETTA: β deriva direttamente da α (α → β), se:

- Le stringhe α, β si possono decomporre in: α = η**A**δ e β=ηγδ
- Esiste la produzione A → γ

DERIVAZIONE: β deriva da α, anche non direttamente, se ∃ una sequenza di N derivazioni *dirette* che da α producono β, cioè $\alpha = \alpha 0 \rightarrow \alpha 1 \rightarrow \alpha 2 \rightarrow \cdots \rightarrow \alpha N = \beta$

SEQUENZA DI DERIVAZIONE: sequenza di passi che producono una forma di frase σ dallo scopo S, in particolare:

- $S \Rightarrow \sigma$ significa che σ deriva da S con **una sola applicazione** di produzioni (in un solo passo)
- $S \overset{+}{\Rightarrow} \sigma$ significa che σ deriva da S con **1+ applicazioni** di produzioni (1+ passi)
- $S \overset{*}{\Rightarrow} \sigma$ significa che σ deriva da S con **0+ applicazioni** di produzioni (0+ passi)

Data una grammatica G, si dice perciò **LINGUAGGIO L_G GENERATO DA G**, l'insieme di frasi derivabili dal simbolo iniziale S applicando le produzioni P, cioè:

$$L_G = \{ s \in VT^* \mid S \overset{*}{\Rightarrow} s \}$$

Esempio

Il linguaggio L = { $a^n b^n$, n>0 } può essere descritto dalla grammatica G = ⟨VT,VN,P,S⟩ dove:

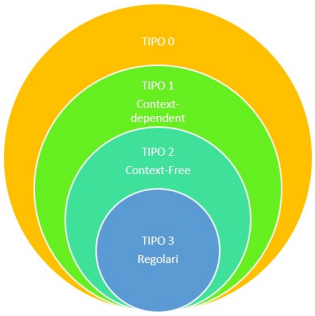
- VT = { a, b}
- VN = { F }
- S ∈ VN = F
- P = {
  F → a b
  F → a F b
  }

- La prima regola stabilisce che F può essere riscritto come ab: è la frase più corta di L.
- La seconda regola stabilisce che lo scopo F può essere riscritto come aFb; data la presenza di F nella forma di frase, è possibile proseguire con un nuovo passo generativo – di nuovo scegliendo *una qualsiasi* delle due regole:
  - se si sceglie la prima, si avrà aabb
  - se si sceglie la seconda, si avrà aaFbb, che apre la porta a un terzo passo.. e così via.
- Il linguaggio contiene dunque infinite frasi, tutte della forma aa…bb con egual numero di a e b.

2.1.7 - GRAMMATICHE EQUIVALENTI

Due grammatiche G1 a G2 si dicono equivalenti se generano lo stesso linguaggio.
È possibile, infatti, che la prima grammatica che ti viene in mente per esprimere un linguaggio sia esageratamente difficile o sbagliata. Supponendo che non sia sbagliata, si può elaborare e produrre una grammatica più semplice ed equivalente.
In generale però, **stabilire se due grammatiche sono equivalenti è un problema indecidibile.**
Grammatiche di diversa struttura comportano **linguaggi con diverse proprietà** e implicano **automi di diversa potenza computazionale** per riconoscere tali linguaggi.

2.2 - CLASSIFICAZIONE DI CHOMSKY



Esistono 4 tipi di grammatiche in base alla **<u>struttura delle produzioni</u>**:

<u>TIPO 3</u> (**grammatica regolare**): caso particolare della context-free

- **<u>TIPO 2</u>** (**context free**): caso particolare di quella context-dependent
- **<u>TIPO 1</u>** (**context-dependent**): caso particolare grammatiche di tipo 0
- **<u>TIPO 0</u>**: la più generica

Siccome nelle grammatiche di tipo 2 e 3 si può generare la stringa vuota, tale gerarchia vale se anche in quelle di tipo 1 si ammette la produzione S→ ε

2.2.1 - TIPO 0

- **Tipo 0:**
  nessuna restrizione sulle produzioni

In particolare, le regole possono specificare riscritture che *<u>accorciano</u> la forma di frase corrente.*

Nelle grammatiche di tipo 0 non ci sono regole da rispettare, si possono usare **tutte le derivazioni che si vogliono.** In queste grammatiche si possono sostituire le regole in diverso ordine e ottenere risultati diversi.
Sono **quelle più difficili da riconoscere** e molto spesso sono "orribili"; inoltre, possono esserci regole che sostituiscono più simboli con meno simboli o con la stringa vuota, accorciando le frasi.

Esempio
S → aSBC CB → BC SB → bF FB → bF FC → cG GC → cG G → ε
Possibile derivazione: S → a**SB**C → a**bF<u>C</u>** → ab**<u>cG</u>** → abcε = abc (risultato finale è più corto)

2.2.2 - TIPO 1 (CONTEXT-DEPENDENT)

In questo tipo di grammatiche si può **sostituire qualsiasi simbolo <u>purché non venga cancellato</u>** ($\alpha \neq \varepsilon$).
Esse sono **dipendenti dal contesto**: va bene solo se trovi il meta-simbolo da sostituire tra due stringhe α e β.

- **Tipo 1 (*dipendenti dal contesto*):**
  produzioni vincolate alla forma:

$$\beta\, A\, \delta \rightarrow \beta\, \alpha\, \delta$$

con $\beta, \delta, \alpha \in (VT \cup VN)^*$, $A \in VN$, $\alpha \neq \varepsilon$

Quindi, **A** può essere sostituita da α solo *<u>nel contesto $\beta A \delta$</u>*
Le riscritture *<u>non accorciano mai</u> la forma di frase corrente.*

Esempio **-** Se faccio "search & replace" e voglio sostituire "qua" con "qui":

- "qua" diventa "qui"
- "quadro" non diventa "quidro" poiché dopo qua c'è un'altra stringa "dro"
- "alquanto" non diventa "alquinto" perché prima c'è "al" e poi c'è "nto"

Una **<u>DEFINIZIONE ALTERNATIVA</u>** equivalente (a parte la generazione della stringa vuota) prevede **produzioni della forma $\alpha \rightarrow \beta$ con $|\beta| \geq |\alpha|$.**

LA DEFINIZIONE ALTERNATIVA: $\alpha \rightarrow \beta$ con $|\beta| \geq |\alpha|$
esprime lo stesso concetto in modo *più pratico*,
ma non esplicita più l'idea di contesto.
Formalmente, essa *ammette produzioni vietate dalla definizione di Chomsky,* come ad esempio **BC → CB**
tuttavia, *esiste sempre una grammatica equivalente che rispetta la definizione di Chomsky,* ad esempio **BC → BD ; BD → CD ; CD → CB**
quindi *i due formalismi sono equivalenti*
[*purché* la definizione originale non venga arricchita ammettendo la produzione **S → ε**, che la definizione alternativa non può esprimere].

In questa definizione è permesso lo **scambio**, che Chomsky non permetterebbe, ma è solo un modo più compatto di esprimere l'applicazione di tre regole, quindi i due formalismi sono equivalenti.

Esempio

S → aBC | aSBC CB → DB DB → DC DC → BC

aB → ab bB → bb bC → bc cC → cc

La lunghezza del lato destro delle produzioni non è mai inferiore a quella del lato sinistro.

S → aBC | aSBC	$\beta = \varepsilon$	$\delta = \varepsilon$
CB → DB	$\beta = \varepsilon$	$\delta = B$
DB → DC	$\beta = D$	$\delta = \varepsilon$
DC → BC	$\beta = \varepsilon$	$\delta = C$
aB → ab	$\beta = a$	$\delta = \varepsilon$
bB → bb	$\beta = b$	$\delta = \varepsilon$
bC → bc	$\beta = b$	$\delta = \varepsilon$
cC → cc	$\beta = c$	$\delta = \varepsilon$

2.2.3 - TIPO 2 (CONTEXT FREE)

In questo caso non abbiamo più il vincolo della stringa vuota (***α* può essere *ε***) e la **grammatica è indipendente dal contesto**.

Esempio - "search & replace" con "pietro" e "paolo"

- "pietro" diventa "paolo"
- "san pietroburgo" diventa "san paoloburgo"

CASO PARTICOLARE: se α ha la forma "u" oppure "u B v" con $u, v \in VT^*$ e $B \in VN$, la **grammatica è lineare**.

- **Tipo 2 (*libere dal contesto*): *produzioni vincolate alla forma:***

  $A \rightarrow \alpha$

  con $\alpha \in (VT \cup VN)^*$, $A \in VN$

  Attenzione: non c'è più il vincolo $\alpha \neq \varepsilon$

  Qui A può *sempre* essere sostituita da α, *indipendentemente dal contesto*, giacché non esiste più l'idea stessa di contesto.

Qui inoltre **non è più permesso lo scambio** perché ho solo un simbolo e non due: un simbolo con cosa lo scambio? Con nulla.

2.2.4 - TIPO 3 (REGOLARE)

- **Tipo 3 (*grammatiche regolari*): *produzioni vincolate alle forme lineari:***

lineare a destra	*lineare a sinistra*
$A \rightarrow \sigma$	$A \rightarrow \sigma$
$A \rightarrow \sigma B$	$A \rightarrow B \sigma$

con $A, B \in VN$, e $\sigma \in VT^*$

Le produzioni di una grammatica di tipo 3 sono tutte vincolate alle forme **lineari a destra o a sinistra, non mischiate**. Ciò significa che le grammatiche lineari a destra, hanno tutti i simboli non terminali alla destra, quelle a sinistra il contrario. Qui ***σ* può essere *ε*.**

Queste grammatiche sono dette **REGOLARI**: in queste grammatiche **si può sempre, ed è spesso conveniente, trasformarle in forma strettamente lineare**: anziché sostituire la stringa di caratteri σ, si sostituire il singolo carattere a.

In questo modo noi **sappiamo sempre per cosa inizia una frase**, e vedremo che sarà molto più semplice per un riconoscitore analizzare grammatiche di questo tipo.

- **non più** $\sigma \in VT^*$ (**σ è una stringa di caratteri**)

lineare a destra	*lineare a sinistra*
$A \rightarrow \sigma$	$A \rightarrow \sigma$
$A \rightarrow \sigma B$	$A \rightarrow B \sigma$

- **ma bensì** $a \in VT$ (**a è un singolo carattere**)

lineare a destra	*lineare a sinistra*
$X \rightarrow a$	$X \rightarrow a$
$X \rightarrow a Y$	$X \rightarrow Y a$

Esempi

VT = { a, +, – }, VN = { S }

- **Grammatica G1** (lineare a sinistra: $A \rightarrow B\ y$, con $y \in VT^*$)
  $S \rightarrow a$ $S \rightarrow S + a$ $S \rightarrow S - a$
- **Grammatica G2** (lineare a destra: $A \rightarrow x\ B$, con $x \in VT^*$)
  $S \rightarrow a$ $S \rightarrow a + S$ $S \rightarrow a - S$
- **Grammatica G3** (G2 resa *strettamente lineare* a destra)
  $S \rightarrow a$ $S \rightarrow a\ A$ $A \rightarrow + S$ $A \rightarrow - S$
- **Grammatica G4** (lineare a destra e anche a sinistra)
  $S \rightarrow$ ciao
- **Grammatica G5** (G4 resa *strettamente lineare* a destra)
  $S \rightarrow c\ T$ $T \rightarrow i\ U$ $U \rightarrow a\ V$ $V \rightarrow o$

G1 e G2 costruiscono le stesse frasi, ma G1 li aggiunge in coda come quando scriviamo, G2 li aggiunge in testa.

Si può con la grammatica G3 renderla strettamente lineare facendo più passaggi: mi serve però un simbolo accessorio che è A.

2.2.5 - IL PROBLEMA DELLA STRINGA VUOTA

Nelle grammatiche di tipo 1 non sono ammesse stringhe vuote sul lato destro delle produzioni ($\alpha \neq \varepsilon$), mentre nelle grammatiche di tipo 2 e 3, sono ammesse: com'è possibile?

Esiste un teorema che afferma che **la sola differenza tra una grammatica di tipo 2 con o senza ε-rules è che il linguaggio generato dalla prima include la stringa vuota.**

TEOREMA
le produzioni di grammatiche di Tipo 2 (e quindi anche 3) possono sempre essere riscritte in modo da evitare la stringa vuota: al più, possono contenere la regola S → ε

TEOREMA
- Se G è una grammatica *context free* con produzioni della forma $A \rightarrow \alpha$, con $\alpha \in V^*$ (cioè, α può essere ε)
- allora *esiste una grammatica context free G'* che genera *lo stesso linguaggio L(G)* ma *le cui produzioni hanno o la forma* $A \rightarrow \alpha$, con $\alpha \in V^+$ (α *non è* ε) *oppure la forma* $S \rightarrow \varepsilon$, ed *S non compare sulla destra in nessuna produzione.*

La stringa vuota in effetti è molto utile perché permette di avere elementi *opzionali* nel linguaggio, caratteristica fondamentale nei linguaggi di programmazione.

2.2.6 - ELIMINAZIONE DELLE ε-RULES

Siano **YESε** l'insieme dei meta-simboli A1..Ak da cui si può ricavare ε e **NOε** l'insieme dei meta-simboli B1..Bm da cui non si può ricavare ε.

Per eliminare le ε-rules e determinare una grammatica equivalente si possono adottare i seguenti accorgimenti:

- Se la grammatica G contiene la regola S → ε — Anche G' contiene tale regola
- Se G contiene altre regole nella forma X → ε — G' non le contiene
- Se G contiene una produzione come X → $C_1C_2...C_r$ ($r \geq 1$) — G' contiene X→$\alpha_1\ \alpha_2...\ \alpha_r$ in cui
  - $\alpha_i = C_i$ — se ci sono meta-simboli da cui non si può ricavare ε — ($c_i \in VT \cup NO\varepsilon$)
  - $\alpha_i = C_i \mid \varepsilon$ — se ci sono meta-simboli da cui si può ricavare ε — ($c_i \in YES\varepsilon$)

Riassumendo, se in tutti i posti in cui c'era A tengo già in conto la stringa vuota, propagando l'effetto di equazione in equazione, salta fuori una grammatica che non ha produzioni che accorciano le frasi.

Attenzione però: nel tipo 2 posso sostituire a tappeto quello che voglio, mentre nel tipo 1 non è scontato poiché ho un pattern da rispettare (contesto) oppure potrei non riuscire a sostituirlo.

Esempio

Si consideri l'esempio a lato.
Qui YESε = {A} e NOε = {S, B}.

- **G *non contiene* la regola S → ε quindi *neppure G' la contiene.***
- **G *contiene una regola* della forma X → ε È la regola A → ε, *che va quindi tolta.***
- **Al suo posto,** poiché A∈YESε, ***ogni occorrenza di A va sostituita da*** **(A | ε)**
- Nel caso della regola **S → A B | B** ciò non ha effetti, poiché **(A | ε) B** riproduce **A B | B**
- **Invece, A → a A diventa A → a (A | ε)**
- **Semplificando e riscrivendo si ottiene la nuova grammatica G'**

Grammatica G (con ε -rules)
```
S → A B | B
A → a A | ε
B → b B | c
```

Grammatica G'
```
S → A B | B
A → a (A | ε)
B → b B | c
```

a(A | ε) = aA
oppure aε = a

Grammatica G'
```
S → A B | B
A → a A | a
B → b B | c
```

La nuova grammatica non genera mai la stringa vuota.

Esistono anche linguaggi in cui apparentemente non ci sono problemi nelle regole, ma sviluppandole scopriamo invece che possono produrre stringhe vuote.

A forza di sostituire, alcune volte può succedere che la stringa vuota vada a finire nella regola di top level, la S. Questo va bene perché significa che posso usare la stringa vuota nel mio linguaggio, ma in nessun momento vengono accorciate le frasi a tradimento.

Grammatica G (con ε -rules)

S → A B
A → a A | ε
B → b B | ε

Esempio

La prima regola APPARENTEMENTE non ha problemi, però tutte le volte in cui vedo A, devo metterci (A|ε), mentre quando c'è B devo mettere (B|ε).
Adesso abbiamo un problema nella prima regola: a forza di sostituire può succedere che la stringa vuota vada a finire nella regola di top level, la S.

Grammatica G'

S → (A|ε) (B|ε)
A → a (A|ε)
B → b (B|ε)

La nuova grammatica G' può generare la stringa vuota solo al primo passo della derivazione, ma non nei passi intermedi e questo va bene perché significa che il linguaggio comprende la stringa vuota, ma le forme di frase non possono mai accorciarsi.

Grammatica G'

S → A B | B | A | ε
A → a A | a
B → b B | b

2.3 - GRAMMATICHE E LINGUAGGI

Poiché le grammatiche sono in relazione gerarchica, può accadere che un linguaggio possa essere generato da più grammatiche, anche di tipo diverso. In pratica il "tipo del linguaggio" è il tipo sotto cui non riesci a scendere, ma generalmente la prima idea che ci viene in mente non è la grammatica più semplice.
Questo significa che **il tipo di un linguaggio può non coincidere col tipo della grammatica che lo genera**: noi da ora in poi useremo la **CONVENZIONE** tale per cui **un linguaggio è di un tipo X se il tipo della grammatica più semplice che lo genera è di tipo X**.

Il linguaggio $L = \{ a^n b^n c^n, n \geq 0 \}$ **è** (almeno) **di Tipo 1** in quanto esiste una grammatica di Tipo 1 che lo genera:

G1: S → aBC | aSBC; CB → DB; DB → DC; DC → BC; aB → ab; bB → bb; bC → bc; cC → cc

La grammatica diventa *più compatta* se espressa con la *definizione alternativa* di grammatica di Tipo 1, che ammette lo scambio:

G2: S → aBC | aSBC; CB → BC; aB → ab; bB → bb; bC → bc; cC → cc

Il linguaggio sarebbe però generabile *anche da una grammatica di Tipo 0*, come ad esempio quella mostrata in precedenza:

G0: S → aSBC; CB → BC; SB → bF; FB → bF; FC → cG; GC → cG; G → ε

Esempio $a^n b^n c^n$

Il linguaggio L non è di tipo 0 perché non contiene regole che accorciano frasi, ma non è nemmeno di tipo 2 perché ha simboli sia a destra che a sinistra.

Allora capiamo che è di tipo 1.
Lo capiamo anche perché prendendo le prime tre regole, vediamo bene che queste possono essere compattate con uno scambio, permesso solo nel tipo 1.

Un esempio ancora più semplice potrebbe essere **G3**: S → abc | aBSc, Ba → aB, Bb → bb
Se ho una coppia e voglio ficcarci in mezzo una coppia come abbiamo fatto con ab, aabb, aaabbb, posso farcela con un tipo 2. Ma se ho una stringa con 3 lettere, non posso ficcarci dentro qualcosa, perché ab posso metterla tra a e b, ma la c devo metterla più avanti. Quindi questo è l'esempio di un linguaggio che può essere solo di tipo 1.

Esempio di derivazione della frase "aabbcc"

Grammatica G3:

S → abc | aBSc
Ba → aB
Bb → bb

Derivazione:

S → aBSc → aBabcc → aaBbcc → aabbcc

Grammatica G2:

S → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

Derivazione:

S → aSBC → aaBCBC → aaBBCC →
→ aabBCC → aabbCC → aabbcC → aabbcc

2.3.1 - RAMI DI DERIVAZIONE "MORTI"

Il tipo 1 ha un piccolo difetto: esistono le strade chiuse. Se scegli male le prime due o tre regole, può succedere che ad un certo punto ci sia un blocco perché nessuna delle regole è applicabile. Questo con il tipo 2 non succederà mai, infatti ciò che frega il tipo 1 è che le sostituzioni hanno sempre delle condizioni, per questo motivo **tutti i linguaggi di programmazione sono settati al tipo 2**.

Esempio

Se sei una macchina devi essere in grado di tornare allo stadio precedente e di tentare un'altra strada.
Ma immaginiamo un linguaggio con molte regole: potrei accorgermi di aver sbagliato passaggi dopo un centinaio di mosse...

Grammatica G2:

S → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

Derivazione su ramo morto:

S → aSBC → aaBCBC → aabCBC →
→ aabcBC → ???????

Quindi cercheremo in tutti i modi di evitare di cascare in trucchi di questo tipo, per poter rendere efficiente una macchina.

2.3.2 - CARATTERISTICHE CRUCIALI DEI TIPI DI GRAMMATICA

Abbiamo detto che **nel tipo 1 la caratteristica principale è lo scambio**, che non è possibile nel tipo 2 in quanto non si hanno due elementi da scambiare, ma uno solo.
Inoltre **nel tipo 1** abbiamo fatto l'esempio del search & replace dicendo che si possono fare sostituzioni solo se si hanno determinate condizioni prima e dopo il meta-simbolo (**contesto**), mentre **nel tipo 2 si possono fare sostituzioni indipendenti dal contesto**.

Per quanto riguarda il confronto tra il tipo 2 e il tipo 3, sappiamo che **nel tipo 3 sono ammessi solo meta-simboli all'inizio o alla fine della frase** in quanto abbiamo una grammatica lineare che cresce sempre e solo a destra o sempre e solo a sinistra.
Nel tipo 2 invece si parla di SELF-EMBEDDING perché è possibile avere meta-simboli all'interno della frase, quindi **si può dividere la frase in tre parti**, cosa che non è possibile nel tipo 3.

- Tipo 1 (*dipendenti dal contesto*): *produzioni vincolate alla forma:*
  $\beta A \delta \rightarrow \beta \alpha \delta$
  con $\beta, \delta, \alpha \in (VT \cup VN)^*$, $A \in VN$, $\alpha \neq \varepsilon$
  Quindi, A può essere sostituita da α solo *nel contesto* β *A* δ
  Le riscritture *non accorciano mai la forma di frase corrente.*

- Tipo 2 (*libere dal contesto*): *produzioni vincolate alla forma:*
  $A \rightarrow \alpha$
  con $\alpha \in (VT \cup VN)^*$, $A \in VN$
  Attenzione: non c'è più il vincolo $\alpha \neq \varepsilon$
  Qui A può *sempre* essere sostituita da α, *indipendentemente dal contesto*, giacché non esiste più l'idea stessa di contesto.

- Tipo 3 (*grammatiche regolari*): *produzioni vincolate alle forme lineari:*
  lineare a destra: $A \rightarrow \sigma$ | $A \rightarrow \sigma B$
  lineare a sinistra: $A \rightarrow \sigma$ | $A \rightarrow B \sigma$
  con $A, B \in VN$, e $\sigma \in VT^*$

Riassumendo con una tabella:

	TIPO 1	TIPO 2	TIPO 3
STRADE CHIUSE	Sì	No	
SCAMBIO	Sì	No	
SOSTITUZIONI CON CONDIZIONI (contesto)	Sì	No	
UN SOLO METASIMBOLO IN CODA/TESTA		No	Sì
SELF EMBEDDING		Sì	No

SELF-EMBEDDING O AUTOINCLUSIONE

Una grammatica contiene self-embedding quando 1+ produzioni hanno la forma:

$$A \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \qquad (con\ \alpha_1, \alpha_2 \epsilon V^+)$$

TEOREMA: una grammatica di Tipo 2 *che non contenga self-embedding* genera un *linguaggio regolare*

- **Se NON c'è self-embedding** in nessuna produzione → **tipo 3**.
- **Se c'è self-embedding → tipo 2 o tipo 3**, dipende se il self-embedding è disattivato da altre regole

Il RUOLO DEL SELF-EMBEDDING è introdurre una ricorsione in cui si aggiungono contemporaneamente simboli a sinistra e a destra garantendo di procedere "di pari passo" (esempio delle colonne che crescono insieme). È essenziale per definire linguaggi in cui servono simboli bilanciati come le **parentesi**.

Es. S → (S) S → a L(G) = {(^{n} a)^n, n≥0}

La grammatica G:
S → aSc S → A A → bAc A → ε
presenta self-embedding e genera il linguaggio L(G):
L(G) = { $a^n b^m c^{n+m}$ n,m≥0 }

Esempio

In questo caso posso infilare qualcosa tra a e b, oppure cose tra b e c. Non ho il problema di dover inserire qualcosa contemporaneamente in mezzo alle due coppie, quindi in questo caso ho un linguaggio di tipo **2**.

Nonostante la presenza di self-embedding, il linguaggio generato può essere regolare se la regola con self-embedding è disattivata da altre regole meno restrittive: si parla in questo caso di FINTO SELF-EMBEDDING.

Esempio

S → a S a | X
X → a X | b X | a | b

Guardando solo la prima regola sembra che avremo un linguaggio con la forma uguale a quella dell'esempio precedente. Però X può aggiungere tutte le "a" che vuole o tutte le "b" che vuole: quindi **avremo una sequenza qualunque di a e di b senza che queste abbiano le stesse occorrenze**: quindi la seconda regola vanifica la prima… Allora è un linguaggio di tipo 3!

S → a b S b a | a b a

Esempio

Qui il self-embedding viene disattivato in modo subdolo perché alternando le "a" e le "b" non si riescono più a distinguere i "pezzetti". Non c'è più un confine tra la parte sinistra e la parte destra: avrei potuto usare una regola che aggiunge cose solo all'inizio o solo alla fine, quindi avrei potuto usare un linguaggio di tipo 3.

S → X a
X → a b | X a b a b

Infatti questo linguaggio produce frasi come: ab ab ab aba ba ba ba. In pratica la frase è una sequenza dispari di "ab" con una a finale, quindi si può facilmente avere una grammatica di tipo 3 equivalente.

Esempio

Avendo usato lo stesso simbolo, non riesco a distinguere la parte sinistra dalla parte destra, quindi il self-embedding è inutile poiché si poteva usare semplicemente un linguaggio di tipo 3.

S → a S a | ε

L(G) = {(aa)^n, n≥0} → Grammatica di tipo 3 equivalente S → aaS | ε

Si può generalizzare quanto appena detto con un teorema:

TEOREMA: ogni linguaggio *context-free di alfabeto unitario* è in realtà un *linguaggio regolare.*

Questo significa che **se ho un solo simbolo, sarà sempre un linguaggio di tipo 3**.

2.4 - RICONOSCIBILITÀ DEI LINGUAGGI

I linguaggi generati da grammatiche di Tipo 0 possono in generale NON essere riconoscibili (decidibili): non vuol dire che non riconosce niente la MdT, però può succedere che su 10 frasi ne riconosca 9 e alla decima vada in loop.

I linguaggi generati invece da grammatiche di Tipo 1 (e di conseguenza di Tipo 2 e 3) sono riconoscibili, tuttavia per avere un **traduttore efficiente** occorre adottare linguaggi generati da classi speciali di grammatiche di Tipo 2, detti **Parser**, o per ottenere particolare efficienza in sotto-parti di uso strettamente frequente, linguaggi generati da grammatiche di Tipo 3, detti **Scanner** (es. per i numeri).

GRAMMATICA	AUTOMI RICONOSCITORI
TIPO 0	Se L(G) è riconoscibile, MdT
TIPO 1	MdT con nastro di lunghezza proporzionale alla frase da riconoscere
TIPO 2	Push-Down Automaton (PDA) o ASF + stack
TIPO 3	Automa a stati finiti (ASF)

2.4.1 - NOTAZIONI: BNF, EBNF

Da ora in poi ci concentreremo sulle grammatiche di tipo 2 e 3. Siccome nelle tastiere non è semplice usare lettere greche e caratteri speciali, bisogna sostituire alcuni termini. Inoltre, i meta-simboli non possono usare le maiuscole, altrimenti non potremo usarle nel linguaggio.

In una **GRAMMATICA BNF**:

- Le regole di produzione hanno la forma $\alpha ::= \beta$ con $\alpha \in V^+, \beta \in V^*$
- I meta-simboli X∈VN hanno la forma **<nome>**
- Il meta-simbolo | indica l'alternativa → in questo modo si può esprimere un insieme di regole con la stessa parte sinistra (es. x ::= A1 | A2 | … | AN)

Esempio - BNF con gatto e topo

G = ⟨VT,VN,P,S⟩, dove:
VT = { il, gatto, topo, sasso, mangia, beve }
VN = { <frase>, <soggetto>, <verbo>, <compl-ogg>, <articolo>, <nome> }
S = <frase>
P = {
<frase> ::= <soggetto> <verbo> <compl-ogg>
<soggetto> ::= <articolo><nome>
<articolo> ::= il
<nome> ::= gatto | topo | sasso
<verbo> ::= mangia | beve
<compl-ogg> ::= <articolo> <nome>
}

ESEMPIO: derivazione della frase
"il gatto mangia il topo"
(ammesso che tale frase *sia derivabile*)

<frase>
→ <soggetto> <verbo> <compl-ogg>
→ <articolo> <nome> <verbo> <compl-ogg>
→ il <nome> <verbo> <compl-ogg>
→ il gatto <verbo> <compl-ogg>
→ il gatto mangia <compl-ogg>
→ il gatto mangia <articolo><nome>
→ il gatto mangia il <nome>
→ il gatto mangia il topo

La **NOTAZIONE EBNF** è una forma estesa della BNF che introduce alcune notazioni compatte per alleggerire la scrittura delle regole di produzione (vedi tabelle).

Forma EBNF	BNF equivalente	significato
X ::= [a] B	X ::= B | aB	a può comparire 0 o 1 volta
X ::= {a}^{n} B	X ::= B | aB | …| a^{n}B	a può comparire da 0 a n volte
X ::= {a} B	X ::= B | aX	a può comparire 0 o più volte

Forma EBNF	BNF equivalente	significato
X ::= (a | b) D | c	X ::= a D | b D | c	raggruppa categorie sintattiche

La terza produzione X ::= B |aX è ricorsiva a destra.

Esempio - EBNF con numeri naturali

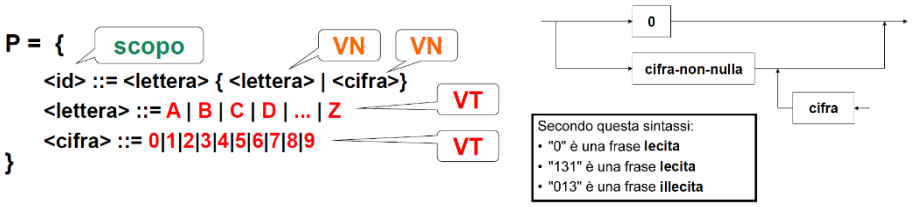
Si può esprimere la sintassi tramite un **DIAGRAMMA SINTATTICO**.

04 non è lecito, però se scrivo da 00 a 07 in C, va bene perché si suppone che lo 0 davanti sia la base ottale.

G = ⟨VT,VN,P,S⟩
dove:
VT = { 0,1,2,3,4,5,6,7,8,9 }
VN = { <num>, <cifra>, <cifra-non-nulla> }
S = <num>
P = {
<num> ::= <cifra> | <cifra-non-nulla> {<cifra>}
<cifra> ::= 0 | <cifra-non-nulla>
<cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
}

EBNF

Esempio – EBNF con identificatori



2.4.2 - ALBERI DI DERIVAZIONE

Per le sole **grammatiche di Tipo 2** si possono utilizzare gli **alberi di derivazione,** in cui:

- Ogni **nodo** dell'albero è associato ad un **simbolo** del vocabolario V = VT UVN
- La **radice** dell'albero coincide con lo **scopo S**
- Se a1, a2, …, ak sono i k figli ordinati di un dato nodo X (associato al simbolo X∈VN), significa che la grammatica contiene la produzione X ::= A1 A2 … AK dove Ai è il simbolo associato al nodo ai

L'albero di derivazione non può esistere per grammatiche di tipo 1 e 0 perché il lato sinistro delle produzioni ha più di un simbolo, quindi i nodi figli avrebbero più di un padre → no albero, ma grafo generico.

Esempio

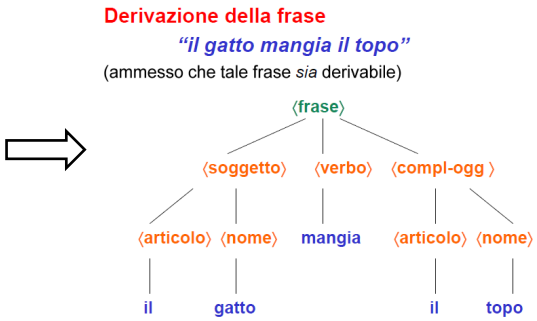
G = ⟨VT,VN,P,S⟩ con

VT = { il, gatto, topo, sasso, mangia, beve }

VN = { <frase>, <soggetto>, <verbo>, <compl-ogg>, <articolo>, <nome> }

S = <frase>

P = {
<frase> ::= <soggetto> <verbo> <compl-ogg>
<soggetto> ::= <articolo><nome>
<articolo> ::= il
<nome> ::= gatto | topo | sasso
<verbo> ::= mangia | beve
<compl-ogg> ::= <articolo> <nome>
}



Le regole EBNF non sono direttamente mappabili su un albero, bisogna riscriverle in BND standard.

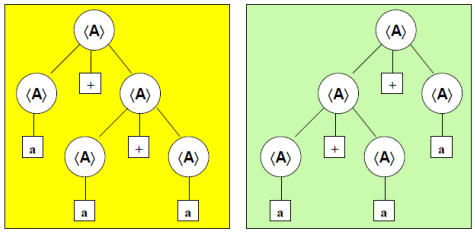
2.4.3 – DERIVAZIONI CANONICHE e AMBIGUITÀ

Esistono due tipi di **Derivazioni Canoniche:**

- **Derivazione left-most**: il simbolo non-terminale è sempre quello più a sinistra.
- **Derivazione right-most**: il simbolo non-terminale è sempre quello più a destra.

Una GRAMMATICA È AMBIGUA se esiste almeno una frase che ammette 2+ derivazioni canoniche sinistre distinte. Grado ambiguità = n alberi sintattici distinti.
Qui, ad esempio, ho due alberi diversi, ma ci sono le stesse foglie: la grammatica è sovrabbondante e quindi il parser diventa scemo perché si incasina.
L'**ambiguità è una di quelle caratteristiche che non sono desiderabili**.

ESEMPIO
A ::= A + A
A ::= a
La frase a+a+a è *ambigua*:



Stabilire se una grammatica di Tipo 2 è ambigua o meno è un problema indecidibile. Spesso, se una grammatica è ambigua, se ne può trovare un'altra che non lo sia.

Esistono poi dei linguaggi detti INTRINSECAMENTE AMBIGUI e sono quelli in cui **tutte le grammatiche che li generano sono ambigue**, come ad esempio:

$$L = \{ a^n b^n c^* \} \cup \{ a^* b^n c^n \}, \text{ con } n \geq 0$$

In questo linguaggio tutte le frasi della forma $a^n b^n c^n$ appartengono a entrambi i sotto-linguaggi, quindi esistono due derivazioni distinte. C'è un caso particolare in cui tutte e tre le lettere a, b e c sono ripetute n volte.

2.4.4 - LA STRINGA VUOTA

La stringa vuota può far parte delle frasi generate da una grammatica di Tipo 0 (le frasi possono accorciarsi)**, ma non può far parte delle frasi generate da una grammatica di Tipo 1** (le frasi non si possono mai accorciare).
Come abbiamo già detto, però, questo è ok perché anche se le grammatiche di Tipo 2 e 3 ammettono la stringa vuota sul lato destro, esiste sempre una grammatica equivalente senza **ε-rules.**
In più, **fa comodo avere la stringa vuota per esprimere parti del linguaggio opzionali**, infatti è possibile farlo senza alterare il tipo della grammatica purché:

- Questa ammetta la **presenza di ε nella sola produzione di top-level S → ε**
- **S non compaia altrove**

In questo modo, la stringa vuota può essere scelta solo all'inizio (cioè al primo passo di derivazione), facendo in modo che le forme di frasi non si accorcino. Più formalmente:

TEOREMA
- **Dato un linguaggio L di tipo 0, 1, 2, o 3**
- **i linguaggi $L \cup \{\varepsilon\}$ e $L - \{\varepsilon\}$ *sono dello stesso tipo.***

Questo teorema è importante perché ci dice che semplicemente **si può scegliere di avere la stringa vuota oppure no: è una nostra scelta**. Non cambia nulla perché comunque un linguaggio di tipo X - con la stringa vuota oppure no, rimane sempre dello stesso tipo.

Esempio - Tipo 2

$S ::= \varepsilon \mid X$
$X ::= ab \mid a\,X\,b$

$=$

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

2.4.5 - FORME NORMALI

Un linguaggio di Tipo 2 <u>non vuoto</u> può essere sempre generato da una grammatica di Tipo 2 in cui:

- **Ogni simbolo compare nella derivazione di qualche frase di L** (esistono solo simboli utili)
- **Non ci sono produzioni della forma A→B** con A, B simboli non terminali (niente produzioni che rinominano i simboli)
- **Se il linguaggio non comprende la stringa vuota, allora <u>non</u> ci sono produzioni A→ ε**

Conosciamo <u>DUE FORME NORMALI</u> a cui possiamo condurre tutte le produzioni:

- **FORMA NORMALE DI CHOMSKY: A → BC | a** con $A,B,C \in VN$, $a \in VT \cup \varepsilon$
  Quindi o produci due simboli non terminali, oppure un solo simbolo terminale.
- **FORMA NORMALE DI GREIBACH: A → a α** con $A \in VN$, $a \in VT$, $\alpha \in VN^*$
  Questa forma vale per linguaggio privi di ε e ogni produzione indica una frase con un simbolo terminale seguito da qualsiasi stringa.

La forma normale di Greibach è molto utilizzata perché evidenziare l'iniziale è molto utile in quanto un Riconoscitore (Parser) appena vede il carattere iniziale della frase, capisce subito quale regola guardare.

Esempio – da Tipo 2 a FN di Chomsky

Esiste un algoritmo che trasforma ogni grammatica di tipo 2 in forma normale di Chomsky.

Per Greibach servono alcune tecniche extra.

• **Grammatica data:**

S → d A | c B
A → d A A | c S | c
B → c B B | d S | d

• ***Forma normale di Chomsky***

S → M A | N B
A → M P | N S | c
B → N Q | M S | d

M → d
P → A A
Q → B B

N → c

2.4.6 - TRASFORMAZIONI IMPORTANTI

Per facilitare la costruzione dei riconoscitori, è spesso rilevante poter trasformare la struttura delle regole di produzione. **Le trasformazioni importanti sono sostanzialmente tre:**

- **Sostituzione**
- **Raccoglimento a fattor comune**
- **Eliminazione della ricorsione sinistra** → questa non è scontata.

La ricorsione destra non ha problemi perché rende evidente per cosa cominciano le frasi. Invece, la ricorsione sinistra è un problema perché un parser deve vedere con cosa inizia una frase per poter capire qual è il ramo giusto dell'albero da percorrere.
Es. Modena e Bologna nei cartelli stradali: Se leggi "M" pensi già a Modena e vai giù di lì, non guardi se termina per "a".

SOSTITUZIONE

Significa espandere un simbolo VN che compare nella parte **destra** di una regola di produzione, sfruttando un'altra regola di produzione.

Esempio

Nella grammatica a lato è possibile sostituire il metasimbolo S nella seconda produzione, usando a tale scopo la prima produzione.	ESEMPIO S → X a X → b Q | S c | d
Espandiamo quindi S come indicato: la nuova regola per X non contiene più alcun riferimento a S	ESEMPIO S → X a X → b Q | X a c | d

RACCOGLIMENTO A FATTOR COMUNE

Significa **isolare il prefisso più lungo comune a due produzioni**.

Esempio

Nella grammatica a lato è possibile isolare il prefisso a S comune alle prime due produzioni.	ESEMPIO S → a S b | a S c
Raccogliamo quindi a fattore comune il prefisso comune a S …	ESEMPIO S → a S (b | c)
…e introduciamo un nuovo meta-simbolo X per esprimere *la parte che segue* il prefisso comune.	ESEMPIO S → a S X X → b | c

In questo esempio ritardo la decisione al momento in cui saprò le cose e procedo in modo **deterministico**. Quando arriva il momento della sostituzione, so già che arriverò alla fine in modo corretto, non faccio più tentativi col rischio di far sbagliare la macchina.

ELIMINAZIONE DELLA RICORSIONE SINISTRA

La ricorsione sinistra ***X→ Xac | p*** nasconde l'iniziale delle frasi prodotte, che si può determinare solo guardando altre regole. In questa produzione tutte le frasi iniziano per p, ma non si vede dalla prima regola, che è ricorsiva a sinistra.

Si può sempre sostituire la ricorsione sinistra con la ricorsione destra, ma ciò ha delle CONSEGUENZE: cambiando le regole, cambia la sequenza di derivazione, ergo è possibile che venga **modificata** anche la **semantica** della grammatica.

La trasformazione è articolata in due passi:

- **Fase 1: elimino cicli ricorsivi a sinistra**
- **Fase 2: elimino ricorsione sinistra diretta**

Esempio

Fase preliminare • si stabilisce una *relazione d'ordine* fra i meta-simboli coinvolti del ciclo ricorsivo • Nel nostro caso, sia dunque C > B > A	ESEMPIO A → B a B → C b C → A c | p
Fase 1 • si modificano tutte le produzioni del tipo Y → Xα *in cui* Y > X, sostituendo a X le forme di frase stabilite dalle produzioni relative a X	Si ottiene quindi: A → B a B → C b C → C b a c | p
Fase 2 • le produzioni ricorsive dirette X → X α | p si modificano introducendo un metasimbolo Z e scrivendo X → p | p Z e Z → α | α Z	Ergo, C → C b a c | p diventa C → p | p Z Z → b a c | b a c Z

Esempio – Espressioni aritmetiche

Nelle espressioni aritmetiche la cultura matematica diffusa richiede associatività sinistra che si ha con regole grammaticali con ricorsione a sinistra. Se trasformiamo il linguaggio per ottenere una ricorsione destra, non possiamo più fare i calcoli perché non sarebbero culturalmente accettati.
Infatti, 13-5-4 = (13-5)-4 = 4 e **NON** 13-(5-4) = 12.

Le trasformazioni appena viste permettono di trasformare una grammatica in forma normale di Greibach.

Esempio

- eliminazione ciclo ricorsivo a sinistra
- eliminazione ricorsione sinistra diretta
- sostituzione
- ridenominazione dei terminali tramite non-terminali ausiliari

Fase 1 • relazione d'ordine fra i simboli non terminali coinvolti del ciclo ricorsivo : X > S	Grammatica data: S → X a X → b S | S c | d
Fase 2 • modifica della produzione X → S c sostituendo a S la produzione S → X a	Si ottiene quindi: S → X a X → (b S | d) | X a c
Fase 3 • eliminazione ricorsione sinistra X→ X α | p, qui con p = (bS | d), introducendo il nuovo simbolo Z tale che Z → α | α Z e X ::= p Z | p	da cui: S → X a Z → a c | a c Z X → (bS | d) Z | (bS | d)
Fase 4 • sostituzione del simbolo X nella prima regola	S → bSa | bSZa | dZa | da Z → a c | a c Z
Fase 5 • introduzione dei non-terminali ausiliari A e C per rappresentare a e c dove appropriato	S → bSA | bSZA | dZA | dA Z → a C | a C Z A → a C → c

2.4.7 - PUMPING LEMMA

Capire se un linguaggio è di tipo 2 o di tipo 3 solo guardandolo, in generale, non è banale. Se ho la grammatica, riesco più facilmente, ma il problema è che per scrivere un linguaggio sono io stesso che devo scrivere le regole. Solo dopo aver chiaro lo scopo del linguaggio, posso pensare alle regole che lo producono. Oltretutto è possibile che all'inizio ci venga in mente una grammatica "difficile", e solo approfondendo poi può venirci in mente una grammatica più semplice.

Ecco allora che **il PUMPING LEMMA ci dà una condizione necessaria (ma non sufficiente) perché un linguaggio sia di tipo 2 o 3,** dimostrando che una cosa NON è vera, e cioè che un linguaggio non può essere di tipo 2 o 3.

Idea di base: in un linguaggio infinito, **ogni stringa sufficientemente lunga deve avere una parte che si ripete**. Essa può essere quindi **"pompata" un qualunque numero di volte ottenendo sempre altre stringhe del linguaggio.**

PUMPING LEMMA PER LINGUAGGI DI TIPO 2 – CONTEXT FREE

Il primo punto si interpreta così: supponendo che la frase inizi per qualsiasi cosa "u", che finisca per qualsiasi cosa "y" e che in mezzo abbia qualsiasi cosa "w", gli elementi che si ripetono sono v e x. (con "u" e "y" che potrebbero anche non esserci)

Se L è un linguaggio di Tipo 2, esiste un intero N 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 entrambi nulle:** $|vx| \geq 1$
- **la 2ª e la 4ª parte possono essere *"pompate" quanto si vuole* ottenendo sempre altre frasi del linguaggio; ovvero,** $uv^iwx^iy \in L \quad \forall\, i \geq 0$

Es. Se posso avere "v" e "x" 1 volta, 2 volte, 100mila volte, **ma non 6 volte**, allora ho un linguaggio almeno di tipo 1 e sicuramente non di tipo 2.

N = lunghezza minima delle stringhe decomponibili in 5 parti, dipende dallo specifico linguaggio.
La dimostrazione si basa sulle lunghezze dei cammini nell'albero di derivazione associato.

PUMPING LEMMA PER LINGUAGGI DI TIPO 3 - REGOLARI

In questo caso c'è un solo "pilastro" che può crescere, non si hanno più delle coppie.
Anche qui N dipende dallo specifico linguaggio, ma stavolta la dimostrazione è basata sull'automa a stati associato.

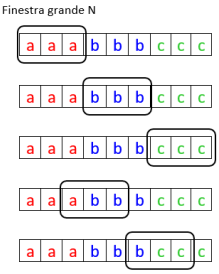
Se L è un linguaggio di Tipo 3, esiste un intero N tale che, per ogni stringa z di lunghezza almeno pari a N:
- **z può essere riscritta come:** $z = xyw$ $|z| \geq N$
- **la parte centrale *xy* ha lunghezza limitata:** $|xy| \leq N$
- ***y* non è nulla:** $|y| \geq 1$
- **la parte centrale può essere *pompata quanto si vuole* ottenendo sempre altre frasi del linguaggio; ovvero,** $xy^iw \in L \quad \forall\, i \geq 0$

Esempio: $L = \{a^p \text{, } p \text{ primo}\}$ **non è un linguaggio regolare.**
Se L fosse regolare, esisterebbe un **intero N** in grado di soddisfare il pumping lemma, quindi consideriamo:
- **P numero primo ≥ N+2**
- **Stringa $z=a^p$**

Ci serve un numero primo P che possa essere scomposto, se no se non è abbastanza lungo non posso fare nulla, infatti scompongo z nei tre pezzi **xyw** con:
- **$|y| = r$** (lunghezza di y)
- **$|xw| = p-r$** (z è lunga p, il pezzo centrale è r, quindi gli altri caratteri sono p-r).

Ora, supponendo che L sia regolare, la nuova stringa $\mathbf{xy^{p-r}w}$ dovrebbe appartenere al linguaggio, ma la sua lunghezza $\mathbf{|xy^{p-r}w| = (p-r)(1+r)}$, perciò non è un numero primo e non appartiene al linguaggio. Allora, L non è regolare, cioè non è di tipo 3.

Esempio: $L = \{a^n b^n c^n\}$ **non è context-free**
Se L fosse context-free, esisterebbe un **intero N abbastanza grande** in grado di soddisfare il pumping lemma. Se consideriamo la stringa $\mathbf{z = a^N b^N c^N}$ e la scomponiamo in cinque pezzi **uvwxy** con **$|vwx| \leq N$,** poiché fra l'ultima "a" e la prima "c" ci sono N posizioni, il pezzo centrale **vwx** non può contenere sia "a" che "c".



È come se avessi una sorta di finestra mobile: se sto a sinistra ho tutte a, se sto in mezzo tutte b, se sto a destra tutte c, se sto non esattamente in mezzo posso avere qualche b e qualche c, ad esempio, ma mai tutte e tre le lettere contemporaneamente.

2.5 - LE ESPRESSIONI REGOLARI

Un formalismo di particolare interesse è quello delle **espressioni regolari**. Le espressioni regolari sono **tutte e sole le espressioni ottenibili tramite le seguenti regole:**
- La stringa vuota ε è una espressione regolare
- Dato un alfabeto A, ogni elemento $a \in A$ è una espressione regolare
- Se X e Y sono espressioni regolari, lo sono anche:

- **Unione**: X + Y = {x | x∈X v x∈Y}. Inteso come operatore insiemistico, è il meno prioritario
- **Concatenazione**: X · Y = {x | x=ab, a∈X ^ b∈Y} {} ·X = {} per qualsiasi x
- **Chiusura**: X* = X0 ∪ X1 ∪ X2 … dove X0=ε e $X^k=X^{k-1}\cdot X$. È l'operatore più prioritario

La **concatenazione** è intesa come una combinazione: ho un prefisso e un suffisso da combinare. Inoltre, la concatenazione è associativa ma non è commutativa.

Esempio

Nell'unione abbiamo i 4 simboli presenti in X1 e X2.

X1 = {00, 11}
X2 = {01, 10}
X1 + X2 = {00, 11, 01, 10}
X1 • X2 = {0001, 1101, 0010, 1110}
X2 • X1 = {0100, 0111, 1000, 1011}
X1* = {ε , 00,11, 0000,0011,1100,1111, 000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111, … }

Nella prima concatenazione abbiamo la combinazione degli elementi di X1 utilizzati come prefissi con gli elementi di X2 usati come suffissi, mentre nella seconda concatenazione abbiamo il contrario.
Nella chiusura di X1 abbiamo i due simboli, poi abbiamo la combinazione dei due simboli secondo le potenze:
$X1^2$ è 0000, 0011, 1100, 1111
$X1^3$ è 000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111

Uno stesso linguaggio può essere descritto da molte espressioni regolari diverse.

Con riferimento a linguaggi:

- ε denota il linguaggio vuoto
- Un elemento a∈A denota il linguaggio {a}
- R1+R2 denota l'unione dei linguaggi denotati da R1 e R2
- R1·R2 denota la concatenazione dei linguaggi denotati da R1 e R2
- R* denota il risultato dell'operatore di chiusura applicato al linguaggio denotato da R

Esempio

ESEMPIO sull'alfabeto A = { 0, 1 }
0 + 1* = {0, ε , 1, 11, 111, 1111, 11111, … }
(0 + 1)* = {0+1, ε , (0+1)(0+1), (0+1)(0+1)(0+1), … } =
= {ε , 0, 1, 00,10,01,11, 000,010,001,011,100,110,101,111 , … }
= A*
(10 • 01)* = (1001)* = { ε , 1001, 10011001, 100110011001, … }

2.5.1 - ESPRESSIONI E LINGUAGGI REGOLARI

Le Regular Expression sono un modo più compatto per esprimere gli elementi di un insieme e hanno una forma regolare.

TEOREMA
i linguaggi generati da *grammatiche regolari*
coincidono
con i linguaggi descritti da *espressioni regolari.*

Grammatiche ed espressioni regolari sono quindi due rappresentazioni diverse della stessa realtà:

- **Una è costruttiva**: le grammatiche ci dicono come costruire le frasi (come si fa, non cosa si ottiene)
- **Una è descrittiva**: le espressioni ci descrivono le frasi (cosa si ottiene, non come si fa)

DALLA GRAMMATICA ALL'ESPRESSIONE REGOLARE

Per passare dalla grammatica all'espressione regolare **si interpretano le produzioni come EQUAZIONI SINTATTICHE** in cui i terminali sono i termini noti e i linguaggi generati da ogni simbolo non terminale sono le incognite. Si risolvono con le normali regole algebriche.

Esempio - ricorsione a destra

Frasi lecite: a, a+a, a-a, a+a-a, a+a+a, …

ESEMPIO: la grammatica lineare a destra vista in precedenza:

$S \rightarrow a \mid a + S \mid a - S$

può essere letta come un'equazione con

- **tre termini noti:** a, +, -
- **una incognita,** L_S

che impone il vincolo *(usiamo per l'unione il simbolo ∪ anziché +)*

$L_S = a \cup (a + L_S) \cup (a - L_S) = (a + \cup a -) L_S \cup a$

la cui soluzione, come vedremo ora, è l'espressione regolare

$S = (a + \cup a -)^* a$

Per risolvere le equazioni sintattiche si usa un ALGORITMO che esiste in due versioni: una per le grammatiche regolari a destra, una per quelle regolari a sinistra; nel primo caso si raccoglie a destra, nel secondo caso si raccoglie a sinistra.

Algoritmo per grammatiche regolari a destra

1. **Riscrivere ogni gruppo di produzioni del tipo** $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ **come** $X = \alpha_1 + \alpha_2 + \dots + \alpha_n$
2. **Poiché la grammatica è lineare a destra, ogni** α_k **ha la forma** uX_k dove $X_k \in VN \cup \varepsilon$, $u \in VT^*$
   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 $X_k \in VN$, $u_k, z_k \in VT^*$
   Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN (cioè il numero di simboli non terminali)
3. **Eliminare dalle equazioni le ricorsioni dirette, data l'equivalenza**
   $X = u X \cup \delta \quad \leftrightarrow \quad X = (u)^* \delta$
   Ognuna delle forme di frase δ conterrà altre incognite, ma non X.
4. **Risolvere il sistema rispetto a S per eliminazioni successive** (metodo di Gauss), eventualmente ri-applicando (2) e (3) per trasformare le equazioni via via ottenute.
5. **La soluzione del sistema è il linguaggio regolare cercato.**

Algoritmo per grammatiche regolari a sinistra

1. **Riscrivere ogni gruppo di produzioni del tipo** $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ **come** $X = \alpha_1 + \alpha_2 + \dots + \alpha_n$
2. **Poiché la grammatica è lineare a sinistra, ogni** α_k **ha la forma** $X_k u$ dove $X_k \in VN \cup \varepsilon$, $u \in VT^*$
   Ergo, si raccolgano a sinistra i simboli non-terminali dei vari $\alpha_1 \dots \alpha_n$ **scrivendo** $X = X_1 (u_1 + u_2 + \dots) + \dots + X_n (z_1 + z_2 + \dots)$
   dove $X_k \in VN$, $u_k, z_k \in VT^*$
   Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN (cioè il numero di simboli non terminali)
3. **Eliminare dalle equazioni le ricorsioni dirette, data l'equivalenza**
   $X = X u \cup \delta \quad \leftrightarrow \quad X = \delta (u)^*$
   Ognuna delle forme di frase δ conterrà altre incognite, ma non X.
4. **Risolvere il sistema rispetto a S per eliminazioni successive** (metodo di Gauss), eventualmente ri-applicando (2) e (3) per trasformare le equazioni via via ottenute.
5. **La soluzione del sistema è il linguaggio regolare cercato.**

Esempio: Grammatica lineare a destra

Fase 1 • scrittura di un'equazione per ogni regola:	Grammatica data: $S \rightarrow a\ B \mid a\ S$ $B \rightarrow d\ S \mid b$
Fase 2 • eventuali raccoglimenti a fattore comune per evidenziare suffissi: *qui non ce ne sono*	Equazioni: $S = a\ B + a\ S$ $B = d\ S + b$
Fase 3 • eliminare la ricorsione diretta $X = u\ X + \delta$ riscrivendola come $X = u^*\ \delta$ (qui $\delta = a\ B$)	$S = a^*\ a\ B$ $B = d\ S + b$
Fase 4 • sostituzione della 2ª equazione nella 1ª e sviluppo dei relativi calcoli	$S = a^*\ a\ (d\ S + b) =$ $= a^*\ a\ d\ S + a^*\ a\ b$
Fase 5 • nuova eliminazione della ricorsione introdotta al punto precedente: risultato finale.	$S = a^*\ a\ d\ S + a^*\ a\ b$ $S = (a^*\ a\ d)^*\ a^*\ a\ b$

Qui ho una ricorsione destra su S

Suppongo di sostituire B dentro S

Risultato: ho 1+ "a" seguite da "ad", tutto almeno una volta, poi ho 1+ "a" seguite da "ab". → es. a ad aa ad a a ab

Esempio - variante: Grammatica lineare a destra

Fase 1 • scrittura di un'equazione per ogni regola:	Grammatica data: $S \rightarrow a\ B \mid a\ S$ $B \rightarrow d\ S \mid b$
Fase 2 • se ora eliminiamo subito B, sostituendo la 2ª equazione nella 1ª e raccogliamo S:	Equazioni: $S = a\ B + a\ S$ $B = d\ S + b$
Fase 3 • eliminando ora la ricorsione $X = u\ X + \delta$ riscrivendola come $X = u^*\ \delta$ (qui $\delta = a\ b$)	$S = a\ (d\ S + b) + a\ S =$ $= (a\ d + a)\ S + a\ b$
• che costituisce già una espressione regolare (risultato finale)	$S = (a\ d + a)^*\ a\ b$

Qui provo a sostituire subito B

Risultato: ho 1+ volte il blocco "ad" oppure "a" seguito da "ab" → es. a ad a a ad a a ab

Per fare una prova - che non dimostra nulla ma aiuta, possiamo eliminare tutti gli elementi con l'asterisco per creare la frase più semplice e vedere se queste due espressioni così diverse hanno almeno una frase in comune:

Es. (a* ad)* a* ab → ~~(a* ad)* a*~~ ab → ab

Es. (ad + a)* ab → ~~(ad + a)*~~ ab → ab

Esiste una **terza espressione deterministica equivalente**: S = a (da + a)* b.

In pratica sono tutte frasi che iniziano per "a", terminano per "b", e hanno eventualmente in mezzo "a" o "da" ripetuti un numero arbitrario di volte, ma **questa terza espressione per un parser è meglio perché gli dice da subito con che lettera inizia la frase!**

Per ottenere espressioni equivalenti dello stesso linguaggio si possono manipolare algebricamente le espressioni di partenza (solo se sono semplici, se no ciao) oppure si può operare sugli automi a stati finiti che hanno algoritmi pratici per trasformare macchine in altre macchine (lo vedremo più avanti).

DALL'ESPRESSIONE REGOLARE ALLA GRAMMATICA

Per passare invece dall'espressione regolare alla grammatica **si interpretano gli operatori dell'espressione regolare in base alla loro semantica, mappandoli in opportune regole:**

- **Sequenza** = simboli accostati nella grammatica
- **Operatore +** = simbolo di alternativa nella grammatica (regole distinte)
- **Operatore *** = regola ricorsiva nella grammatica (ciclo)

Esempio

ESEMPIO: l'espressione regolare vista in precedenza

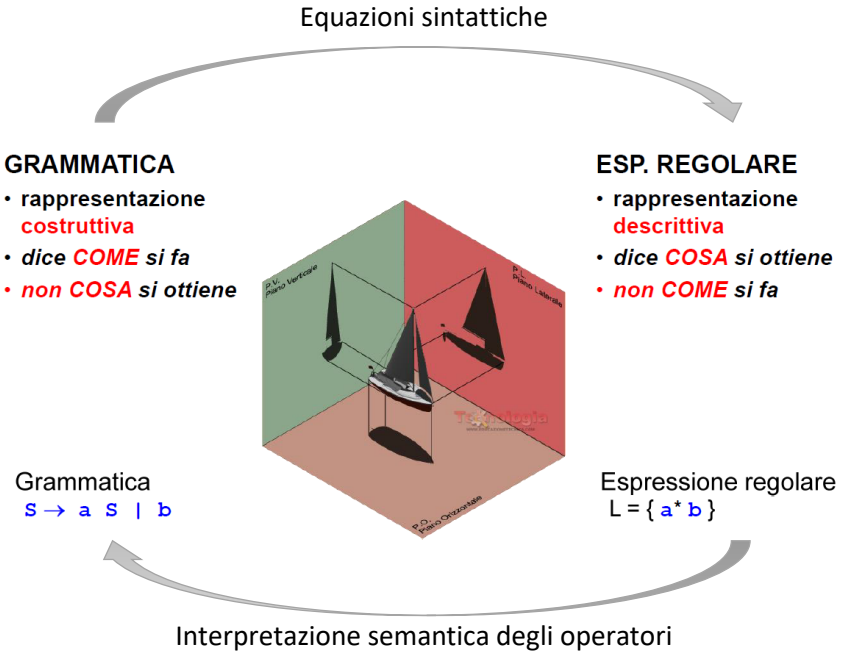
$L = \{ a^* b \}$

Tutte le frasi di L sono composte dal prefisso a^* (che può mancare) e dal suffisso b (che invece c'è sempre)

$S \rightarrow A\ b \mid b$

Il prefisso a^* può essere prodotto da una regola ricorsiva, del tipo:

$A \rightarrow A\ a$ o anche $A \rightarrow a\ A$



3 - PARADIGMA IMPERATIVO E DICHIARATIVO

Nel PARADIGMA IMPERATIVO, il programma è una sequenza di istruzioni, di ordini (di alto livello o meno), ma alla fine sono segnali elettrici la cui conseguenza è fare operazioni semplici e in un **determinato ordine**.
È intuitivo per noi umani ed immediato da mappare sulla macchina, ma è intrinsecamente non invertibile.
Se abbiamo dei problemi da risolvere molto complessi, per dominare la complessità devo prevedere tutti i casi perché la macchina non si inceppi, ma a volte non possiamo nemmeno immaginarli certi problemi, perciò è veramente difficile.

Esempio - Equazione y=x+2
Io umano posso vedere la simmetria in un'equazione, ad esempio, perché la parte destra è uguale alla sinistra e viceversa. In una macchina con un linguaggio imperativo non posso ragionare allo stesso modo: o risolvo tutto in base a x o risolvo tutto in base a y, non posso variare a caso. Devo scegliere adesso quali sono gli input e quali sono gli output, e indietro non si torna.
Supponiamo di voler creare una funzione *int solve(Integer x, Integer y)*, come dividiamo i casi? Se i due argomenti sono entrambi null cosa può succedere? Illegal Exception? La funzione sa solo calcolare, non sa generare interi. E se volessi poter trovare due valori che soddisfano l'equazione? In un caso come questo non è possibile perché diamo per scontato che siano tutti e due input, ma chi l'ha detto?
Inoltre il null è un problema, perché è difficilissimo da gestire. Hanno inventato l'Optional, ma anche quello si usa per un output che non sappiamo se è stato prodotto o meno, non è bello usarlo per un input.
In più, per ogni caso bisogna prevedere un if: come facciamo a gestirli se gli argomenti sono molti di più? Diventa una roba inguardabile. SOLUZIONE: CAMBIARE APPROCCIO.

I linguaggi imperativi per questo tipo di operazioni non sono utili né facili da gestire, ecco perché si può utilizzare un paradigma dichiarativo. **Nel PARADIGMA DICHIARATIVO non si esprimono ordini, ma si esprimono le relazioni fra le entità**: questo è meno intuitivo per l'essere umano, ma è **intrinsecamente invertibile** e **si limita ad affermare ciò che è vero. SI DELEGA IL CONTROLLO.**
Nell'esempio di prima, usando un linguaggio dichiarativo come il Prolog, si può calcolare l'equazione ricavando x, ricavando y o addirittura generando dei valori di x e y che soddisfino l'equazione.

Esempio - append di due liste
In un linguaggio imperativo dobbiamo prevedere tutti i possibili casi e ritornare un risultato, mentre in un linguaggio dichiarativo possiamo semplicemente esprimere cosa significa fare append di due liste e a questo punto possiamo interrogare un IDE di un linguaggio dichiarativo per ottenere un risultato o per chiedergli di generare delle liste che appese tra loro danno un risultato che gli diamo noi.

```
append([], L, L).
append([H|T], L, [H|Tr]) :- append(T,L,Tr).
```

3.1 - MINI-CORSO DI PROLOG

- Un programma Prolog è un insieme di **regole (teoria logica)** espresse secondo la notazione: ***testa :- corpo***.
- L'operatore **:-** esprime l'implicazione logica (←) cioè, se il corpo è vero, allora è vera anche la testa
- La testa ha la forma ***funtore(lista_args),*** in cui la lista degli argomenti può mancare
- Il corpo è una congiunzione di termini separati da virgole, ognuno dei quali ha la stessa forma della testa
- Testa e corpo possono contenere variabili (***Es***. p(a,12,X) :- q(a), r(13), s(X,1).)

- **Lo scope di una variabile è la singola regola**: se abbiamo la stessa variabile in più regole, questa è legata solo alla regola in cui è inserita e non alle altre
- Se servono termini che iniziano per maiuscola, bisogna mettere gli apici

Esempio

1° Query: ?- father(X, paul).

Il programma va a cercare la regola father() e sostituisce a Y la variabile X e a C il nome Paul. Guarderà a questo punto tutti gli uomini del database (Adam e Peter) e tutti i genitori di Paul (Adam e Mary) e darà come risultato Adam.

ESEMPIO 1

A chi piace cosa:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

Assiomi (fatti).

SEMANTICA INFORMALE

- **Y è padre di C** se
  Y è un uomo e inoltre
  Y è genitore di C
- **X è madre di C** se
  X è una donna e inoltre
  X è genitore di C

ESEMPIO 2

Figli e genitori:

```
man(adam).
man(peter).
woman(mary).
woman(eve).
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).
father(Y,C):-
 man(Y), parent(Y,C).
mother(X,C):-
 woman(X),parent(X,C).
```

Assiomi (fatti)

parent(genitore, figlio)

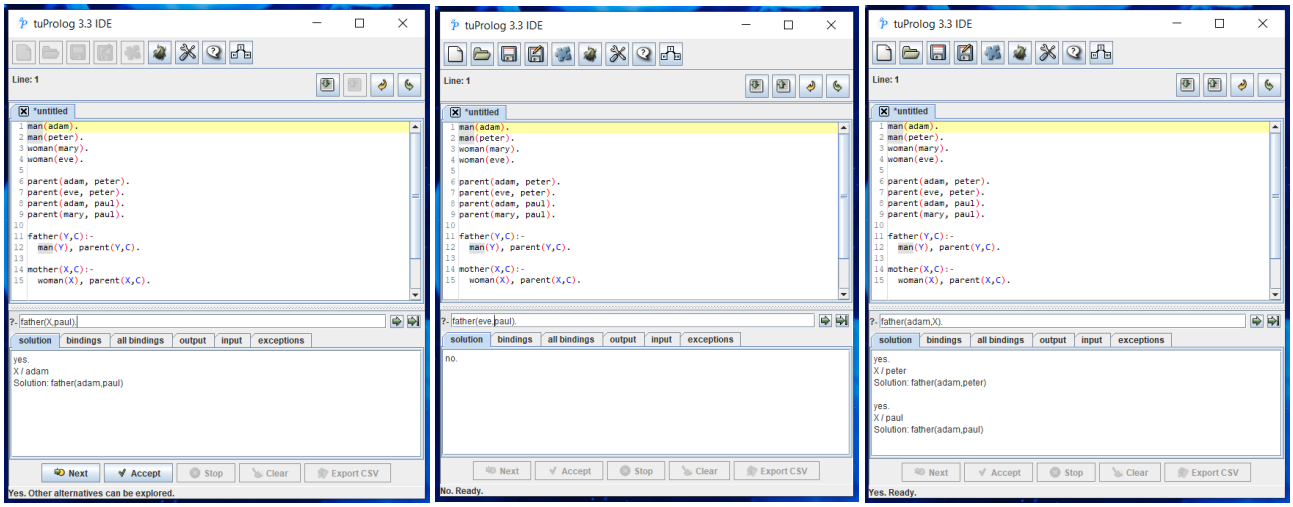
Regole

2° Query: ?- father(eve, paul).

Qui vogliamo sapere se Eve è il padre di Paul, ma Eve è femmina quindi la query fallisce alla prima verifica.

3° Query: ?- father(adam, X).

Qui vogliamo sapere di chi è padre Adam: il sistema cercherà tutti i genitori di sesso maschile che hanno come genitore Adam e troverà due risultati: Peter e Paul. Prima di tutto proporrà come risultato Peter; se non viene accettato, allora proporrà Paul. Se non dovesse essere accettato nemmeno lui risponderà con "no" che significa "Adam non ha altri figli".



Esempio - numeri

Per i numeri naturali non è opportuno adottare l'usuale rappresentazione 1,2,3,4… perché i simboli sono legati ad un certo significato che esiste nella nostra testa, ma non si può spiegare così com'è ad una macchina. Allora si usa la teoria matematica: i numeri naturali partono da 1 e sono suoi **successori**.

Indichiamo:

- s(N) il successore di N
- eq(X,Y) la relazione x+2=y → eq(X,Y) :- s(s(X))=Y. oppure eq(X, s(s(X))).

Adesso, se fornisco entrambi gli elementi, il programma verifica se la relazione è vera o falsa (es. se X=2, Y=4), altrimenti se dichiaro una sola variabile, il sistema mi genera i valori per cui è verificata l'equazione.

Esprimere i numeri con la notazione successore è molto scomodo: il Prolog accetta i numeri reali nella notazione decimale (l'equivalente del Double).

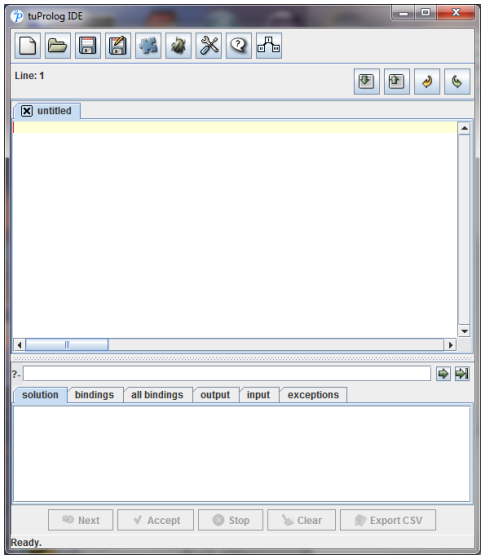
Per fare calcoli però bisogna mettere in pista la semantica dei numeri reali, quindi per dire "uguale =" bisogna utilizzare il **predicato speciale "is"**, che è embedded.

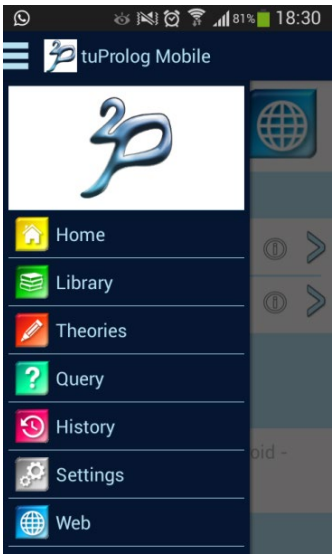
Ecco allora che, mentre “Value = 13-4” mi dà come risultato “13-4”, **“Value is 13-4” mi dà come risultato 9** perché viene fatta una **valutazione semantica** del lato destro, del dominio dei numeri reali.

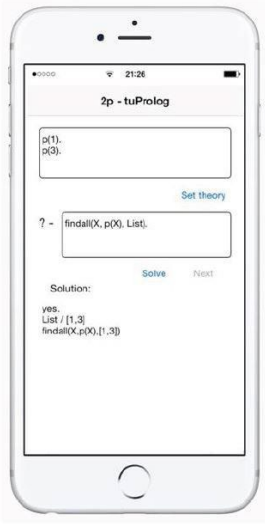
3.1.1 - STRUMENTI

Esistono diversi sistemi Prolog, di cui alcuni gratuiti, ***es.*** SWI-Prolog e GNU Prolog.

Esiste poi il programma **tuProlog**, creato dall’università di Bologna: in questo programma c’è una coesistenza tra Java e Prolog, e posso usare il linguaggio che preferisco in base a quel che devo fare. La licenza è opensource ed è multi-piattaforma.







4 - AUTOMI RICONOSCITORI

Abbiamo detto nel capitolo 2.4. che i linguaggi generati da grammatiche di Tipo 0 possono in generale non essere decidibili, mentre quelli generati da grammatiche di Tipo 1, 2 e 3 sono riconoscibili.
Per realizzare traduttori efficienti si usano linguaggi generati da grammatiche di Tipo 2 (Parser), mentre per ottenere particolare efficienza in sotto-parti di uso estremamente frequente si adottano linguaggi generati da grammatiche di Tipo 3 (Scanner).

GRAMMATICA	AUTOMI RICONOSCITORI
TIPO 0	Macchina di Turing, se L(G) è riconoscibile
TIPO 1	Macchina di Turing con nastro
TIPO 2	Push-Down Automaton (PDA) o ASF + stack
TIPO 3	Automa a stati finiti (ASF)

4.1 - RICONOSCITORI A STATI FINITI (RSF)

Un linguaggio regolare (tipo 3) è riconoscibile da un Automa a Stati Finiti (ASF) e il **Riconoscitore a Stati Finiti (RSF)** è una **specializzazione dell'ASF**.
Riprendendo la definizione di ASF, facciamo un confronto.

DEFINIZIONE DI ASF

⟨I, O, S, mfn, sfn⟩

dove
- I = insieme dei simboli di ingresso
- O = insieme dei simboli di uscita
- S = insieme degli stati
- mfn: I × S → O *(machine fun.)*
- sfn: I × S → S *(state function)*

DEFINIZIONE DI RSF

⟨A, S, S_0, F, sfn*⟩

dove
- A = alfabeto *(A* = chiusura)*
- S = insieme degli stati
- S_0 = stato iniziale ∈ S
- F = insieme degli *stati finali* ⊆ S
- sfn*: A* × S → S *(state function)*

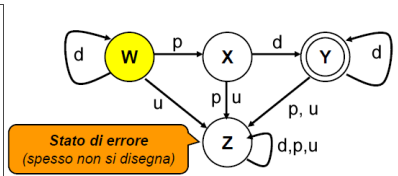
sfn* a noi serve per dire che dopo aver elaborato una certa stringa, saremo in un certo stato: noi sappiamo che la macchina elaborerà un carattere alla volta, è solo una notazione per gli umani.
Più formalmente, sfn* **definisce l'evoluzione dell'automa a partire dallo stato iniziale s0 in corrispondenza di ogni sequenza di ingresso x∈A*.**
Essa è definita in termini della funzione sfn: AxS→S, cioè tratta una sequenza di simboli di A (una stringa).
Per una stringa lunga N, avremo N stati + quello finale = N+1.
Si pone quindi $\forall s \in S$, sfn*(ε,s) = s, sfn*(xa, s) = sfn (a, sfn*(x, s)) dove x∈A* e a∈A.

FRASE ACCETTATA: **frase** x∈A* **che porta il riconoscitore, a partire dallo stato iniziale, in uno stato finale, ovvero sfn*(x, so)** ∈ F**.** Questo significa che, se quando inserisco delle frasi alla fine non sono in uno stato finale, allora tali frasi non sono riconosciute nel linguaggio.

Esempio
Quella in verde assomiglia alla tabella delle transizioni. Nel pallogramma lo stato iniziale è indicato con un pallino giallo, mentre lo stato finale è costituito di due cerchi.

ESEMPIO di RSF
Sia:
- A = { d, p, u }
- S = { W, X, Y, Z }
- S_0 = W
- F = { Y }
- sfn: A × S → S

	d	p	u
W	W	X	Z
X	Y	Z	Z
Y	Y	Z	Z
Z	Z	Z	Z



stato iniziale stato finale

Frasi accettate: dpd, dddpd, pd, pdd, ...
Frasi non accettate: dp, u, dpdp...

Le frasi accettate dal linguaggio possono iniziare per 0+ "d" o per 1 "p". In base al carattere andremo a finire in un determinato stato. Se alla fine della frase ci troviamo nello stato Y, significa che la frase è accettata nel linguaggio. **Lo stato Z è uno stato di errore** da cui non si può uscire: è il male, il pozzo del diavolo.

Il linguaggio L(R) accettato dal riconoscitore R è → infinito se la rappresentazione grafica presenta cicli
→ finito se non ce ne sono

TEOREMA DEL "CACCIA ALLO STATO FINALE"

TEOREMA 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

Tanto per cominciare, il riconoscitore accetta almeno una stringa oppure neanche quella? Si potrebbero creare degli automi che ritornano sempre "no", automi che rifiutano tutto o che qualsiasi percorso tu scelga, fanno a finire in uno stato di errore. Se l'automa finito ha N stati, lo stato finale deve essere al massimo dopo N-1 passi.

Dimostrazione: Se L(R) non è vuoto, R accetta una stringa x. Se L_x>N (se la stringa è più lunga di N), l'automa R deve passare più di una volta per uno stesso stato s_1, cioè deve esserci un ciclo da qualche parte.
Se è così, allora esiste una stringa con lunghezza L_y<L_x in cui sono stati eliminati tutti i cicli che porta allo stato finale.

TEOREMA DEL "CACCIA AL CICLO"

TEOREMA 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.

Se ho un automa a N stati e il risultato è una stringa a N+1 elementi, da qualche parte c'è nascosto un ciclo! Sarà all'ultimo passo, o al penultimo, o al terzultimo, o al quartultimo... o nel caso peggiore al primo passo, quindi alla peggio avrò una stringa lunga 2N-1.

Possiamo verificare se un linguaggio accettato è infinito o meno in un tempo finito… Magari molto lungo, ma finito. Cioè, **decidere se un linguaggio regolare sia vuoto o infinito è un problema risolubile:**

- Per decidere se un linguaggio regolare è vuoto, basta vedere se esiste una stringa accettata di lunghezza minore di N
- Per decidere se esso è infinito, basta verificare se esiste una stringa accettata fra quelle di lunghezza compresa tra N e 2N-1.

Nel Tipo 3 non ci sono problemi di decidibilità, quindi queste proprietà sono decidibili nel Tipo 2, mentre non lo sono nel Tipo 1 e 0.

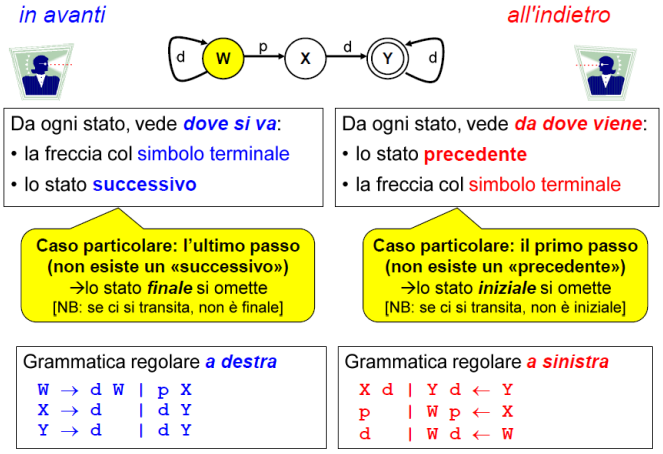
Riassumendo: se immagino l'automa come una black box, posso solo fare esperimenti: se so che ha 5 stati, l'unico esperimento che posso fare è vedere se esiste uno stato finale; se c'è, vuol dire che può essere nel primo, nel secondo, nel terzo, nel quarto o nel quinto stato. Se lo stato finale è quello iniziale vuol dire che la lucina è sempre verde, ancora prima di iniziare. Se è il secondo, vuol dire che accetta stringhe lunghe 1, se è il terzo le accetta lunghe 2 e così via. Alla peggio, con N stati, accetterà stringhe lunghe N-1. Bisogna provare tutte le possibili stringhe lunghe X combinando i vari caratteri dell'alfabeto.
Se tutte le prove che facciamo dicono sempre "no", significa che la macchina non ha uno stato finale, quindi la lucina verde non si accenderà mai.
Per capire se un automa accetta stringhe finite bisogna sapere se ha dei cicli dentro, ma se ho una black box, come faccio? Faccio il contrario e cioè con 5 stati vuol dire che l'automa può accettare una stringa lingua 5, 6, 7, ecc. ecc. quindi potrei avere un ciclo lungo 1, lungo 2, lungo 3... Se ho 3 stati e ho un ciclo lungo 2, vuol dire che accetto 3, 5, 7, 9… fino ad arrivare a 2N-1.

4.1.1 - DAI RICONOSCITORI AI GENERATORI

Nel riconoscitore di 4.1 abbiamo che l'automa può accettare, in base allo stato in cui si trova, determinati simboli. Se sostituiamo la parola "accettare" con "generare", otteniamo non più un riconoscitore di frasi, ma un **generatore**: infatti **le frasi sono interpretabili come produzioni di una qualche grammatica.**
Si può dunque automatizzare la costruzione di un RSF a partire dalla grammatica – o viceversa, risalire ad una grammatica dato un RSF.
Generare è sinonimo della parola produrre perciò queste frasi sono mappabili sulle regole di una grammatica:

Mapping tra:	**Stati**	**↔**	**Simboli non terminali**
	Transizioni	**↔**	**Produzioni**
	Scopo	**↔**	**Uno stato particolare**

DAL RSF ALLA GRAMMATICA INTUITIVAMENTE
Si immagini un osservatore che, stando in ogni stato, guardi rispettivamente:



4.2 - MAPPING FRA GRAMMATICHE E RICONOSCITORI

AUTOMA RICONOSCITORE		GRAMMATICA
Stati dell'automa	↔	**Metasimboli** della grammatica
Transizioni dell'automa	↔	**Produzioni** della grammatica
Uno stato dell'automa	↔	**Scopo** della grammatica

Tale mapping presenta alcuni gradi di libertà:

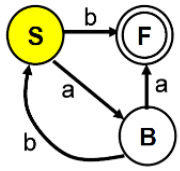
- Se la grammatica è regolare a **destra** → si ottiene un automa riconoscitore **top down**
- Se la grammatica è regolare a **sinistra** → si ottiene un automa riconoscitore **bottom up**

4.2.1 - RICONOSCITORI TOP DOWN

Data una grammatica regolare lineare a **destra**, il riconoscitore:

- **Ha tanti stati quanti i simboli non terminali**
- Ha come **stato iniziale lo scopo S:** quindi nella derivazione si parte dallo scopo
- **Per ogni regola del tipo X→xY,** l'automa, con ingresso x, si porta dallo stato X allo stato Y
- **Per ogni regola del tipo X→x**, l'automa, con ingresso x, si porta dallo stato X allo stato finale F

DERIVAZIONE TOP-DOWN: Si parte dallo scopo della grammatica e si tenta di coprire la frase data tramite produzioni successive.



Esempio

Se G è una grammatica lineare a destra caratterizzata dalle produzioni

S → aB | b B → bS | a

Ottengo l'automa top-down in due schemi equivalenti: uno con uno stato finale, uno con due stati finali. È indifferente in realtà, ma con la riduzione è più comodo perché si ha uno stato finale soltanto. Il nome dello stato finale posso anche non metterlo (quindi non indicheremo F).

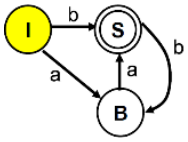
Viene riconosciuta la frase "abaa" partendo dallo scopo S e generando via via la forma di frase, trovandosi, a simboli esauriti, nello stato finale: S → aB → abS → abaB → abaa

4.2.2 - RICONOSCITORI BOTTOM UP

Data una grammatica regolare lineare a **sinistra**, il riconoscitore:

- **Ha tanti stati quanti i simboli non terminali**
- **Ha come stato finale lo scopo S:** quindi nella derivazione si termina nello scopo
- **Per ogni regola del tipo X→Yx**, l'automa, con ingresso x, si porta dallo stato Y allo stato X
- **Per ogni regola del tipo X→x,** l'automa, con ingresso x, si porta dallo stato iniziale I allo stato X

DERIVAZIONE BOTTOM-UP: Si parte dalla frase data e si risale verso lo scopo S tramite riduzioni successive.



Esempio

Partendo dalla grammatica: S → Ba|b B → Sb|a

Bisogna leggere: come arrivo allo stato S? Se sono nello stato B ci arrivo con "a" oppure se sono nello stato iniziale ci arrivo con "b". E come arrivo in B? Se sono nello stato S ci arrivo con "b" e se sono nello stato iniziale ci arrivo con "a".

La frase "aaba" è riconosciuta partendo dallo stato iniziale e riducendosi ogni volta ad una forma di frase più corta fino a risalire allo scopo S: aaba → Baba → Sba → Ba → S

4.2.3 - DALL'AUTOMA ALLE GRAMMATICHE

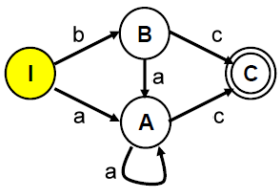
Dato un automa riconoscitore se ne possono trarre:

- Una grammatica regolare a destra interpretandolo top-down (**APPROCCIO A GENERAZIONE**)
- Una grammatica regolare a sinistra interpretandolo bottom-up (**APPROCCIO A RIDUZIONE**)

Esempio

In questo caso si ha uno stato finale e quindi è una situazione ideale, inoltre né lo stato iniziale né lo stato finale hanno archi in ingresso.

Es. aa*c = a*c perché aa* significa "'a' seguito da almeno una 'a'"



Analisi *top-down* (***C** finale, si omette*):
I → b B B → a A B → c
I → a A A → a A A → c

Grammatica *G1 regolare a destra*:
I → b B | a A
B → a A | c
A → a A | c

L(*G1*) =
b L(B) + a L(A) = (ba+a) L(A) + bc =
= (ba+a)a*c + bc = ba*c + aa*c =
= (b + a) a* c

Analisi *bottom-up* (***C** iniziale, **C** ≡ **S***):
b ← B B a ← A B c ← S
a ← A A a ← A A c ← S

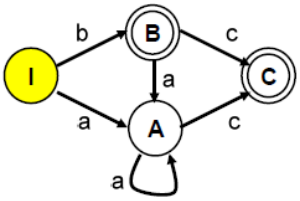
Grammatica G2 regolare a sinistra:
S → B c | A c
A → B a | A a | a
B → b

L(*G2*) =
L(B) c + L(A) c = (b + L(A)) c =
(b + (b a + a) a*) c = (b + a) a* c
perché (b + b a a*) = b a*

Quasi sempre questi conti non li faremo perché mapperemo gli automi in grammatiche e grammatiche in automi, senza passare dalle espressioni regolari perché non sono immediate.

Esempio

Se ho più stati finali, è difficile fare l'analisi bottom-up perché non so da dove iniziare e che scopo adottare. Infatti l'analisi top-down non dà problemi, ma l'analisi bottom-up non sa più che scopo adottare, B o C?



Analisi *top-down (c'è una regola in più)*:
$I \to b\,B \mid b \quad B \to a\,A \quad B \to c$
$I \to a\,A \quad A \to a\,A \quad A \to c$

Analisi *bottom-up* (*B,C iniziali... S ??*):
$b \leftarrow B \quad B\,a \leftarrow A \quad B\,c \leftarrow C$
$a \leftarrow A \quad A\,a \leftarrow A \quad A\,c \leftarrow C$

Grammatica G2' (assume $C \equiv S$):
$S \to B\,c \mid A\,c$
$A \to B\,a \mid A\,a \mid a$
$B \to b$
$L(G2') = (b + a)\,a^*\,c$

Grammatica G2'' (assume $B \equiv S$):
$b \leftarrow S$
le altre regole sono inutili, essendo i loro metasimboli irraggiungibili !
$L(G2'') = b$

Grammatica G2 (riunificazione di G2' e G2''):
$S \to B\,c \mid A\,c \mid b$
$A \to B\,a \mid A\,a \mid a \qquad B \to b$
$L(G2) = (b + a)\,a^*\,c + b$

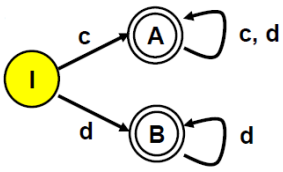
Attenzione: rimappando G2 in un automa, esso potrebbe risultare *non-deterministico.*

Bisogna allora **esprimere il linguaggio riconosciuto o generato come unione di due sotto-linguaggi**: un linguaggio andrà verso B, l'altro verso C.

La grammatica G2' è uguale a quella di prima, mentre nella grammatica G2'' abbiamo inserito una sola regola in più.

Più in generale:

Nel caso bottom-up, *in presenza di più stati finali:*
- si assume come (sotto-)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come *unione dei vari sotto-linguaggi*, definendo lo scopo globale come $S \to S_1 \mid S_2 \mid \ldots \mid S_n$



Esempio

Assumendo S1≡A: A→Ac | Ad | c Assumendo S2≡B: B → Bd | d
Linguaggio complessivo: S → S1 | S2 = A | B
Che è diverso da S → Sc | Sd | Sd | c | d (genera "dc")
Sarebbe quindi S → Ac | Ad | Bd | c | d (non genera "dc")

Due approcci:

- Il mapping tratta in modo particolare lo stato finale nel caso top-down o lo stato iniziale nel caso bottom-up: ciò garantisce che le produzioni non presentino mai ε-rules.
- Si può anche evitare di inserire uno stato iniziale/finale, accettando la presenza di produzioni con stringhe vuote.
  - **Pro**: un solo tipo di mapping, valido sia per il caso generale, sia per i casi particolari
  - **Contro**: si hanno produzioni con le ε-rules, da eliminare in un secondo tempo

4.3 - IMPLEMENTAZIONE DI RSF DETERMINISTICI

Un <u>riconoscitore a stati finiti deterministico</u> è facilmente realizzabile in un linguaggio imperativo, ma le scelte che si fanno possono **influenzare fortemente l'estendibilità, la modularità e la riusabilità.**

- Ciclo while con if annidati
  - Automa cablato nel codice
  - Estendibilità nulla
  - Scarsa leggibilità

- Ciclo while con switch
  - Automa cablato nel codice
  - Poca estendibilità
  - Un po' più leggibile

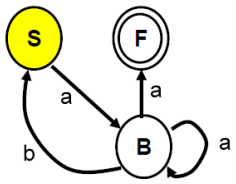
- Ciclo while con tabella separata
  - Automa non cablato nel codice
  - Poco estendibile
  - Ottima leggibilità

Nel terzo caso posso leggere la descrizione dell'automa dall'esterno, quindi è un po' più riutilizzabile.

4.3.1 - AUTOMI E RICONOSCITORI NON DETERMINISTICI

Certe grammatiche possono portare ad un **AUTOMA NON DETERMINISTICO: per ogni frase di L(G) esiste almeno una computazione che porta l'automa dallo stato iniziale S allo stato finale F,** cioè con una **tabella di transizioni con più stati futuri per la stessa configurazione.**
Un automa riconoscitore non deterministico dev'essere intrinsecamente dotato della capacità di scegliere in quale stato portarsi, quando ha più alternative.

Esempio
Lo stato B ha due strade: o vado in F o rimango in B. Questo tipo di situazione non è deterministico perché costringe la macchina ad effettuare una scelta, ma la macchina non è in grado di tornare indietro se sbaglia strada.



	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Se il linguaggio supporta il non determinismo, come i **linguaggi dichiarativi**, **implementare il riconoscitore è semplice** perché ogni produzione della grammatica diventa una regola e la VM del linguaggio può tornare indietro e provare le possibili alternative.
In un linguaggio imperativo, invece, **occorre costruirsi a mano tutte le strutture per "ricordare la strada" e mettere in piedi il "motore" per gestirle**, cioè bisogna ricostruirsi la stessa capacità che il motore Prolog ha già innata.

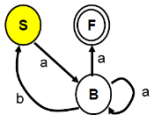
4.3.2 - DA AUTOMI NON DETERMINISTICI AD AUTOMI DETERMINISTICI

TEOREMA:
un automa non deterministico può sempre essere *ricondotto a un automa deterministico equivalente* (eventualmente minimizzabile secondo le regole usuali).

Per rendere deterministico un automa che di per sé non lo è, bisogna seguire un **procedimento preciso:**

1. Si definisce un automa i cui stati corrispondano a set di stati dell'automa originale
2. Si costruisce una tabella delle transazioni del nuovo automa aggiungendo righe via via che si analizzano nuovi casi. (Attenzione: stati finali sono sempre distinti da stati non finali)
3. Si rinominano gli stati in modo significativo
4. Si minimizza la tabella delle transizioni

Esempio



Originale:

	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Tabella nuovo automa:

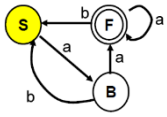
	a	b
[S]	[B]	[E]
[B]	[B,F]	[S]
[B,F]	[B,F,E]	[S,E]
[S,E]	[B,E]	[E]
[B,E]	[B,F,E]	[S,E]
[B,F,E]	[B,F,E]	[S,E]
[E]	[E]	[E]

Rinominata:

	a	b
S0	S1	S6
S1	S2	S0
S2	S5	S3
S3	S4	S6
S4	S5	S3
S5	S5	S3
S6	S6	S6

Minimizzata:

	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6



Minimizzazione:

S1	S1/S2,S0/S6				
S2	X	X			
S3	S1/S4	S2/S4,S0/S6	X		
S4	S1/S5,S3/S6	S2/S5,S0/S3	X	S4/S5,S3/S6	
S5	X	X		X	X
S6	S1/S6	S2/S6,S0/S6	X	S4/S6	S5/S6,S3/S6	X
	S0	S1	S2	S3	S4	S5

Stati indistinguibili:
- **S2/S5 (finali)**
- **S0/S3, S1/S4**

MINIMIZZAZIONE

1. Si crocettano tutti gli stati finali perché chi fa coppia con uno stato finale è uno stato indistinguibile.

S1						
S2	X	X				
S3			X			
S4			X			
S5	X	X		X	X	
S6			X			X
	S0	S1	S2	S3	S4	S5

2. Si guarda dove finiscono gli stati con "a" e con "b", uno alla volta.

Lo stato S1 con "a" va in S2, con "b" va in S0. Lo stato S6 sia con "a" sia con "b" va in S6.

S1						
S2	X	X				
S3			X			
S4			X			
S5	X	X		X	X	
S6		S2/S6, S0/S6	X			X
	S0	S1	S2	S3	S4	S5

Lo stato S3 con "a" va in S4, con "b" in S6. Lo stato S6 con entrambi va in S6.

S1						
S2	X	X				
S3			X			
S4			X			
S5	X	X		X	X	
S6		S2/S6, S0/S6	X	S4/S6		X
	S0	S1	S2	S3	S4	S5

S1	S1/S2, S0/S6					
S2	X	X				
S3	S1/S4	S2/S4, S0/S6	X			
S4	S1/S5, S3/S6	S2/S5, S0/S3	X	S4/S5, S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6, S0/S6	X	S4/S6	S5/S6, S3/S6	X
	S0	S1	S2	S3	S4	S5

3. Eliminiamo le coppie che sono state crocettate

Se guardiamo S1 e S0, abbiamo S1/S2 come stati conseguenti alla ricezione di "a". La coppia S1S2 è crocettata: eliminiamo la cella.

S1	S1/S2, S0/S6					
S2	X	X				
S3	S1/S4	S2/S4, S0/S6	X			
S4	S1/S5, S3/S6	S2/S5, S0/S3	X	S4/S5, S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6, S0/S6	X	S4/S6	S5/S6, S3/S6	X
	S0	S1	S2	S3	S4	S5

Se guardiamo S4 e S3, abbiamo che con "a" si va in S4/S5. Essendo la coppia S4S5 crocettata, eliminiamo.

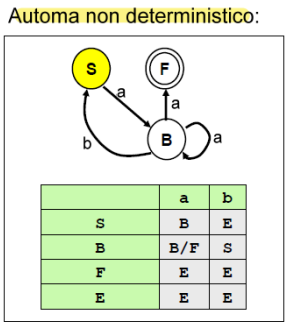
S1	S1/S2, S0/S6					
S2	X	X				
S3	S1/S4	S2/S4, S0/S6	X			
S4	S1/S5, S3/S6	S2/S5, S0/S3	X	S4/S5, S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6, S0/S6	X	S4/S6	S5/S6, S3/S6	X
	S0	S1	S2	S3	S4	S5

Ottenendo così in rosso gli stati che possono essere eliminati e in verde gli stati indistinguibili:

S1	S1/S2, S0/S6					
S2	X	X				
S3	S1/S4	S2/S4, S0/S6	X			
S4	S1/S5, S3/S6	S2/S5, S0/S3	X	S4/S5, S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6, S0/S6	X	S4/S6	S5/S6, S3/S6	X
	S0	S1	S2	S3	S4	S5

4. Ora riscriviamo la tabella delle transizioni con gli stati verdi:

	a	b
S0, S3	S1, S4	S6
S1, S4	S2, S5	S0, S3
S2, S5	S5	S3
S6	S6	S6

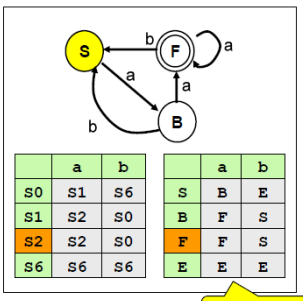


	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6

Automa non deterministico:

	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Automa deterministico minimo:



	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6

	a	b
S	B	E
B	F	S
F	F	S
E	E	E

Stati rinominati

Il nuovo automa riconosce anch'esso le due frasi
`abaa` (sequenza di transizioni: `S → B → S → B → F`)
`abaaa` (sequenza di transizioni: `S → B → S → B → F → F`)

In questo caso abbiamo ottenuto un automa a stati con un numero di stati uguale a quello iniziale, ma non sempre sarà così bello.

4.3.3 - IMPLEMENTARE UN AUTOMA

Una volta tradotta la grammatica**, l'automa deterministico minimo può essere implementato sia in un linguaggio imperativo sia, più rapidamente, in un linguaggio dichiarativo come il Prolog.**

L'uso di un linguaggio che supporta il non-determinismo, come il Prolog, non implica che si debba necessariamente pagare il prezzo del non-determinismo anche quando non è necessario.
Notiamo infatti che, in realtà, se so già di non aver bisogno del non determinismo perché so per certo che non ci sono bivi, allora il **CUT** (!) si può mettere per dire che: "se oltrepassi questo punto, indietro non tornerai" perché si è sicuri che è la strada giusta e non ci sono diramazioni. Si tagliano le alternative.

4.3.4 - DAL RICONOSCITORE AL GENERATORE

Riassunto
Il modello dichiarativo è intrinsecamente invertibile perché ragiona a regole: se fornisco tutti i dati funziona come riconoscitore, se invece lo interrogo con delle variabili, allora può essere utilizzato come generatore perché è lui che fa i calcoli per fare in modo di trovare una stringa che soddisfi le regole della grammatica che ho interrogato.
Quando genera delle risposte una ad una, sei tu che decidi se ti può piacere come stringa generata oppure se la macchina deve andare avanti cercando altre soluzioni. Se il numero di soluzioni è finito, prima o poi le elenca tutte e le esaurisce, ma alcuni linguaggi ammettono infinite frasi, perciò questa cosa andrebbe avanti per sempre.
In questo caso **L'ORDINE DI SCRITTURA DELLE REGOLE È IMPORTANTE** perché **la VM Prolog le esplora nell'ordine in cui sono scritte**: se come prima regola ne ho una che ammette infinite soluzioni, non si arriverà mai alla fine. Ad esempio, la regola che chiude la ricorsione deve essere la prima.

4.4 - ESPRESSIONI, LINGUAGGI REGOLARI E AUTOMI

TEOREMA:
l'insieme dei linguaggi riconosciuti da un ASF coincide con l'insieme dei linguaggi regolari, ossia quelli descritti da *espressioni regolari.*

Espressioni regolari e ASF fanno parte della stessa realtà, ma sono metodi descrittivi appartenenti a due differenti categorie:

- Gli **automi** a stati sono un metodo di **descrizione operazionale** (evidenziano passi computazionali da compiere per riconoscere le frasi)
- Le **espressioni regolari** sono un metodo di **descrizione denotazionale** (un'entità è specificata tramite operatori di tipo matematico)

Riprendendo l'esempio del capitolo 2.5.1, abbiamo detto che per la grammatica seguente abbiamo tre espressioni regolari che possono generarla:

Grammatica data:
S → a B | a S
B → d S | b

LA PRIMA ESPRESSIONE ottenuta:	S = (a* a d)* a* a b
LA SECONDA ESPRESSIONE ottenuta:	S = (a d + a)* a b
UNA TERZA ESPRESSIONE equivalente:	S = a (d a + a)* b

Quali automi verrebbero originati dalla grammatica originale e dalle varie espressioni regolari ottenute?

- **Ogni * equivale ad un ciclo (ε-rule)**
- **Ogni + equivale ai percorsi alternativi**

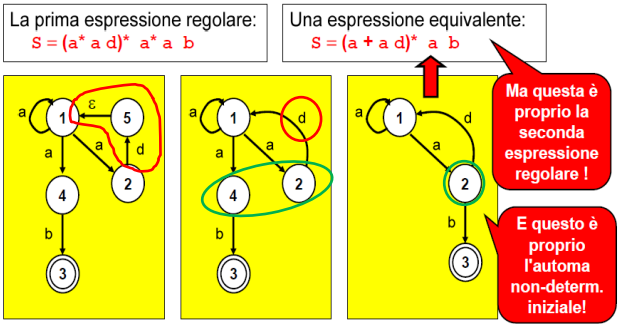
L'operatore * introduce **ε-archi** eliminabili subito dopo con una semplice trasformazione

Esempio

Partendo dalla prima espressione: S = (a* ad)* a* ab, posso distinguere 3 blocchi e fare un pezzetto alla volta, e come passo intermedio fittizio quando vedo un * ci metto un arco come per dire "posso tornare indietro" però con niente, è un arco finto che chiamo ε.

- Primo blocco: (a*ad) → devo avere un automa che mi permetta di fare aaaaaa e poi anche ad.
- Secondo blocco: a* → lo stato 1 è già un a* quindi non ne creo un altro.
- Terzo blocco: ab. Creo altri due stati (4, 3).

Naturalmente, siccome avevo un arco fittizio, lo tolgo in questo caso cassando lo stato 5 e facendo andare lo stato 2 direttamente in 1.



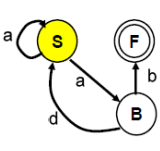
L'automa risultante (figura centrale) non è deterministico perché posso restare in 1, posso andare in 2 o posso andare in 4: ho troppe scelte e implementarlo con un linguaggio imperativo è difficile.

Posso unire lo stato 2 e lo stato 4 perché entrambi accettano "a". In questo modo **ho ottenuto l'automa non deterministico iniziale**.

Cosa succede se rendiamo deterministico e minimo l'automa ottenuto?

I girotondi non possono sparire, semplicemente cambiano posto. **A volte per rendere un automa deterministico, basta posporre la scelta**: in questo modo so intanto cosa scrivere all'inizio, poi più avanti deciderò se tornare indietro con d, se rimanere in S1 con a o se proseguire verso lo stato finale S3.

Partendo dall'automa non deterministico:



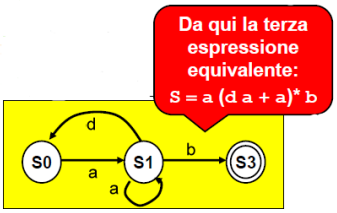
	a	b	d
S	S/B	E	E
B	E	F	S
F	E	E	E
E	E	E	E

Si possono espandere gli stati e cercare di minimizzare l'automa:

	a	b	d	
[S]	[S,B]	[E]	[E]	S0
[S,B]	[S,B,E]	[E,F]	[E,S]	S1
[S,B,E]	[S,B,E]	[E,F]	[E,S]	S2
[E,F]	[E]	[E]	[E]	S3
[E,S]	[S,B,E]	[E]	[E]	S4
[E]	[E]	[E]	[E]	E

Classi di equivalenza:
- **S1 / S2**
- **S0 / S4**

	a	b	d
S0	S1	E	E
S1	S1	S3	S0
S3	E	E	E
E	E	E	E

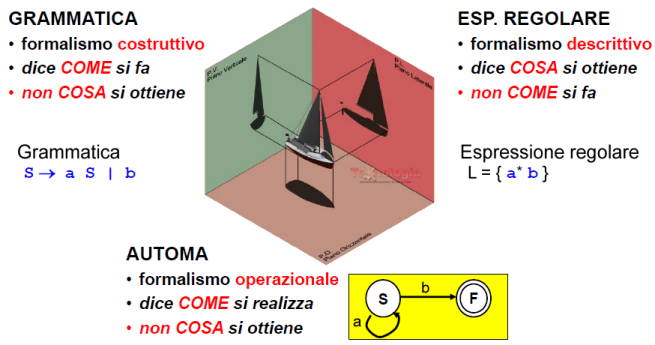


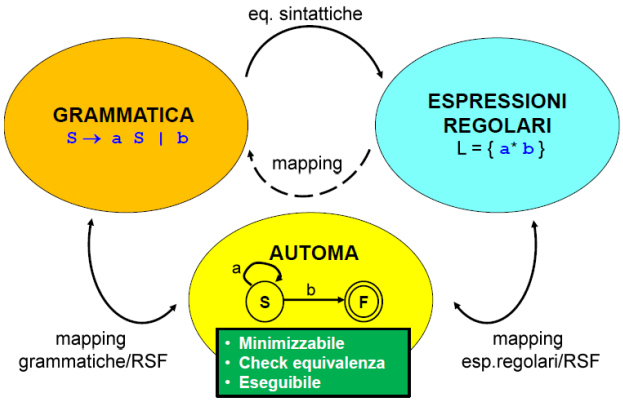
In questo modo otteniamo l'automa equivalente alla terza espressione.

4.5 – I TRE FORMALISMI

Riprendendo il discorso già fatto sulla rappresentazione diversificata della stessa realtà, ecco che finalmente abbiamo tutti e tre i punti di vista della stessa entità.

La grammatica è un formalismo costruttivo perché è un po' come montare i mobili dell'Ikea: segui i passi che sono indicati senza sapere bene cosa aspettarti, ma alla fine arrivi ad un qualcosa.
Le espressioni regolari sono un formalismo descrittivo perché è come vedere la foto del mobile dell'Ikea: sai come verrà ma non sai come costruirlo, che passi seguire.
L'automa è un formalismo operazionale e anche in questo caso, come nella grammatica, sai che passi devi fare ma non sai per ottenere cosa. La cosa vantaggiosa di questa rappresentazione è che **passare dall'espressione regolare all'automa o viceversa è molto semplice.**



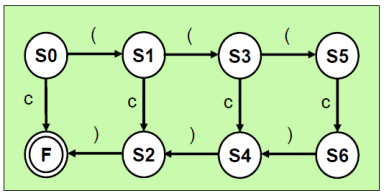


5 - RICONOSCITORI CON PDA

5.1 - LIMITI DEI RICONOSCITORI A STATI FINITI

Un RSF ha un limite intrinseco: non può riconoscere un linguaggio di tipo 2 perché non può riconoscere stringhe di cui non si conosce la lunghezza massima a priori.

Es. il bilanciamento delle parentesi L = { $(^n$ c $)^n$, n≥0 } G = { S → (S) | c }
In questo linguaggio, non sappiamo a priori quante parentesi ci saranno perché non sappiamo il valore di n: potremmo barare mettendo a priori un limite, oppure potremmo non avere questo problema se le parentesi che si aprono non fossero lo stesso numero di quelle che si chiudono, quindi con L = { $(^n$ c $)^m$, n,m≥0 }
Nell'automa a stati disegnato ho solo quattro possibilità ($0 \leq n \leq 3$):



- Non ci sono parentesi
- Ce n'è 1
- Ce ne sono 2
- Ce ne sono 3

Devo avere un limite superiore noto a priori, che in questo caso è 3, per poter disegnare un automa a stati finiti come questo. Se invece devo riconoscere un linguaggio in cui non è specificato quante parentesi saranno usate, manca il parametro di dimensionamento dell'automa, ergo non posso disegnarlo così com'è.

5.2 - PUSH-DOWN AUTOMATON (PDA)

Riprendiamo lo schema del capitolo 2.4. Se pensiamo allo stack come ad una sorta di nastro che va solo in un verso, esso è un buon candidato per essere aggiunto all'ASF come "supporto esterno".

GRAMMATICA	AUTOMI RICONOSCITORI
TIPO 0	Macchina di Turing, se L(G) è riconoscibile
TIPO 1	Macchina di Turing con nastro
TIPO 2	Push-Down Automaton (PDA) o ASF + stack
TIPO 3	Automa a stati finiti (ASF)

Per riconoscere un linguaggio di Tipo 2 serve un nuovo tipo di automa, che superi il limite di memoria finita (limitata a priori) del RSF. Tale automa dovrà necessariamente appoggiarsi a una **struttura dati esterna** ad esso, **espandibile** quanto serve (Es. uno STACK). Allora:

PDA (PUSH-DOWN AUTOMATON) = RSF + STACK

Poiché parliamo di macchine astratte, **con il termine "stack"** non si fa riferimento ad una struttura dati fisica, ma ad un **modello astratto**:

- **Lo stack è definito formalmente come una sequenza di simboli** definita in modo che si possa operare solo sul simbolo più a destra, che è quello "in cima" alla pila
- Come un RSF, **un PDA legge un simbolo d'ingresso e transita in un nuovo stato; in più, ad ogni passo altera lo stack**, producendo una nuova configurazione
- **Un PDA può o meno prevedere ε-mosse**, sorta di transizioni spontanee che manipolano lo stack senza consumare simboli di ingresso (qui le ε-mosse potrebbero non essere eliminabili)

Un PDA è definito da una sestupla ed ha le seguenti caratteristiche:

- **Lo stato futuro dipende anche da quello che c'è sullo stack.** Esso è fatto contemporaneamente dalla pop del vero stato futuro e dalla push di certe cose (eventualmente dalla push di ε)
- **sfn → in certi casi**, dato un certo stato e data una certa cosa sullo stack, **la macchina si mantiene il diritto di cambiare stato e cambiare lo stack anche senza leggere dall'esterno**. Questo serve se voglio "sistemare" la macchina tra una lettura e l'altra

DEFINIZIONE DI PDA

Un PDA è una *sestupla:*

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

dove

- A = alfabeto
- S = insieme degli stati
- S_0 = stato iniziale $\in S$
- $sfn: (A \cup \varepsilon) \times S \times Z \to W$ con W sottoinsieme finito di $S \times Z^*$
- Z = alfabeto dei *simboli interni*
- $Z_0 \in Z$ = *simbolo iniziale* sullo stack

- **Questa definizione include il caso delle ε-mosse**: per escluderle si definisce sfn sul dominio AxSxZ.
- **Z → lo stack manipola simboli di un alfabeto spesso diverso da quello delle frasi del linguaggio.** Quando catturo qualcosa dall'input, poi spesso uso un alfabeto interno da mettere nello stack perché copiare quello in input potrebbe essere limitativo
- **Z0 → all'inizio lo stack non può essere vuoto**, se no appena chiamo la sfn avrei quella che in Java sarebbe la NullPointerException()

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

- **CRITERIO DELLO STATO FINALE**: come in un RSF, il linguaggio accettato è l'insieme di tutte le stringhe di ingresso che portano il PDA in uno degli stati finali. **Un PDA diventa allora una settupla** perché bisogna aggiungere l'insieme degli stati finali
- **CRITERIO DELLO STACK VUOTO**: appoggiandosi al nuovo concetto di stack, il linguaggio accettato è definito come l'insieme di tutte le stringhe di ingresso che portano il PDA nella configurazione di stack vuoto

La funzione **sfn**, dati:

- Un simbolo di ingresso a∈A
- Lo stato attuale s∈S
- Il simbolo interno attualmente al top dello stack z∈Z

Opera come segue:

- **Consuma il simbolo d'ingresso** a
- **Effettua una POP dallo stack** prelevando il simbolo interno al top (z)
- **Porta l'automa nello stato futuro s'**dove s' = $\mathrm{sfn}(a, s, z)_{\prod S}$ (proiezione su S)
- **Effettua una PUSH sullo stack** di 0+ simboli interni z'∈Z* dove z' = $\mathrm{sfn}(a, s, z)_{\prod Z}$ (proiezione su Z)

Esempio: riconoscitore di stringhe palindrome

Si consideri il linguaggio generato da:
Alfabeto: A = { 0, 1, c }
Produzioni: P = { S → 0 S 0 | 1 S 1 | c }
Linguaggio: L = { *word* c *word*^{R} }
dove *word*^{R} indica il ribaltamento di *word* e *word*, che a sua volta indica tutte le possibili sequenze di 0 e 1, inclusa la stringa vuota ε.

Questo automa accetta L con il criterio dello stack vuoto.
Q1 è lo stato di accumulo in cui sono memorizzati i simboli prima della "c" centrale, mentre **Q2 è lo stato di svuotamento** in cui si recuperano i simboli dello stack e si verifica che corrispondano a quelli in input.
Si commuta da Q1 a Q2 quando si trova l'elemento centrale "c".

Se alla fine lo stack si è svuotato, vuol dire che è andato tutto bene, se invece esplode vuol dire che sto chiudendo più cose di quelle che avevo aperto e, infine, se rimangono elementi, vuol dire che sto chiudendo meno cose di quelle che erano state aperte.
Molto spesso nei PDA gli stati sono solo due perché ci servono per commutare da uno stato all'altro, ma poi si lavora molto sullo stack.

"01c10": questa frase è legittima?

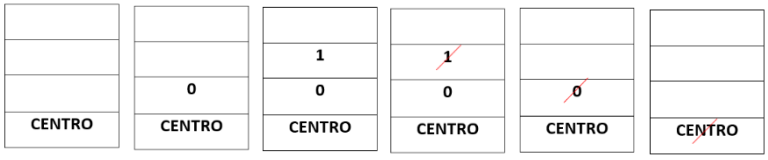
1. Stato = Q1 (accumulo)
   Nello stack c'è "c" → POP di c, PUSH di c, PUSH di c0
2. Leggo 1, Stato = Q1
   Sullo stack c'è c0 → POP di 0, PUSH di 0, PUSH di 1
3. Leggo C, sullo stack c'è c01
   Non metto sullo stack "c", tanto da ora so che dovrò cancellare
4. Leggo 1, stato = Q2, sullo stack c'è c01 → POP di 1
5. Leggo 0, stato = Q2, sullo stack c'è c0 → POP di 0
6. Se il prossimo carattere è ε e dentro c'è il simbolo iniziale, posso cancellare.
   Leggo ε, stato = Q2, sullo stack c'è c → POP di c
   Ora lo stack è vuoto, cioè la frase è effettivamente palindroma.

Definiamo il PDA come segue:

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

A = { 0, 1, c }
S = { Q1=S_0, Q2 }
Z = { Zero, Uno, Centro }, Z_0 = Centro

A ∪ ε	S	Z	S x Z*
0	Q1	Centro	Q1 x CentroZero
1	Q1	Centro	Q1 x CentroUno
c	Q1	Centro	Q2 x Centro
0	Q1	Zero	Q1 x ZeroZero
1	Q1	Zero	Q1 x ZeroUno
c	Q1	Zero	Q2 x Zero
0	Q1	Uno	Q1 x UnoZero
1	Q1	Uno	Q1 x UnoUno
c	Q1	Uno	Q2 x Uno
0	Q2	Zero	Q2 x ε
1	Q2	Uno	Q2 x ε
ε	Q2	Centro	Q2 x ε



5.3 - PDA NON DETERMINISTICI

Anche un PDA può essere non deterministico: qui **la funzione sfn produce insiemi di elementi di W** dove W è sottoinsieme finito di SxZ*. Un PDA è non deterministico se **l'automa in uno stato Q_i**, con simbolo interno in cima allo stack Z e ingresso a, **ha più di uno stato futuro possibile**: in base alla evoluzione, cambia anche il set di simboli da porre sullo stack.
Nella definizione, **include il caso delle ε-mosse** (per escluderle, basta avere sfn sul dominio AxSxZ).

DEFINIZIONE DI PDA

Un PDA è una *sestupla:*

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

dove
- A = alfabeto
- S = insieme degli stati
- S_0 = stato iniziale ∈ S
- sfn: $(A \cup \varepsilon) \times S \times Z \rightarrow W$
  con W sottoinsieme finito di $S \times Z^*$
- Z = alfabeto dei *simboli interni*
- $Z_0 \in Z$ = *simbolo iniziale* sullo stack

DEFINIZIONE DI PDA *nondet*

Un PDA *nondet* è una sestupla:

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

dove
- A = alfabeto *(A* = chiusura)*
- S = insieme degli stati
- S_0 = stato iniziale ∈ S
- sfn: $(A \cup \varepsilon) \times S \times Z \rightarrow W^k$
  con W sottoinsieme finito di $S \times Z^*$
- Z = alfabeto dei *simboli interni*
- $Z_0 \in Z$ = *simb. iniz.* sullo stack

In alcuni casi riusciremo a trasformare un PDA non deterministico in un PDA deterministico, in altri casi no.

Il non-determinismo dell'automa può emergere sotto due aspetti:

- L'automa dallo stato Qi, con simbolo interno in cima allo stack Z, e con ingresso x, **può leggere o non leggere il simbolo d'ingresso x** (se scatta la transizione spontanea, l'input non viene letto e quindi non è noto)
- L'automa dallo stato Q1, con simbolo interno in cima allo stack Z, e con ingresso x, **può portarsi in più stati futuri**

5.3.1 - VANTAGGI E SVANTAGGI

TEOREMA
La classe dei linguaggi riconosciuti da un PDA non-deterministico *coincide con la classe dei linguaggi context-free*: perciò qualungue linguaggio context free può *sempre* essere riconosciuto da un opportuno PDA nondet.

PRO: In generale un PDA non deterministico riconosce qualunque linguaggio di Tipo 2.
CONTRO: a volte non posso trasformare un PDA non deterministico in un PDA deterministico.

Problemi:

- **La complessità di calcolo del PDA non deterministico può essere esponenziale** con algoritmo non ottimizzato: se ho una stringa più lunga di qualche unità, sono spacciato.
- **I migliori algoritmi hanno complessità dell'ordine del cubo della lunghezza della stringa da riconoscere**, che si riduce al quadrato se la grammatica non è ambigua (è comunque sempre un quadrato, quindi 100^2 anziché 100^3 è meglio, ma è sempre 100^2).

5.4 - VERSO PDA DETERMINISTICI

Riassunto: abbiamo visto il concetto di PDA come automa a stati. Siccome l'automa ha bisogno di sapere a priori il numero di stati che bisogna usare, si può usare solo in linguaggi particolari.
Se abbiamo dei cicli o dei costrutti in cui ci sono tre parti, non può essere gestito da un automa finito: viene aggiunto lo stack. In generale gli stati a questo punto diventano almeno due, ma generalmente sono comunque pochi.
Nello stack devo mettere un elemento iniziale, se no quando faccio la prima POP esplode tutto.
Spesso si ha bisogno di mosse ε, che sono archi fittizi che comportano un cambiamento di stato.
Un PDA può essere non deterministico, come l'automa a stati, solo che l'automa a stati può essere **sempre** trasformato in un automa deterministico, mentre un PDA **non sempre** può diventare deterministico.

TEOREMA
Esistono linguaggi *context-free riconoscibili soltanto da PDA non-deterministici*.

In generale per riconoscere qualunque linguaggio, probabilmente servirà un PDA non deterministico. La soluzione non sarà efficiente perché ci saranno degli errori e quindi bisognerà tornare indietro per trovare la strada giusta.
In molti casi di interesse pratico, esistono linguaggi di tipo 2 riconoscibili da PDA deterministici (linguaggi context-free deterministici), che hanno una complessità lineare. Da ora in avanti vogliamo solo linguaggi riconoscibili da PDA DETERMINISTICI, che sono solo un sottoinsieme, ma sono quelli che ci interessano.

5.5 – PDA DETERMINISTICI

Per ottenere un PDA deterministico, viste le condizioni precedenti, **non deve succedere che l'automa, in un dato stato Q0, con simbolo in cima allo stack Z e ingresso x, possa:**

- **Portarsi in più stati futuri**
  sfn(Qi, x, Z) = {(Q1,Z1), (Q2,Z2), …, (Qk, Zk) }
- **Optare se leggere o non leggere il simbolo di ingresso x** a causa della presenza di entrambe le mosse sfn(Qi,x,Z) e sfn(Qi,ε,Z) di cui la seconda è una ε-mossa

Bisogna allora **capire come tradurre questi vincoli sulla grammatica** per assicurarsi che il risultato sia un linguaggio deterministico.

5.5.1 – PROPRIETÀ DEI PDA DETERMINISTICI

Spesso la gente mischia linguaggi più piccoli in un linguaggione grande, quindi se faccio un **MIX FRA LINGUAGGI DETERMINISTICI** devo sapere che:

- **L'unione, l'intersezione e il concatenamento** di due linguaggi deterministici non dà necessariamente luogo a un linguaggio deterministico
- **Il complemento** di un linguaggio deterministico invece è deterministico

Se invece faccio un **MIX FRA LINGUAGGI DETERMINISTICI E REGOLARI**, devo sapere che:

- Se L è un linguaggio deterministico e R un linguaggio regolare, il **linguaggio quoziente L/R** (ossia l'insieme delle stringhe di L private di un suffisso regolare) è deterministico
- Se L è un linguaggio deterministico e R un linguaggio regolare, il **concatenamento L.R** (ossia l'insieme delle stringhe di L con un suffisso regolare) è deterministico

Quindi, mischiare due linguaggi di tipo 2 non vuol dire ottenere un linguaggio di tipo 2. Ma se mischio un linguaggio di tipo 2 con uno di tipo 3, normalmente ottengo un linguaggio di tipo 2.

Esempio - Unione
$L1 = \{ a^n b^n, n>0 \}$ deterministico, $L2 = \{ a^n b^{2n}, n>0 \}$ deterministico, $L1 \cup L2$ non è deterministico
Consideriamo come dovrebbe operare il riconoscitore:

- Prima, dovrebbe porre sullo stack le “a” che trova (diciamo che siano N)

Poi, dovrebbe:

- Se la stringa appartiene a L1, estrarre una “a” quando trova una “b”
- Se la stringa appartiene a L2, estrarre una “a” quando trova due “b”

MA, per decidere quale usare dovrebbe sapere quante sono le b, cosa impossibile perché dovrebbe esaminare una parte illimitata della stringa. Quindi, il solo modo per riconoscere il linguaggio-unione è seguire entrambe le vie, non-deterministicamente.

5.5.2 – SOTTOCLASSI PARTICOLARI

Passando da PDA non-deterministico a deterministico vengono meno alcune proprietà. In particolare, per un PDA deterministico:

- Il criterio dello stack vuoto risulta meno potente del criterio degli stati finali
- Una limitazione sul numero di stati interni o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- L'assenza di ε-mosse riduce l'insieme dei linguaggi riconoscibili

Per realizzare un PDA deterministico si può seguire la definizione oppure si può **adottare un approccio che manipoli uno stack con la stessa logica di un PDA**: un PDA alla fine della fiera cosa fa? Ci sono due stati che vogliono dire: con uno accumuli, con uno svuoti. Allora, qualunque strumento concreto che manipola uno stack è un ottimo candidato.
Infatti, la presenza dello stack è la vera differenza rispetto al RSF: una macchina virtuale che abbia uno stack può essere fatta funzionare come un PDA pilotandola "opportunamente".

5.5.3 - ANALISI RICORSIVA DISCENDENTE (TOP-DOWN RECURSIVE-DESCENT PARSING)

In particolare, **si potrebbe pilotare uno stack** in modo esplicito, "a mano", ma è molto più comodo **sfruttare eventuali costrutti dei linguaggi di programmazion**e che lo facciano per noi → **CHIAMATE (RICORSIVE) DI FUNZIONI.**
I linguaggi di programmazione che supportano chiamate ricorsive di funzioni gestiscono già implicitamente uno stack: possiamo sfruttarlo!
Ogni chiamata di funzione implica l'allocazione sullo stack di un record di attivazione e quando la funzione termina, il record di attivazione viene automaticamente deallocato. Ergo, **basta mettere i dati da manipolare nelle variabili locali e negli argomenti della funzione**, gestendo tutto con furbizia.

Questa è la tecnica di gran lunga più usata (se non quasi l'unica) per costruire un PDA sfruttando le chiamate a funzioni per implementare lo stack. Esistono strumenti che lo fanno in automatico: io specifico le grammatiche e poi spingo un tasto. Dalle regole poi vengono generati automaticamente i linguaggi con le relative chiamate a funzioni → Questi strumenti sono i **PARSER GENERATOR**.

Nell'Analisi Ricorsiva Discendente:

- Si introduce **una funzione per ogni meta-simbolo** della grammatica e **la si chiama ogni volta che si incontra quel meta-simbolo** (normalmente se si hanno i meta-simboli A, B, S e si creano delle funzioni come parseA, parseB, parseS)
- **Ogni funzione copre le regole di quel meta-simbolo,** ossia riconosce il sotto-linguaggio corrispondente
    - **Termina normalmente**, o restituisce un segno di successo, se incontra simboli coerenti con le proprie regole. (Le funzioni ausiliarie vedono solo pezzettini di stringhe, quindi non terminano quando è terminata la stringa, ma terminano quando incontrano dei simboli corretti. Solo la funzione di top level termina quando finisce la stringa.)
    - **Abortisce**, o restituisce un qualche segno di fallimento, se incontra simboli che non corrispondono alle proprie regole.

Esempio – Stringa di numeri palindroma
Il "solito" linguaggio L = { word c word^R} con Alfabeto A = { 0, 1, c } e regole: S → 0S0 |1S1 |c

1. Introduco tante funzioni quanti meta-simboli → qui c'è solo S, quindi una funzione sola S()
2. Chiamo una funzione ogni volta che ne incontro il meta-simbolo → ogni volta che trovo S, invoco S()
3. Ogni funzione deve coprire le regole di quel meta-simbolo → qui abbiamo 3 regole
    a. Se input = 0 → seguo la prima regola
    b. Se input = 1 → seguo la seconda regola
    c. Se input = c → seguo la terza regola

Prima regola S → 0S0 (voglio uno 0, poi qualsiasi cosa, poi di nuovo 0)

- Consumo 0
- Invoco ricorsivamente la funzione S()
- Consumo un nuovo carattere e verifico che sia 0
    - Se sì, simboli coerenti, quindi termina normalmente o restituisce successo
    - Se no, simboli incoerenti, quindi abortisce o restituisce fallimento

La seconda regola è uguale ma con 1.
Terza regola S → c (voglio il carattere c)

- Consumo c
- Non invoco altre funzioni
- Non consumo nuovi caratteri e non verifico niente. Se ho incontrato c, successo, se no no

Una possibile implementazione in C

```
char ch; // globale
bool S(){
 char first;
 bool result;
 switch (ch){
  case 'c':  ch = nextchar(); result = true; break;
  case '0':
  case '1':  first = ch;                                   /* push */
             ch = nextchar();
             if (S()) if(ch==first) {                      /* pop */
                        ch = nextchar(); result = true;
                      }
                      else result = false;
             else result = false;
  default:   break;
 }
 return result;
}
```

Le istanze di `first` nei record di attivazione rappresentano *de facto* lo stack del PDA.

SEPARARE MOTORE E GRAMMATICA

Applicare l'analisi ricorsiva discendente è un processo meccanico semplice, ma dà luogo a un insieme di funzioni che cablano nel codice il comportamento del PDA.

Può essere opportuno **SEPARARE il motore** (invariante rispetto alle regole) **dalle regole della specifica grammatica.** Si costruisce a questo scopo una **TABELLA DI PARSING** simile alla tabella delle transizioni di un RSF, ma che indica la prossima produzione da applicare.

Il motore del parser (Parsing Engine) svolgerà le singole azioni consultando la tabella di Parsing.

Esempio

Il solito linguaggio: L = { ***word*** c ***word***^{R} }

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

L = { if c then cmd (endif | else cmd) }
Produzioni:
S → if c then cmd X X → endif | else cmd

	if	c	then	endif	else	cmd
S	S → if c then cmd X	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	*error*	X → endif	X → else cmd	*error*

PRO	CONTRO
• È **immediato** scrivere il riconoscitore a partire dalla grammatica • **Il mapping fra struttura della grammatica e del riconoscitore riduce la probabilità di errori** e **migliora la leggibilità** e la **modificabilità del codice** • **È facilitata l'inserzione di azioni nella fase di analisi** (come la creazione di rappresentazioni interne del programma)	• L'analisi ricorsiva discendente **non è sempre applicabile** • L'approccio **funziona solo se non ci sono mai "dubbi" su quale regola applicare** in una qualsiasi situazione

Ciò suggerisce di identificare una classe ristretta di grammatiche context-free, che garantisca il determinismo dell'analisi sintattica discendente.

5.6 - GRAMMATICHE LL(k)

Per rendere deterministica l'analisi ricorsiva discendente bisogna mettersi nelle condizioni di poter dedurre la mossa giusta dalle informazioni "disponibili", senza dover mai tirare a indovinare.

Le informazioni disponibili sono le regole che abbiamo usato fin lì e i simboli di input che abbiamo letto e consumato fin lì. Spesso, però, la mera conoscenza del passato non basta: si ipotizza perciò di poter "vedere avanti" di k simboli (solitamente 1), ossia di poter "sbirciare" l'input ancora da leggere.

Si definiscono GRAMMATICHE LL(k) quelle che sono analizzabili in modo deterministico

- Procedendo Left to right (perché in occidente l'input va in questo modo)
- Applicando la Left-most derivation (derivazione canonica sinistra)
- Guardando avanti di al più k simboli

In sostanza, se una grammatica è LL(k), è sempre possibile scegliere con certezza la produzione da usare per procedere, guardando avanti al più di k simboli sull'input.

Rivestono particolare interesse le **GRAMMATICHE LL(1)**, quelle in cui basta guardare avanti di un solo simbolo per poter operare in modo deterministico. Con k=1, ho una tabella due dimensioni, facilmente gestibile.

Esempio - LL(1)

Si consideri la grammatica: VT = { p, q, a, b, d, x, y}, VN = { S(scopo), X, Y}

Produzioni: S → p X | q Y X → a X b | x Y → a Y d | y

Le parti destre delle produzioni di uno stesso meta-simbolo iniziano tutte con un simbolo terminale diverso. È quindi sufficiente guardare avanti di un carattere per scegliere con certezza la produzione con cui proseguire l'analisi. Se non esistono produzioni compatibili con quell'input, ERRORE.

Riconoscimento della frase "**p a a a x b b b**":

FRASE	PRODUZIONE	DERIVAZIONE
p a a a x b b b	S ::= pX	p**X**
a a a x b **b**	X ::= aXb	pa**X**b
a a x b **b**	X ::= aXb	paa**X**bb
a x **b**	X ::= aXb	paaa**X**bbb
x	X ::= x	paaa**x**bbb
ε	Nessuna	paaaxbbb

Esempio - PDA deterministico LL(1)

Per una codifica di principio in linguaggio C / C++ serve un contratto chiaro su chi legge l'input e quando lo fa. IPOTESI: ogni funzione trova nella variabile globale "ch" il prossimo carattere da analizzare, già letto ma non ancora analizzato. Ergo, ogni funzione, prima di ritornare, effettua una lettura da input a beneficio di chi verrà dopo di lei. Il main effettua la prima lettura prima di invocare la funzione di top-level.

Tocca al main stabilire cosa fare quando la funzione di top-level ritorna: si pretende che ch==EOF/EOLN o va bene qualunque carattere?

```
char ch ;/* variabile globale*/
int main() {
        ch = nextchar(); /* recupera il prossimo carattere di ingresso */
        if ( S() ) printf("Frase accettata\n"); else printf("Errore\n");
}
bool X() {/* analogamente si scrive la funzione Y() per le altre produzioni */
        switch(ch) {
                case 'a': ch = nextchar(); /* produzione X ::= aXb*/
                        if( X() ) {
                                if(ch=='b') { ch = nextchar(); return true; }
                                else return false; /* non corrisponde: ERRORE */
                        } else return false; /* non corrisponde: ERRORE */
                case 'x': ch= nextchar(); return true; /* produzione X ::= x */
        } return false; /* nessuna produzione corrispondente: ERRORE */
}
bool S() {
        switch(ch) {
                case 'p': ch= nextchar(); returnX(); /* produzione S ::= pX*/
                case 'q': ch= nextchar(); returnY(); /* produzione S ::= qY*/
        } return false; /* nessuna produzione corrispondente: ERRORE */
}
```

Qui nessuno verifica cosa ci sia in ch al ritorno di S(), si accettano anche frasi con caratteri "spuri" in coda a una frase corretta.

Costruendo la tabella di parsing è facile vedere che il parser è deterministico, ossia, che la **grammatica è LL(1)** perché ogni cella contiene una sola produzione e quindi non c'è mai dubbio su quale sia la prossima mossa da fare.

	p	q	a	b	d	x	y
S	S → p X	S → q Y	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	X → a X b	*error*	*error*	X → x	*error*
Y	*error*	*error*	Y → a Y d	*error*	*error*	*error*	Y → y

Il ruolo del main al ritorno della funzione di top-level dipende dalla **frase in input**, che può essere:

- **COMPLETA** → Non ci deve essere nient'altro dopo, "ch" al ritorno di S() deve contenere EOF/EOLN/altro terminatore. Se uso questa opzione vuol dire che dopo devo verificare che la stringa sia finita, serve per vedere se c'è ancora qualcosa, per non buttare caratteri.

Di default, questo è quello che fanno i compilatori, perché se alla fine di una classe metto 100 spazi, me li prende lo stesso. Se però una volta finito tutto scrivo "aiefjhwe" oppure anche solo un altro ";", allora non va bene e va in errore. Quindi, il compilatore controlla che ci sia l'EOF o almeno che non ci sia nulla di rilevante.

- **SOLO UNA PARTE** → può esserci altro dopo, che però non ci interessa. "ch" al ritorno di S() può contenere qualunque cosa (probabilmente S() non vede davvero tutto il linguaggio…)

Esempio - PDA deterministico LL(1) con frase completa

Se è necessario imporre che la frase sia completa, la grammatica dev'essere completata con una produzione esplicita di top-level.

Si consideri la grammatica: VT = { p, q, a, b, d, x, y, **$**}, VN = { **Z**(scopo), S, X, Y}

Produzioni: **Z → S $** S → p X | q Y X → a X b | x Y → a Y d | y

(da notare che in questo caso è tutto in forma di Greibach, ma normalmente non è così)

Il simbolo $ è un segnaposto per il terminatore scelto che può essere EOF, EOLN, ';' o altro. Sotto questa ipotesi, dopo la frase non deve esserci nient'altro, es. "paaaxbbb$" è corretta, mentre "paaaxbbb $" è errata.

5.6.1 - GENERALIZZAZIONE

Spesso però le parti destre delle produzioni di uno stesso meta-simbolo non iniziano tutte con un simbolo terminale, quindi, non è immediatamente chiaro quali siano gli input "ammissibili". Occorre ragionare considerando più produzioni fino a "svelare l'arcano".

Si consideri l'esempio a lato:

- Se la frase d'ingresso inizia con "p", va scelta la produzione S → RY, poiché solo la sua (sotto)produzione R→pXb può produrre una "p" iniziale.
- Se la frase di input inizia con "q", va scelta la produzione S → TZ, poiché solo la sua (sotto)produzione T→qYd può produrre una "q" iniziale.

Questo porta a generalizzare il concetto di "simbolo iniziale".

VT = { p, q, a, b, d, y }
VN = { S, X, Y, T, Z, R }
Produzioni:
S → R Y | T Z
R → p X b
T → q Y d
Z → ...
X → a X b | x
Y → a Y d | y

STARTER SYMBOLS SET

Lo **STARTER SYMBOLS SET** della riscrittura α è l'insieme **SS(α) = {a∈VT | $\alpha \overset{*}{\Rightarrow} a\beta$}** con $\alpha \in V^+, \beta \in V^*$.

In sostanza, **gli Starter Symbols sono le iniziali di una forma di frase α**, ricavate anche applicando più produzioni (si conviene che SS(ε)=∅). **L'operatore * sulla derivazione ⇒ cattura il caso limite in cui α ∈VT (è già un terminale)**, che non richiede di applicare derivazioni.

Questa definizione non cattura però situazioni del tipo $\alpha \to \varepsilon$, in quanto la loro derivazione non può generare frasi della forma a β. Per questo si introduce l'analogo **INSIEME FIRST**, che può includere ε:

$$\textbf{FIRST}(\alpha) = \textbf{trunc1}(\{ x \in VT^* \mid \alpha \overset{*}{\Rightarrow} x \}) \qquad \text{con } \alpha \in V^*$$

dove trunc1 denota il troncamento della stringa al primo elemento. Visto che x∈VT*, include ε.

In particolare, se α è costituita da un singolo meta-simbolo A:

$$\textbf{SS(A)} = \{a \in VT \mid A \overset{+}{\Rightarrow} a\beta\} \qquad \text{con } A \in VN, \beta \in V^*.$$

In questo caso, dato che A∈VN, almeno una derivazione è sicuramente necessaria, da cui l'operatore + sulla derivazione.

Ciò permette di generalizzare l'approccio precedente: se prima affermavamo che le parti destre delle produzioni relative a uno stesso meta-simbolo dovevano iniziare tutte con un simbolo terminale distinti, ora generalizziamo l'idea affermando che **le parti destre relative a produzioni di uno stesso meta-simbolo debbano essere caratterizzate da Starter Symbol Set distinti**.

La condizione diventa anche sufficiente se nessun meta-simbolo genera la stringa vuota.

CONDIZIONE NECESSARIA perché una grammatica sia LL(1) è che _gli starter symbols_ relativi alle parti destre di uno stesso metasimbolo siano _disgiunti_

Riassumendo, se l'insieme delle iniziali della prima alternativa è diverso da quelle nell'insieme delle iniziali della seconda alternativa, ok. Non devo avere un carattere che appartenga ad entrambi gli insiemi, altrimenti non ho più una macchina deterministica.

VT = { p, q, a, b, d, y }
VN = { S, X, Y, T, Z, R }
Produzioni:
S → R Y | T Z
R → p X b
T → q Y d
Z → ...
X → a X b | x
Y → a Y d | y

Esempio

Gli SSS delle parti destre delle due produzioni alternative di S sono:
SS(RY) = SS(R) = {p} SS(TZ) = SS(T) = {q}
Essendo disgiunti, la grammatica può essere LL(1) (condizione necessaria verificata).
Poiché nessuna produzione genera la stringa vuota, la condizione è anche sufficiente: dunque, la grammatica è effettivamente LL(1).
La stringa vuota fa differenza perché se una produzione genera la stringa vuota, quel meta-simbolo in realtà sparisce quando viene sostituito in un'altra regola, ergo, regole che sembrano iniziare con un certo meta-simbolo in realtà iniziano col successivo e questo va messo in conto.

Esempio 2 con e senza ε-rules

Produzioni (caso senza ε-rules):
S → A B | B
A → a A | d
B → b B | c

SS(A) = SS(aA) U SS(d) **= {a, d}** dove SS(aA) = {a}, SS(d) = {d}
SS(B)= {b, c} dove SS(bB) = {b}, SS(c) = {c}
Calcoliamo gli SS relativi alle due possibili produzioni alternative di S:
SS(AB) = SS(A) = {a, d} SS(B) = SS(B) = {b, c}
Dato che nessun metasimbolo genera la stringa vuota, gli SS di ogni riscrittura coincidono con quelli del relativo metasimbolo iniziale. Inoltre sono disgiunti, quindi la condizione necessaria è verificata.
Poiché nessuna produzione genera la stringa vuota, la condizione è anche sufficiente: la grammatica è LL(1).

Produzioni (caso con ε-rules):
S → A B | B
A → a A | ε
B → b B | c

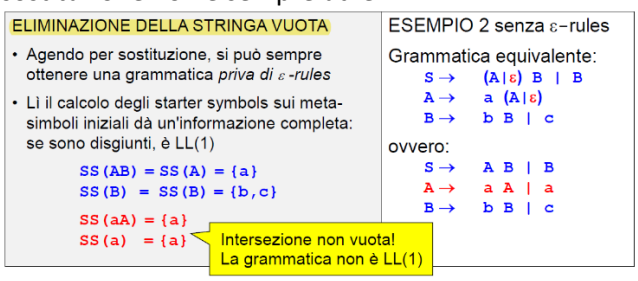
SS(B) = {b,c} poiché SS(bB) = {b} e SS(c) = {c}.
SS(A) = SS(aA) U SS(***ε***) = **{a}** dove SS(aA) = {a}, SS(***ε***) = ∅
Stavolta gli SS di ogni riscrittura non coincidono più con quelli del relativo metasimbolo iniziale, infatti SS(AB) ≠ SS(A) perché se A si annulla, l'iniziale è decisa da B. Allora: SS(AB) = SS(A) U SS(B) = {a,b,c} SS(B) = SS(B) = {b,c}
Sebbene SS(A) e SS(B) siano disgiunti, SS(AB) e SS(B) non lo sono: la condizione LL(1) non è verificata.

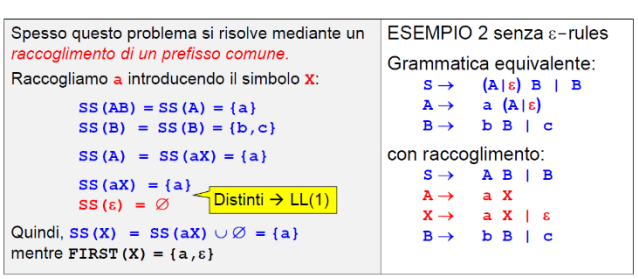
Per risolvere abbiamo due possibilità:

- **Si elimina la stringa vuota agendo per sostituzione**
- **Si amplia in qualche modo la nozione di Starter Symbol Set**

ELIMINARE STRINGA VUOTA PER SOSTITUZIONE

La sostituzione concettualmente si può sempre fare, ma spesso si usano le epsilon per catturare parti del linguaggio che sono opzionali. Se pensiamo ad un linguaggio vero ci sono talmente tante combinazioni di parti opzionali, che sostituire la stringa vuota con tutti i possibili casi diventerebbe ingestibile, quindi la sostituzione non è sempre utile.





Sostituendo la parentesi (A| ε) con X, pospongo la decisione.
Quindi, intanto mi mangio la a, poi quando farò nextchar() vedrò il carattere ancora dopo, il che significa che intanto adesso ho fatto una scelta e poi la stringa vuota la gestirò dopo .

Esempio 3

Produzioni:

S → A B
A → P Q | B C
P → p P | ε
Q → q Q | ε
B → b B | d
C → c C | f

In questo esempio il problema risiede nelle due produzioni di P e Q, che possono annullarsi. Ciò accade nonostante gli SS non lo evidenzino esplicitamente, anzi: SS(P)={p} SS(Q)={q}.
Di conseguenza, anche SS(PQ) dà una "falsa sensazione": sembra distinto SS(BC):
SS(PQ) = SS(P) U SS(Q) = {p,q} SS(BC) = SS(B) = {b,d}
In realtà l'intero blocco PQ è annullabile, il che si ripercuote sul metasimbolo A "falsando" anche lo SSS di S. Come nell'esempio precedente, SS(AB) = SS(A) U SS(B), ma qui SS(A) = SS(PQ) U SS(B), quindi SS(B) è già compreso: SS(AB) = SS(A).
Gli SS relativi a produzioni alternative sono distinti, MA la condizione LL(1) non è verificata.

In questi casi, la descrizione con l'insieme **FIRST**, che include la stringa vuota (se presente), può risultare più chiara, mettendo sull'avviso:

Con gli starter symbols:	SS(P)={p}	SS(Q)={q}
	SS(PQ)=SS(P) U SS(Q) ={p, q}	SS(BC)=SS(B) ={b, d}
Con gli insiemi FIRST:	FIRST(P) = {p, ε}	FIRST(Q) = {q, ε}
	FIRST(PQ)= {p, q, ε}	FIRST(BC)= SS(B) = {b, d}

Agendo per sostituzione, la grammatica assume una nuova forma. Ripetendo i calcoli per le produzioni di A:

S → A B | B
A → P Q | P | Q | B C
P → p P | p
Q → q Q | q
B → b B | d
C → c C | f

SS(P)={**p**} SS(Q)={q} SS(PQ) = SS(P) = {**p**} SS(BC) = SS(B) = {b,d}
Perciò SS(A) = SS(P) U SS(Q) U SS(B) con SS(P) e SS(pq) in conflitto
Gli SS relativi ai meta-simboli A e B, che compaiono all'inizio delle parti destre della regola di top-level S → AB|B, risultano: SS(A) = {p,q,b,d} SS(B) = {b,d}
Essi non sono disgiunti, confermando che la grammatica non è LL(1).

Sulle Parsing Table:

- **Se la grammatica è LL(1)**, ogni cella contiene al più una sola regola
- **Se essa non è LL(1)**, in almeno una cella ci saranno invece più regole

Abbiamo detto che **avevamo due possibilità, ma in realtà ne abbiamo 3:**

- **Si elimina la stringa vuota** agendo **per sostituzione**
- **Si amplia la nozione di Starter Symbol Set** estendendo la nozione di SSS verso la completa nozione di **DIRECTOR SYMBOLS set** (o Look-Ahead set)
- **Si agisce sulla Parsing Table** formalizzando il concetto di **BLOCCO ANNULLABILE** e integrando nella tabella l'informazione sulle stringhe che possono scomparire

	p	q	b	c	d	f
S	S → A B	S → A B	S → A B S → B	error	S → A B S → B	error
A	A → P Q A → P	A → Q	A → B C	error	A → B C	error
B	error	error	B → b B	error	B → d	error
C	error	error	error	C → c C	error	C → f
P	P → p P P → p	error	error	error	error	error
Q	error	Q → q Q Q → q	error	error	error	error

PARSING TABLE CON BLOCCHI ANNULLABILI

L'idea è quella di costruire la Parsing Table tenendo conto a priori delle ε-rules utilizzando un **BLOCCO ANNULLABILE**, cioè **una stringa che può degenerare in** ε.
In presenza di blocchi annullabili, un meta-simbolo che non sembrava iniziale può trovarsi "di fatto" a inizio frase. Quindi, bisogna considerare anche l'**insieme FOLLOW**, cioè **i simboli che possono seguire quelli annullabili.**
Sulla Parsing Table un blocco annullabile deve essere previsto anche in corrispondenza di quei terminali che possono seguire il blocco annullabile, perché potrebbero averlo davanti "senza saperlo".

Esempio 3

Riprendendo l'esempio precedente, il blocco PQ è annullabile perché sia P sia Q possono degenerare in ε. La regola A → PQ va quindi prevista in tutte le colonne della riga A i cui simboli possono seguire A (cioè b, d) perché potrebbero avere un "PQ invisibile" davanti.

ESEMPIO 3 con ε -rules

S → A B
A → P Q | B C
P → p P | ε
Q → q Q | ε
B → b B | d
C → c C | f

	p	q	b	c	d	f
S	S → A B	S → A B	S → A B	*error*	S → A B	*error*
A	A → P Q	A → P Q	A → B C A → P Q	*error*	A → B C A → P Q	*error*
B	*error*	*error*	B → b B	*error*	B → d	*error*
C	*error*	*error*	*error*	C → c C	*error*	C → f
P	P → p P	P → ε	P → ε	*error*	P → ε	*error*
Q	*error*	Q → q Q	Q → ε	*error*	Q → ε	*error*

DIRECTOR SYMBOLS SET

L'idea è ampliare la nozione di Starter Symbols, definendo un nuovo insieme che integri a priori l'effetto delle ε-rules. Quando una produzione genera la stringa vuota, lo Starter Symbols set dà un'informazione incompleta (potenzialmente fuorviante), quindi, come detto prima, bisogna considerare anche l'**insieme FOLLOW.**

Si può creare allora un nuovo insieme caratterizzante detto **DIRECTOR SYMBOLS SET** (o Look-Ahead Set) **identico allo Starter Symbols Set quando non c'è la stringa vuota, ma integrato dal nuovo insieme FOLLOW quando qualche produzione genera la stringa vuota** (e quindi blocchi annullabili).

Definiamo **DIRECTOR SYMBOLS SET** della produzione A→α l'unione dei due insiemi:

$$DS(A \to \alpha) = SS(\alpha) \cup \textbf{FOLLOW(A)} \qquad \text{se } \alpha \overset{*}{\Rightarrow} \varepsilon$$

dove FOLLOW(A) denota l'insieme dei simboli che possono seguire la frase generata da A:

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

Durante la derivazione possono apparire fra "A" e "a" dei simboli non-terminali, che però scompaiono successivamente trasformandosi in ε.

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

Si può anche definire **DS(A → α) = trunc1(FIRST(α) · FOLLOW(A))**, cioè quel che si ottiene prendendo l'iniziale delle frasi ottenute concatenando (·) FIRST con ciò che segue A.

Se FIRST non contiene ε, le iniziali sono tutti e soli i simboli di SS, quindi DS(A→ α) e SS coincidono; quando però FIRST contiene ε, DS(A→α) include anche ciò che segue A, completando la descrizione con l'informazione mancante.

Si può allora riformulare la condizione LL(1):

CONDIZIONE NECESSARIA E SUFFICIENTE **perché una grammatica sia LL(1) è che i Director Symbols set relativi a produzioni alternative siano *disgiunti.***

A questo punto, è inutile che io sviluppi tutte le regole con le stringhe nulle ritrovandomene molte di più di quelle che avevo all'inizio, basta che io mi calcoli il DSS.

In molti testi, soprattutto di origine anglosassone, **tali insiemi sono chiamati LOOK-AHEAD SET**, perché servono per "guardare avanti" guidando deterministicamente il parser.

Esempio 3

ESEMPIO 3 con ε -rules

S → A B
A → P Q | B C
P → p P | ε
Q → q Q | ε
B → b B | d
C → c C | f

SS: SS(P)={p} SS(Q)={q}
SS(PQ)=SS(P) U SS(Q) = {p, q} SS(BC)=SS(B) ={b, d}
FIRST: FIRST(P) = {p, ε} FIRST(Q) = {q, ε}
FIRST(PQ)= {p, q, ε} FIRST(BC)= SS(B) = {b, d}

Director Symbols Set DSS: DS(A → PQ) = SS(PQ) U FOLLOW(A) = {p, q, b, d} perché PQ genera ε
DS(A → BC) = SS(BC) = {b, d} perché BC non genera ε
in quanto i simboli che possono seguire la frase generata da A, che compare solo in S → AB, sono tutti e soli quelli prodotti da B.

Calcolo alternativo con l'altra definizione:
DS(A → PQ) = trunc1(FIRST(PQ) · FOLLOW(A)) = {p,q, ε } · {b,d} = trunc1({pb,qb,b,pd,qd,d}) = {p,q,b,d}
DS(A → BC) = trunc1(FIRST(BC) · FOLLOW(A)) = {b,d} · {b,d} = trunc1({bb,db,bd,dd}) = {b,d}

5.6.2 – PROBLEMA della RICORSIONE SINISTRA

La ricorsione sinistra non va d'accordo con LL(1) perché **le produzioni ricorsive a sinistra, della forma A::=Aα|a, danno sempre luogo a SSS identici per le due alternative.**

In teoria questo non è un problema perché la ricorsione sinistra si può sempre eliminare, **ma in pratica lo è** perché, facendo operazioni per eliminare la ricorsione, la grammatica cambia diventando ricorsiva a destra e con essa cambia l'albero di derivazione.

Questo a livello di **linguaggio riconosciuto** non importa (le frasi lecite restano le stesse), ma se si sfrutta l'albero per inserire azioni semantiche, allora è importante perché un **albero diverso implica una semantica diversa**!

Se la grammatica non è LL(1):

- Si può cercare di riorganizzarla con sostituzioni, raccoglimenti, …
- Si può cercare di modificare qualche regola "fastidiosa"
- Si può passare al caso LL(k), con k>1
    - Le definizioni di FIRST, FOLLOW, etc. si estendono facilmente al caso LL(k) con k>1

Tuttavia, tutto ciò potrebbe non bastare: non tutti i linguaggi context-free possiedono una grammatica LL(k): esistono linguaggi deterministici che non sono LL(k) per nessun k.
In generale non si può sapere se un linguaggio ammette una grammatica LL(1): **stabilire se un linguaggio sia LL(1) è un problema indecidibile.** Al contrario **stabilire se una grammatica sia LL(1) è un problema decidibile** perché basta usare i Director Symbols.

Esempio - Raccoglimento
Un raccoglimento a fattor comune non consente di rendere sempre la grammatica LL(1), ma in svariati casi pratici funziona. La tecnica consiste nell'**isolare il prefisso più lungo comune a due produzioni**.
Es. S → a S b | a S c S → ε non è LL(1), ma se isolo "aS" comune e introduco X…
S → a S X | ε X → b | c ora è LL(1)
La tecnica non funziona nei casi in cui il raccoglimento porta a un ciclo di riscritture senza fine.
Tanto per cominciare, questo linguaggio non è regolare perché ha simboli un po' a caso.

Se calcolo i DS, già vedo che c'è qualcosa che non va perché c'è "d" in conflitto.
Nella prima variante abbiamo una "d" davanti e quindi è già meglio. Ora proviamo a fare quello che abbiamo fatto nell'esempio precedente.

- La grammatica a lato non è LL(1), poiché DS(A → Bb) = { d,f } e DS(A → Cc) = { d,h }
- Apparentemente non c'è prefisso comune, tuttavia si può provare a sostituire B e C

ESEMPIO 5 (base)

A → B b | C c
B → d B e | f
C → d C g | h

- La prima regola (che conteneva due alternative) si suddivide perciò in quattro alternative
- Nelle prime si può tentare il raccoglimento:

ESEMPIO 5 (variante a)

A → d B e b | d C g c
A → f b | h c

- Sfortunatamente, per E sorge lo stesso problema che c'era inizialmente per A
- Tentare di raccogliere ancora non risolve nulla: il problema si ripropone ulteriormente.

ESEMPIO 5 (variante b)

A → d E | f b | h c
E → B e b | C g c

Ora ho ottenuto un E che non è poi così bello. Se continuo a fare queste elaborazioni, scopro che non solo rimane brutto uguale, ma peggiora addirittura. Quindi si deve accendere la lampadina: se non sono arrivato ad un risultato, questo è un loop, quindi il raccoglimento è inutile.

Riassunto:

- Le grammatiche LL(k) consentono l'analisi deterministica delle **frasi Left to right**, con **Left-most derivation** e usando k simboli di Look-ahead.
- Non tutti i linguaggi context-free possiedono una grammatica LL(k)

Esistono però tecniche più potenti dell'analisi LL: le grammatiche LR(k) consentono l'analisi deterministica delle **frasi Left to right**, con **Right-most derivation**, usando k simboli di Look-ahead.
L'analisi LR è meno naturale dell'analisi LL ma è superiore dal punto di vista teorico perché arriva dove l'LL non arriva. Infatti, vi sono linguaggi context-free deterministici non analizzabili in modo deterministico con tecniche LL, ma riconoscibili in modo deterministico con tecniche LR.

Esperimenti con Parsing Emulator (slide 69-90 pack 05) → software **Easy Free Parsing Emu**

6 – DAI RICONOSCITORI AGLI INTERPRETI

Finora abbiamo considerato puri riconoscitori in cui si può sempre sostituire una grammatica con una equivalente, perché l'effetto finale è identico.
Un **interprete** è più di un puro riconoscitore:

- **Riconosce se una stringa appartiene al linguaggio**
- **Esegue azioni in base al significato (semantica) della frase**
  - Può svolgerle direttamente → valutazione immediata
  - Può costruire strutture dati per permettere di svolgerle in un secondo tempo

Conseguenze: la sequenza di derivazione diventa importante perché contribuisce a definire il significato della frase, quindi non si può sempre sostituire una grammatica con una equivalente perché l'equivalenza è tale solo "ai morsetti"!

Un interprete è di solito organizzato in un'**architettura client/server** ed è strutturato su **due componenti**:

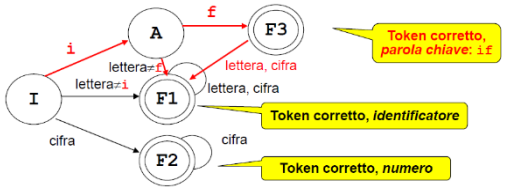
- **SCANNER**: l'analizzatore lessicale.
  Analizza le parti regolari del linguaggio, fornendo al parser singole parole (token) già aggregate, evitandogli di doversi occupare dei dettagli relativi ai singoli caratteri
- **PARSER**: l'analizzatore sintattico-semantico.
  Riceve dallo scanner i singoli token, considerandoli elementi terminali del suo linguaggio per valutare la correttezza della loro sequenza: opera sulle parti context-free del linguaggio

6.1 – ANALISI LESSICALE

L'analisi lessicale consiste nella individuazione delle singole parole (token) che compongono una frase. Ciò viene fatto raggruppando i singoli caratteri dell'input secondo le produzioni regolari associate alle diverse possibili categorie lessicali (identificatore, numero, etc.) e utilizzando più stati finali.
L'analizzatore lessicale (SCANNER) può categorizzare i token mentre li analizza semplicemente osservando in quale stato finale del suo RSF si viene a trovare.

Tuttavia, **cablare ogni dettaglio nella struttura del RSF non è una strategia vincente** poiché un linguaggio comprende parole chiave e simboli speciali. Per tenerne conto, un RSF dovrebbe avere regole ultra specifiche capaci di "schivarle" una ad una e si otterrebbe una struttura RSF complicatissima.

Esempio per includere "if" nel linguaggio. Se dovessi fare così per ogni parola chiave diventerei idiota e avrei tantissimi stati.



6.1.1 - TABELLE

Per evitare di cristallizzare nella struttura del RSF le parole chiave, i simboli speciali, etc. conviene agire diversamente:

- Categorizzare le parole chiave come identificatori → grammatica e struttura RSF molto semplici
- Ri-categorizzarle correttamente consultando opportune **tabelle** che incapsulano la conoscenza di dettaglio del linguaggio:
  - **Tabella delle parole-chiave del linguaggio**
  - **Tabella dei simboli speciali del linguaggio**

In questo modo, si sposta la decisione da un piano puramente sintattico a uno sintattico-semantico ottenendo uno **scanner modulare, estendibile, snello.**

Esempio Java
In Java le funzioni dell'analisi lessicale si possono ottenere sfruttando opportunamente le classi fornite:

- **java.io.StreamTokenizer + java.util.StringTokenizer**: basati su idea di definire i separatori possibili. La funzione nextToken restituisce il prossimo token
- **java.util.Scanner + java.util.Regex**: approccio più sofisticato e flessibile in cui i separatori sono descrivibile mediante espressioni regolari
- **metodo String.split()**: si possono esprimere con espressioni regolari anche altri parti del linguaggio ottenendo un array di token già suddivisi

6.2 – ANALISI SINTATTICA TOP-DOWN

In presenza di grammatiche LL(1), l'analisi top-down ricorsiva discendente offre una tecnica semplice e diretta per costruire il riconoscitore.
Negli esempi visti, ogni funzione restituiva un boolean (puri riconoscitori), quindi **per passare ad un interprete** occorre **propagare** qualcosa di più:

- Un **valore**, se l'obiettivo è la **valutazione immediata** (interprete) in un qualche dominio
- Un **albero**, se l'obiettivo è la **valutazione differita** (compilatore o interprete a più fasi). Qui la vera valutazione nel dominio di interesse avviene più avanti

Esempio – espressioni aritmetiche
Si supponga di voler riconoscere espressioni aritmetiche con le quattro operazioni.
Un puro riconoscitore deve solo dire se sono corrette (ogni funzione restituisce un boolean), mentre un interprete deve anche dire quanto valgono:

- Se il **dominio** sono gli **interi (Z)**, il risultato può essere un **valore int**
- Se il **dominio** sono i **reali (R)**, il risultato può essere un **valore double**
- Se l'obiettivo è invece la **valutazione differita**, il risultato può essere un opportuno oggetto adatto a rappresentare un albero

SINTASSI → Tipicamente si usa una **notazione infissa** basata sui quattro operatori "+ - x :" sostituiti dagli informatici con "+ - * /". Ad essa si accompagnano spesso le **parentesi** per esprimere priorità e associatività "non standard".
SEMANTICA → Nel dominio aritmetico usuale:

- Valori numerici si assumono espressi in notazione posizionale su base dieci ("15" = quindici)
- Significato inteso dei quattro operatori è quello di somma, sottrazione, moltiplicazione, divisione
- Si introducono le nozioni di priorità e associatività:
    - Priorità fra operatori diversi → gli operatori moltiplicativi prioritari su quelli additivi
    - Associatività fra operatori equi-prioritari → solitamente si associa a sinistra

Consideriamo il linguaggio E(G) relativo alla seguente grammatica per espressioni aritmetiche. Supponiamo che "num" sia riconosciuto da un interprete di tipo 3. **È una grammatica ambigua: espando EXP di dx o di sx?**

Semantica informale:

- Ogni Exp è un'espressione
- **Se Exp è "num"**, l'espressione denota un intero e il valore dell'espressione coincide con quello del numero.

- **Se invece Exp è "EXP op EXP"** l'espressione denota il valore ottenuto applicando l'operatore ai valori denotati dalle due espressioni e1, e2. Si conviene che gli operatori "*" e ":" abbiano priorità maggiore rispetto agli operatori "+" e "-". Operatori equi-prioritari sono valutati da sinistra a destra.

```
VN = { EXP }
VT = { +, *, -,  :, num }
S = EXP
P = {
  EXP ::= EXP + EXP     // plusexp
  EXP ::= EXP - EXP     // minusexp
  EXP ::= EXP * EXP     // timesexp
  EXP ::= EXP : EXP     // divexp
  EXP ::= num           // numexp
}
```

num denota la notazione in base 10 di un numero intero senza segno, che si suppone nota (cfr. tipo 3)

Frasi lecite: 5 + 3 2 + 3 * 4 5 – 3 - 1
Frasi illecite: 5 + 3 + (2 + 3) * 4 -5

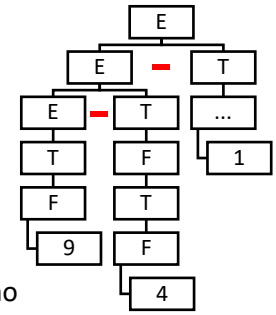
Le descrizioni ambigue vanno evitate il più possibile. Sarebbe meglio cablare nella grammatica gli elementi e le scelte cruciali. Si può dare una **struttura gerarchica alle espressioni**, ottenendo una "**GRAMMATICA A STRATI**" ed esprimendo così intrinsecamente priorità e associatività degli operatori.

Una soluzione come questa è preferibile perché faccio prima la fatica che se no avrei dovuto far fare al parser dopo. Da notare che:

```
VN = { EXP, TERM, FACTOR }
VT = { +, *, -,  :, (, ), num }
P = {
EXP    ::= TERM
EXP    ::= EXP + TERM
EXP    ::= EXP - TERM
TERM   ::= FACTOR
TERM   ::= TERM * FACTOR
TERM   ::= TERM : FACTOR
FACTOR ::= num
FACTOR ::= ( EXP )
}
```

1	Priorità MIN
2	Priorità MED
3	Priorità MAX

- **Il livello 1 è regolare**
- **Il livello 3 è LL1**
- **I livelli 1 e 2 hanno una ricorsione sinistra**: a livello di puro riconoscitore sarebbe uguale fare una ricorsione destra, ma se faccio la ricorsione dx o sx creo un albero diverso, quindi per le operazioni devo usare quella sinistra



Ogni strato considera terminali gli elementi linguistici definiti in altri strati:

- **EXP** considera terminali +, - e TERM
- **TERM** considera terminali *, : e FACTOR
- **FACTOR** considera terminali num, (,) e EXP

Esempio di frase: 9-4-1 → Prima sottraggo 9-4, poi sottraggo 1.
Se avessimo la ricorsione destra, avremmo un albero specchiato e otterremmo un'espressione del tipo 9-(4-1).

Ogni strato definisce quindi un suo sotto-linguaggio che usa quei "terminali":

- L(EXP) = TERM ± TERM ± TERM …
- L(TERM) = FACTOR */: FACTOR */: FACTOR …
- L(FACTOR) = num | (EXP)

Il sotto-linguaggio di num (regolare) è indipendente: potrebbe adottare una sintassi binaria, romana…

La grammatica non è più ambigua perché ogni strato può produrre solo certi operatori:

- Somme e sottrazioni sono di competenza del livello 1
- Moltiplicazioni e divisioni sono di competenza del livello 2
- Singoli valori e sotto-espressioni competono al livello 3

Gli strati superiori aggregano entità prodotte in strati inferiori, inoltre la stratificazione induce priorità fra gli operatori, perché le entità di strati bassi devono essere sintetizzate per prime.
Entro ogni strato, la ricorsione (se presente) stabilisce come si aggregano entità di pari livello:

- Ricorsione SINISTRA = associatività operatori A SINISTRA (livello 1 e 2)
- Ricorsione DESTRA = associatività operatori A DESTRA
- Nessuna ricorsione = operatori NON ASSOCIATIVI (livello 3)

La grammatica è quindi ricorsiva a sinistra per inglobare nella sua struttura l'associatività desiderata per motivi culturali. **PECCATO che la ricorsione sinistra sia incompatibile con l'analisi ricorsiva discendente**, principale tecnica di costruzione di parser!
Se si riscrivessero le regole in forma ricorsiva a destra, l'ordine di priorità non cambierebbe, ma l'associatività fra operatori equi-prioritari sarebbe ora a destra, dando luogo a un'aritmetica alquanto inusuale, quindi la semantica basata sull'albero di derivazione risulterebbe diversa. ☹

```
VN = { EXP, TERM, FACTOR }
VT = { +, *, -,  :, (, ), num }

P = {
  EXP    ::= TERM
  EXP    ::= TERM + TERM
  EXP    ::= TERM - TERM
  TERM   ::= FACTOR
  TERM   ::= FACTOR * FACTOR
  TERM   ::= FACTOR : FACTOR
  FACTOR ::= num
  FACTOR ::= ( EXP )
}
```

Pur mantenendo lo stesso ordine di priorità fra gli operatori, si potrebbe anche fare del tutto a meno dell'associatività, ottenendo una **GRAMMATICA NON ASSOCIATIVA**. Queste regole non usano ricorsione né a sinistra né a destra: **risolvono il problema obbligando sempre a usare le parentesi**, anche quando tipicamente non vengono usate.
La nuova grammatica, non ricorsiva né a sinistra né a destra, è compatibile con l'analisi ricorsiva discendente. Ma si può davvero convincere un'intera cultura ad adottare una sintassi "verbosa" solo perché farebbe comodo a noi?

6.3 – RICORSIONE SINISTRA e ANALISI TOP-DOWN

Il riconoscitore è un PDA, perché la grammatica ha self-embedding nella regola EXP ::= … ::= FACTOR ::= (EXP)
PROBLEMA: la sintassi delle espressioni include produzioni ricorsive a sinistra, quindi così com'è, la grammatica non è LL(1) → l'analisi ricorsiva discendente non è applicabile.

OBIETTIVO: rendere la grammatica LL(1).

1. **Riconsideriamo i sotto-linguaggi generati dai diversi strati.** I primi due sono regolari e sono facilmente descrivibili da **espressioni regolari**:
   a. L(EXP) = TERM ± TERM ± TERM … → L(EXP) = TERM (± TERM)*
   b. L(TERM) = FACTOR */: FACTOR */: FACTOR … → L(TERM) = FACTOR (*/: FACTOR)*
   c. L(FACTOR) = num | (EXP)
2. **Ricordiamo che la notazione EBNF** offre un modo per esprimere la ripetizione senza far uso diretto di ricorsioni, tramite la notazione $\{\alpha\}$. Si possono allora mappare le espressioni regolari che descrivono i due sotto-linguaggi su regole EBNF
   a. L(EXP) = TERM (± TERM)* → EXP ::= TERM { (+ | -) TERM }
   b. L(TERM) = FACTOR (*/: FACTOR)* → TERM ::= FACTOR { (* | :) FACTOR }
   c. L(FACTOR) = num | (EXP)

   Ora non abbiamo più ricorsione esplicita e ho un processo computazionale iterativo, implementabile anche senza far uso di ricorsione. 😊
3. **RISULTATO**: la grammatica è analizzabile con la tecnica ricorsiva discendente senza rischiare l'esplosione dello stack, ergo è una **grammatica LL(1)**

Infatti, nel caso di EXP:

- All'inizio c'è sicuramente un TERM
- Poi o non c'è niente (ε o altro terminatore), o c'è uno dei due simboli + – seguito da un altro TERM
- SSS distinti → LL(1) (idem per TERM)

Si può esplicitare quanto asserito scrivendo:
EXP ::= TERM AFTERTERM AFTERTERM ::= ε | +EXP | -EXP
TERM ::= FACTOR AFTERFACTOR AFTERFACTOR ::= ε | *TERM | :TERM

Questa grammatica è LL(1), ma è anche ricorsiva a destra → non va dunque utilizzata "operativamente" per derivare la frase e costruire l'albero sintattico perché l'associatività degli operatori risulterebbe culturalmente "sbagliata", ma serve solo per verificare che la grammatica precedente sia LL(1).
Il riconoscitore sarà implementato seguendo la grammatica EBNF precedente, che esclude la ricorsione.

6.4 – DALLA GRAMMATICA AL PARSER

6.4.1 - ANALISI TOP-DOWN nella pratica

Analisi ricorsiva discendente (top down):

- Una **procedura o funzione per ogni simbolo non terminale**
- **Invocazione ricorsiva solo per il caso con self-embedding** (realizza il PDA riconoscitore per grammatica di tipo 2)

Ogni funzione restituisce:

- Un boolean, nel caso di puri riconoscitori
- Un opportuno valore/oggetto, nel caso di parser completi che effettuino anche una valutazione (o meta-valutazione)

Ogni funzione termina quando:

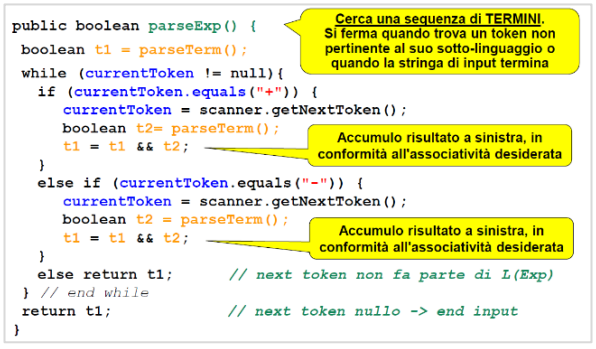
- La stringa di input finisce
- Incontra un simbolo non appartenente al sotto-linguaggio di sua pertinenza (approccio prudente)

Esempio – Schema di Parser

Ogni funzione analizza il sotto-linguaggio di pertinenza:

- **parseExp** analizza L(EXP), per il quale +, - e TERM sono l'alfabeto terminale
- **parseTerm** analizza L(TERM), per il quale *, : e FACTOR sono l'alfabeto terminale
- **parseFactor** analizza L(FACTOR), il cui alfabeto terminale è costituito da (,) ed EXP

Puro riconoscitore → ogni funzione restituisce un **boolean** e all'insegna dell'**approccio prudente**, eventuali simboli successivi nella stringa, non appartenenti al linguaggio, saranno ignorati, ma non scateneranno una segnalazione di errore.




```
public boolean parseExp() {
 boolean t1 = parseTerm();
 while (currentToken != null){
   if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      boolean t2= parseTerm();
      t1 = t1 && t2;
   }
   else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      boolean t2 = parseTerm();
      t1 = t1 && t2;
   }
   else return t1;       // next token non fa parte di L(Exp)
 } // end while
 return t1;              // next token nullo -> end input
}
```

parseExp() deve cercare un termine. Poi ho tre possibilità:

- È finita
- Ho un "+" e un termine
- Ho un "-" e un termine

Prima di tutto devo trovare il primo termine, se non c'è quello, ciccia. Se non ho un + o un -, quindi ho qualcosa che non conosco, allora ritorno quello che ho letto fino ad ora (else return t1 prima della end while).

6.4.2 – ARCHITETTURA

Una possibile scelta:

```
public boolean parseTerm(){
 boolean f1 = parseFactor();
 while (currentToken != null){
   if (currentToken.equals("*")) {
      currentToken = scanner.getNextToken();
      boolean f2 = parseFactor();
      f1 = f1 && f2;
   }
   else if (currentToken.equals(":")) {
      currentToken = scanner.getNextToken();
      boolean f2 = parseFactor();
      f1 = f1 && f2;
   }
   else return f1;       // next token non fa parte di L(Term)
 } // end while
 return f1;              // next token nullo -> end input
}
```

Cerca una sequenza di FATTORI. Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

Accumulo risultato a sinistra, in conformità all'associatività desiderata

Accumulo risultato a sinistra, in conformità all'associatività desiderata

```
public boolean parseFactor() {
  if (currentToken.equals("(")) {
    currentToken = scanner.getNextToken();
    boolean innerExp = parseExp();        // self-embedding
    if (currentToken.equals(")")) {
         currentToken = scanner.getNextToken();
        return innerExp;        // parentesi irrilevanti
    } else return false;        // manca la parentesi chiusa
  }
  else                          // dev'essere un numero
  if (currentToken.isNumber()) {
    currentToken = scanner.getNextToken();
    return true;
  }
  else              // non è un fattore, quindi
    return false;   // non è qualcosa di riconosciuto
}
```

Cerca un SINGOLO FATTORE
Se trova un token non pertinente *restituisce null*

- **Componente software per il parser dinamico**, istanza di una classe MyParser, dove currentToken è una variabile privata di istanza
- **Scanner di supporto come componente dinamico**, istanza di una classe MyScanner. Per comodità, una classe derivata da StringTokenizer che richiede spazi fra i token ed è pure deprecata, ma dopo tutto, è un problema dell'utente, mica nostro
- **Rapporto architetturale fra parser e scanner**: il main crea sia lo scanner, sia il parser, poi il costruttore del parser riceve un riferimento allo scanner
- **Concretizzazione per l'astrazione TOKEN**: un oggetto progettato ad hoc, che incapsula una stringa

```
public class Token {
  private String tk;
  public Token(String tk){ this.tk = tk;}
  public boolean isNumber() {
    try{ Integer.parseInt(tk); }
    catch(NumberFormatException e){ return false; }
    return true;
  }
  public String toString(){ return tk; }
  public boolean equals(Object o){
        if (o instanceof String) {
          return this.tk.equals((String)o);
        }
        else if (o instanceof Token) {
         Token that = (Token)o;
         return this.tk.equals(that.tk);
        }
        else return false;
  }
}
```

```
class MyScanner extends java.util.StringTokenizer {

  public MyScanner(String txt){
        super(txt);
  }

  public Token getNextToken(){
    try{
        return new Token(nextToken().trim());
    }
    catch (java.util.NoSuchElementException e){
        return null;
    }
  }

}
```

```
public class TestRiconoscitore {
  public static void main(String args[]) {
     String expression = "( 3 - 1 ) * ( 4 - 5 )";
     MyScanner scanner = new MyScanner(expression);
     MyParser parser = new MyParser(scanner);
     boolean result = parser.parseExp();
     System.out.println("Expression is" +
       (result ? " correct" : " wrong"));
  }
}
```

```
---------- Java Run ----------
Expression is correct
```

Un cliente dovrà:

- Creare l'istanza di scanner (qui, specifica per una stringa)
- Creare l'istanza di parser che usa tale scanner
- Invocare parseExp per riconoscere la frase

6.5 – DAL PARSER AL VALUTATORE

Cosa sappiamo fare:

- **Dato il linguaggio desiderato, trovare una grammatica** adatta che lo descriva
- **Data la grammatica, scrivere il puro riconoscitore** per il corrispondente linguaggio

Cosa manca → Data la grammatica, scrivere il parser completo per il corrispondente linguaggio che effettui anche una valutazione (o meta-valutazione).

Cosa serve → Valutare significa attribuire significato alle frasi, quindi serve la specifica della **semantica** che il parser dovrà applicare.

6.5.1 – SPECIFICARE LA SEMANTICA

Occorre un modo sistematico e formale per stabilire con precisione e senza ambiguità il <u>significato</u> di ogni possibile frase del linguaggio:

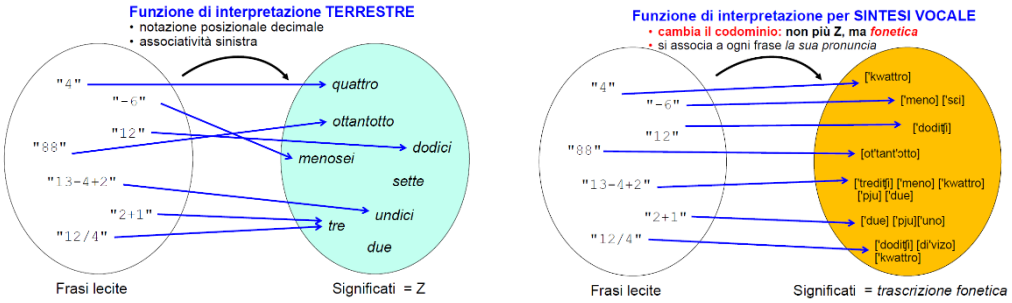
- **Se il linguaggio è finito**, basta un elenco
- **Se è infinito**, serve una notazione finita (applicabile a infinite frasi)

Un modo è definire una **<u>funzione di interpretazione</u>**:

- **DOMINIO**: il **linguaggio** (insieme delle frasi lecite, ossia stringhe)
- **CODOMINIO**: l'**insieme dei possibili significati**, ossia l'insieme degli oggetti che si vogliono far corrispondere a tali frasi (Es. per espressioni su interi, il codominio è Z)

La funzione di interpretazione è definita:

- Se il linguaggio è finito, da una tabella (stringhe → significati)
- Altrimenti, serve una funzione definita in modo ricorsivo



SEMANTICA DENOTAZIONALE

Quando la semantica di un linguaggio è espressa in questo modo si parla di **semantica denotazionale** e se il linguaggio è infinito, la semantica è definita tramite funzione ricorsiva.

Un buon **modo per specificare la semantica** è seguendo pari pari la sintassi:

- Per ogni regola sintattica, una regola semantica
- Non si rischiano dimenticanze, mapping pulito e chiaro da leggere
- Nel nostro caso la sintassi prevede Exp, Term e Factor: la semantica prevedrà tre funzioni fExp, fTerm e fFactor

```
VN = { EXP, TERM, FACTOR }
VT = { +, *, -,  :, (, ), num }
P = {
  EXP    ::= TERM
  EXP    ::= EXP + TERM
  EXP    ::= EXP – TERM
  TERM   ::= FACTOR
  TERM   ::= TERM * FACTOR
  TERM   ::= TERM : FACTOR
  FACTOR ::= num
  FACTOR ::= ( EXP )
}
```

```
fExpr (s)  = fTerm (s)
fExpr (s1 + s2)  =  fExpr (s1) + fTerm (s2)
fExpr (s1 – s2)  =  fExpr (exp) - fTerm (s2)
fTerm (s)  = fFactor (s)
fTerm (s1 * s2)  =  fTerm (s1) * fFactor (s2)
fTerm (s1 : s2)  =  fTerm (s1) / fFactor (s2)
fFactor ( (s) ) = fExpr (s)
fFactor (num) = valueof (num)
```

Si specifica un comportamento per ogni possibile "pattern" di frase.

Es. se guardo la regola 2 vuol dire che per trovare il significato di S1+S2, devo prima trovare il significato del termine S1 e devo aggiungerlo al significato del termine S2.

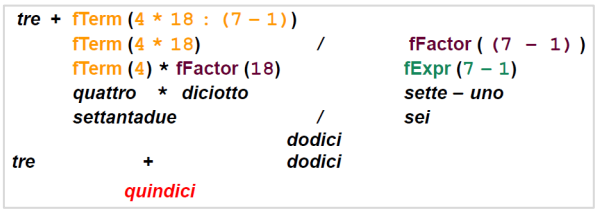
In **rosso** ho i simboli sintattici della grammatica, mentre in **nero** ho le operazioni "note" sul dominio N.

Espressione: "3 + 4 * 18 : (7 – 1)"

L'espressione ha la forma **Exp + Term**, cui corrisponde l'interpretazione: fExpr(s1 + s2) = fExpr(s1) + fTerm(s2)

ossia, nel caso in esame: fExpr (3 + 4 * 18 : (7 – 1)) = **fExpr (3)** + fTerm (4 * 18 : (7 – 1))

Poi fExpr (s) = fTerm (s) → **fExpr (3)** = fTerm (3) → fTerm (3) = fFactor (3) = valueof (3) = **tre**
E così via…



Esempio – Schema di Interprete con valutazione immediata
Ogni funzione analizza il sotto-linguaggio di pertinenza:

- **parseExp** analizza L(EXP), per il quale +, - e TERM sono l'alfabeto terminale
- **parseTerm** analizza L(TERM), per il quale *, : e FACTOR sono l'alfabeto terminale
- **parseFactor** analizza L(FACTOR), il cui alfabeto terminale è costituito da (,) ed EXP

Qui ogni funzione restituisce un int o un double e dipende dal dominio di interpretazione che si sceglie.

```
public int parseExp(){
 int t1 = parseTerm();
 while (currentToken != null){
   if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      int t2 = parseTerm();
      t1 = t1+t2;
   }
   else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      int t2 = parseTerm();
      t1 = t1-t2;
   }
   else return t1;       // next token non fa parte di L(Exp)
 } // end while
 return t1;              // next token nullo -> end input
}
```

Cerca una sequenza di TERMINI. Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

Accumulo risultato a sinistra, in conformità all'associatività desiderata

Accumulo risultato a sinistra, in conformità all'associatività desiderata

```
public int parseTerm(){
 int f1 = parseFactor();
 while (currentToken != null){
   if (currentToken.equals("*")) {
      currentToken = scanner.getNextToken();
      int f2 = parseFactor();
      f1 = f1*f2;
   }
   else if (currentToken.equals(":")) {
      currentToken = scanner.getNextToken();
      int f2 = parseFactor();
      f1 = f1/f2;
   }
   else return f1;       // next token non fa parte di L(Term)
 } // end while
 return f1;              // next token nullo -> end input
}
```

Cerca una sequenza di FATTORI. Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

Accumulo risultato a sinistra, in conformità all'associatività desiderata

Accumulo risultato a sinistra, in conformità all'associatività desiderata

```
public int parseFactor() {
 if (currentToken.equals("(")) {
   currentToken = scanner.getNextToken();
   int innerExp = parseExp(); // self-embedding
   if (currentToken.equals(")")) {
        currentToken = scanner.getNextToken();
        return innerExp;      // parentesi irrilevanti
   } else throw new IllegalArgumentException("missing )");
 }
 else                         // dev'essere un numero
 if (currentToken.isNumber()) {
   int value = currentToken.getAsInt();
   currentToken = scanner.getNextToken();
   return value;
 }
 else throw new IllegalArgumentException("unrecognised");
}
```

Cerca un SINGOLO FATTORE Se trova un token non pertinente *restituisce null*

```
public class TestInterprete {
 public static void main(String args[]) {
    String expression = "( 3 + 4 ) * 5";  // 1st run
    String expression = "3 + 4 * 5";      // 2nd run
    MyScanner scanner = new MyScanner(expression);
    MyInterpreter parser = new MyInterpreter(scanner);
    int result = parser.parseExp();
    System.out.println(expression + " = " + result);
 }
}
```

```
( 3 + 4 ) * 5  = 35
3 + 4 * 5  = 23
```

6.5.2 - ARCHITETTURA

Un cliente dovrà:

- Creare l'istanza di scanner (qui, specifica per una stringa)
- Creare l'istanza di ~~**parser**~~ **interprete** che usa tale scanner
- Invocare parseExp per ~~**riconoscere**~~ **interpretare** la frase

Esempio – Aggiungere l'elevamento a potenza
Riassunto: abbiamo visto che per ragionare a livelli bisogna avere ben chiare le priorità.
Per la parte di valutazione, abbiamo scelto di esprimere la semantica con regole che stiano il più vicino possibile a quelle sintattiche. Questo ci dà il vantaggio di traslare quella che era l'implementazione del riconoscitore SI/NO in un riconoscitore che fa anche da valutatore, cioè un interprete.
Per le potenze, mentre quando scriviamo a mano è semplice vedere la priorità delle potenze, quando lo scriviamo in sequenza al calcolatore è più difficile capire al volo la priorità. Avendo l'associazione a destra, basta usare una grammatica destra, quindi si riesce bene ad implementare.

- Nuovo operatore ^ → $3*4^2$ si scrive 3*4^2

- **Priorità**: in matematica l'elevamento a potenza è più prioritario di moltiplicazioni e divisioni ($3*4^2$ si legge 3x16, non 12^2) → 3*4^2 si dovrà interpretare come 3*(4^2)
- **Associatività**: in matematica la notazione ad apice rende evidente cosa si vuol fare, ma con l'operatore in linea occorre precisarlo.

Es. 4^2^3 si deve leggere come $4^{2^3} = 16^3 = 4^8$ o come $4^{2x3} = (4^2)^3 = 4^6$?
Poiché la seconda è facilmente esprimibile con le proprietà delle potenze, la scelta classica è considerare 4^2^3 come 4^8 ossia considerare l'**operatore ^ associativo a destra.**

Serve allora un nuovo livello intermedio nella nostra grammatica.

```
EXP    ::= TERM                    1
EXP    ::= EXP + TERM
EXP    ::= EXP - TERM
TERM   ::= POT                     2
TERM   ::= TERM * POT
TERM   ::= TERM : POT
POT    ::= FACTOR                  3
POT    ::= FACTOR ^ POT   (associativo a destra)
FACTOR ::= num                     4
FACTOR ::= ( EXP )
```

```
public int parseTerm(){
 int f1 = parsePot();          // Identica, ma cerca sequenza di potenze anziché di fattori
 while (currentToken != null){
   if (currentToken.equals("*")) {
      currentToken = scanner.getNextToken();
      int f2 = parsePot();
      f1 = f1*f2;
   }
   else if (currentToken.equals(":")) {
      currentToken = scanner.getNextToken();
      int f2 = parsePot();
      f1 = f1/f2;
   }
   else return f1;       // next token non fa parte di L(Term)
 } // end while
 return f1;              // next token nullo -> end input
}
```

```
public int parsePot() {
int f1 = parseFactor();
 if(currentToken != null && !currentToken.equals("")){
 if (currentToken.equals("^")) {
   currentToken = scanner.getNextToken();
   int p2 = parsePot();          // L'operatore associativo a destra consente l'uso diretto di una funzione ricorsiva a destra
   int res;
   for (res=1;p2>0;p2--){        // Implementa la semantica dell'elevamento a potenza fra interi non negativi (e gli altri..?)
        res = res*f1;
   }
   f1 = res;                     // occhio, se il risultato eccede l'int...
 }
 else return f1;
 } // end while
 return f1;          // next token nullo -> end input
}
```

```
public class TestInterprete {
 public static void main(String args[]) {
    String expr = "3 ^ 2 ^ ( 2 + 1 )";      // 1st run
    String expr = "( 3 ^ 2 ) ^ ( 2 + 1 )"; // 2nd run
    String expr = "3 ^ 2 ^ 2 + 1";          // 3rd run
    MyScanner scanner = new MyScanner(expr);
    MyInterpreter parser = new MyInterpreter(scanner);
    int result = parser.parseExp();
    System.out.println(expr + " = " + result);
 }
}
```

```
3 ^ 2 ^ ( 2 + 1 ) = 6561
( 3 ^ 2 ) ^ ( 2 + 1 ) = 729
3 ^ 2 ^ 2 + 1 = 82
```

Nel blocco rosso, siccome l'operazione non esiste, quindi va implementata ex novo. Ma se il risultato dovesse eccedere dal tipo **int** che ho utilizzato?
Invocazione da parte del cliente con l'architettura di test.

Riprendendo quanto detto nel capitolo 6.5, **cosa sappiamo fare:**

- **Dato il linguaggio desiderato, trovare una grammatica** adatta che lo descriva
- **Data la grammatica, scrivere il puro riconoscitore** per il corrispondente linguaggio
- **Data la grammatica, scrivere il parser completo** con **valutazione immediata** per il corrispondente linguaggio che effettui anche una valutazione (o meta-valutazione)

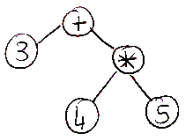
Cosa manca: Data la grammatica, scrivere il parser completo che effettui una **meta-valutazione** → generazione dell'**albero sintattico.**

6.5.3 – ALBERI SINTATTICI ASTRATTI (AST)

La GRAMMATICA descrive la struttura effettiva delle frasi, ossia la **sintassi concreta** del linguaggio. Tale grammatica è studiata in modo che il linguaggio risulti non solo ben definito, ma anche chiaro per chi legge. A tal fine la sintassi concreta include spesso elementi (punteggiatura, parole chiave, ...) non strettamente necessari, ma utili per migliorare la chiarezza e la leggibilità delle frasi.
Se la valutazione non è immediata, il parser rappresenta le frasi sintatticamente corrette tramite una opportuna rappresentazione interna, solitamente ad albero.

Si potrebbe usare l'ALBERO DI DERIVAZIONE, ma in generale esso darebbe luogo a una **rappresentazione ridondante** poiché illustra tutti i singoli passi di derivazione, ma molti di essi servono solo durante la costruzione dell'albero, per ottenere "proprio quell'albero" e non un altro. Inoltre, un albero con molti nodi e livelli è complesso da visitare (la visita è ricorsiva), determinando quindi **inutili inefficienze**.
Conviene adottare un albero più **compatto**, che contenga **solo i nodi indispensabili** e possibilmente sia pure **binario**. Tale albero è detto ABSTRACT PARSE TREE (APT) O ABSTRACT SYNTAX TREE (AST).

Esempio



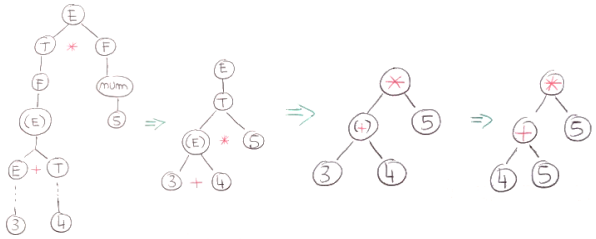
L'albero enorme che avevamo diminuisce e diventa:
Adesso è più semplice portarsi dietro questo alberino, che è la versione "zippata" dell'altro.
Quindi quello di prima con tutte le chiamate del parser è l'albero concreto, questo invece è l'albero astratto.
E poi non è del tutto finita: un albero che non è binario non è bello perché non si sa quanti figli ha, oltretutto si hanno un E e una T, che sono residui dell'albero precedente. Si potrebbero manipolare di nuovo, come se E avesse l'attributo +, trasformando la E in +, stessa cosa per T. Ottengo allora:
Questo sì che è un albero binario zippato e bellissimo: è l'AST.

Nodi non indispensabili:

- I **nodi terminali (foglie)** non legati ad alcunché di significativo sul piano semantico
  - **Segni di punteggiatura**, zucchero sintattico in genere
  - Nel caso delle espressioni non ne abbiamo
- I **nodi terminali (foglie)** che, pur utili durante la costruzione dell'albero per ottenere "quell'albero e non un altro", esauriscono con ciò la loro funzione
  - Nel caso delle espressioni: **le parentesi**
- **I nodi non terminali che hanno un unico nodo figlio**
  - Nel caso delle espressioni: sequenze Exp → Term → Factor

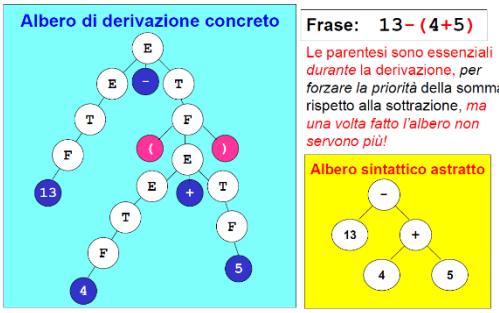
Si può **rendere l'albero binario** inserendo informazioni come proprietà del nodo-padre anziché come sotto-nodo figlio

- **Nelle espressioni**: spostando "in su" i simboli degli operatori
- **Analogo XML**: rappresentare una informazione come proprietà di un nodo anziché come sotto-elemento



Nel caso delle espressioni,
l'AST è così definito:

- Ogni operatore è un nodo con due figli
  - Il figlio sinistro è il primo operando
  - Il figlio destro è il secondo operando
- I valori numerici sono le foglie



Questi alberi sono **"alberi come valori"** perché sono unici, essi esprimono esattamente l'espressione a cui sono legati. Non avrebbe senso ordinarli, essi sono alberi immodificabili perché ad espressione data, questi alberi sono così e basta.

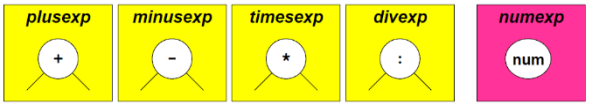
Così, la rappresentazione è univoca: una espressione è rappresentabile in un solo modo e **la struttura dell'albero fornisce intrinsecamente l'ordine corretto di valutazione.** È impossibile valutare un nodo senza disporre prima dei due figli, quindi occorre PER FORZA valutare prima la parte "in basso". Poi si risale fino a valutare la radice, che fornisce il risultato.

6.5.4 – SINTASSI ASTRATTA

OBIETTIVO: estendere il parser facendogli generare un opportuno AST

- **Analisi ricorsiva discendente (top down)**
- Ogni funzione restituisce:
  - NEL 1° CASO: un boolean (puro riconoscitore)
  - NEL 2° CASO: un intero (valutazione immediata)
  - ORA: un opportuno valore/oggetto che descriva un AST

Per far generare al parser un opportuno AST occorre prima stabilire come dev'essere fatto, ossia quali e quanti tipi diversi di nodo possa avere. Serve un nodo diverso per ogni possibile tipo di espressione, ma diverso in che senso? Proprietà diversa, un tipo diverso… Abbiamo 4 operazioni + costanti numeriche, quindi 5 tipi di nodo. Una possibile scelta potrebbe essere quella in figura, ma non è una descrizione formale.



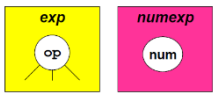
Una **SINTASSI ASTRATTA** ha precisamente l'**obiettivo di descrivere come è fatto l'AST:**

- Non è destinata all'utente: descrive una **rappresentazione interna**
- **Ogni produzione indica un possibile nodo**, ossia un possibile modo di costruire quel "tipo" di espressione

```
EXP ::= EXP + EXP   // plusexp
EXP ::= EXP - EXP   // minusexp
EXP ::= EXP * EXP   // timesexp
EXP ::= EXP : EXP   // divexp
EXP ::= num         // numexp
```

La sintassi astratta non è unica: poiché descrive una possibile rappresentazione interna, varia al variare di quest'ultima. **Sintassi astratte diverse descrivono sistemi software diversi.**

```
EXP ::= EXP OP EXP
EXP ::= num
OP ::= + | - | * | :
```



Se scegliamo una certa architettura, il sistema software che viene fuori ha determinate proprietà. In questo modo un attributo che aveva 4 versioni, può averne 5,6,7... Mentre prima dovevamo aggiungere una classe per ogni operatore. Però se usiamo un "enum", in realtà una classe nascosta c'è. E se usassimo le stringhe? Sarebbe scomodo perché poi dobbiamo usare la equals() e la compare(). Prima però, avendo un operatore per classe, si usava il polimorfismo dinamico ed era super efficiente. Invece ora si perde tempo e non è efficiente il sistema software.

LEGGE GENERALE DELL'UNIVERSO: se fai un'architettura più complessa perché hai disaggregato i problemi, poi è più semplice scrivere il codice. Se fai un'architettura molto semplicistica, poi devi impazzire a scrivere il codice.

Possibile implementazione Java con UML a slide 82-86 del pacchetto 6. Viene mostrato come l'UML finale rappresenti l'albero binario disegnato prima, dove ogni classe è un operatore. Ovviamente la costruzione dell'AST deve tenere conto della priorità degli operatori.

Esempio – Parser che genera un AST

```
public Exp parseFactor() {
  if (currentToken.equals("(")) {
     currentToken = scanner.getNextToken();
     Exp innerExp = parseExp();     // PDA: gestione self-embedding
     if (currentToken.equals(")")) {
          currentToken = scanner.getNextToken();
          return innerExp;          // le parentesi vengono omesse
     } else return null;
  }
  else                // dev'essere un numero
  if (currentToken.isNumber()) {
     int value = currentToken.getAsInt();
     currentToken = scanner.getNextToken();
     return new NumExp(value);
  }
  else                // errore: non è un fattore
     return null;     // non si è costruito nulla, restituiamo null
}
```

Cerca un SINGOLO FATTORE
Se trova un token non pertinente
restituisce null

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

```
public Exp parseExp() {
 Exp termSeq = parseTerm();
 while (currentToken != null){
   if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      Exp nextTerm = parseTerm(); -> (Prima c'era t1+t2 con associatività sx)
      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;          // next token non fa parte di L(Exp)
 } // end while
 return termSeq;   // next token è nullo -> stringa di input finita
}
```

Cerca una sequenza di TERMINI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

```
public class TestInterpreteAST {
  public static void main(String args[]) {
     String expression = "( 3 - 1 ) * ( 4 - 5 )";
     MyScanner scanner = new MyScanner(expression);
     MyParser parser = new MyParser(scanner);
     Exp ast = parser.parseExp();
     System.out.println(ast);      // da valutare poi
  }
}
```

```
public Exp parseTerm() {
 Exp factorSeq = parseFactor();
 while (currentToken != null){
   if (currentToken.equals("*")) {
      currentToken = scanner.getNextToken();
      Exp nextFactor = parseFactor();
      if (nextFactor != null)      // costruzione APT a sinistra
         factorSeq = new TimesExp(factorSeq, nextFactor);
      else return null;
   } else
   if (currentToken.equals(":")) {
      currentToken = scanner.getNextToken();
      Exp nextFactor = parseFactor();
      if (nextFactor != null)      // costruzione APT a sinistra
         factorSeq = new DivExp(factorSeq, nextFactor);
      else return null;
   }
   else return factorSeq;        // next token non fa parte di L(Term)
 } // end while
 return factorSeq;  // next token è nullo -> stringa di input finita
}
```

Cerca una sequenza di FATTORI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

Un cliente dovrà:

- Creare l'istanza di scanner
- Creare l'istanza di **parser** ~~**interprete**~~ che usa tale scanner
- Invocare parseExp per **riconoscere** ~~**interpretare**~~ la frase, **ottenendo l'AST**

In definitiva, dunque, **l'AST non rispecchia più, strutturalmente, la sintassi concreta.** **Riflette invece una sintassi astratta invisibile all'utente e all'esterno**, utile solo a specificare il formato interno usato dall'interprete. **Conseguenza: la sintassi astratta può essere ambigua**. Infatti, non serve a guidare il riconoscitore, ma solo a descrivere come è fatto l'AST.

6.5.5 – VALUTARE GLI ALBERI

Assodato che l'albero sintattico astratto sia il modo più compatto per rappresentare un'espressione, come lo valutiamo? La teoria degli alberi introduce il concetto di VISITA:

- **VISITA IN PRE-ORDER**: radice, figli (da sinistra a destra) → **NOTAZIONE PREFISSA**
  - **Produce**: operatore, 1° operando, 2° operando
- **VISITA IN POST-ORDER**: figli (da sinistra a destra), radice → **NOTAZIONE POSTFISSA**
  - **Produce:** 1° operando, 2° operando, operatore
- **VISITA IN IN-ORDER**: figlio sinistro, radice, figlio destro (saltella fra i livelli) → **NOTAZIONE INFISSA**
  - **Produce:** 1° operando, operatore, 2° operando

L'uso della notazione infissa è un aspetto culturale: ci siamo talmente abituati da pensare che sia l'unica "possibile", ma in realtà è solo uno dei modi per rappresentare espressioni e neanche il più felice.
In effetti, è proprio l'infelice scelta di scrivere l'operatore in mezzo agli operandi che costringe a introdurre priorità, associatività e parentesi, **è la notazione infissa stessa a creare ambiguità nell'ordine di esecuzione delle operazioni!**

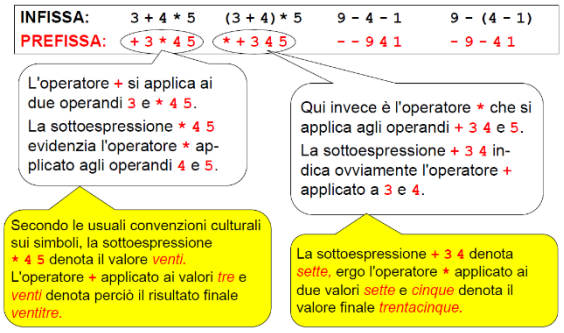
Adottando, ad esempio, una classica notazione funzionale, il problema non si pone: 9-4-1 diventa f(f(9,4),1), dove parentesi e virgole qui sono puramente estetiche! Potrei scrivere f f 9 4 1, cioè - - 9 4 1.

ATTENZIONE ALLA NOTAZIONE INFISSA!

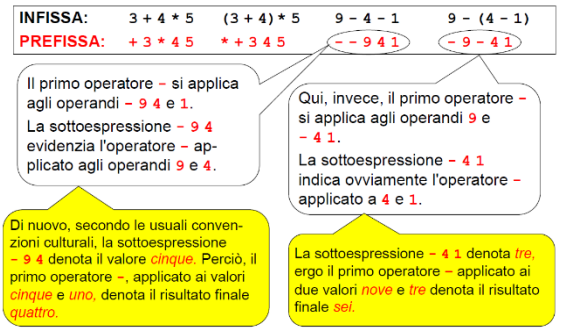
La visita IN-ORDER dà luogo alla notazione infissa, ma non evidenziando il "livello" (tramite parentesi o altri artifici grafici), l'algoritmo può dar luogo a frasi che, secondo le usuali convenzioni, useremmo per un'espressione diversa! (Es. 13-4-5 dovrebbe essere 13-(4-5))

NOTAZIONE PREFISSA

Nella **NOTAZIONE PREFISSA** ho prima l'operatore, poi gli operandi (è la tipica notazione funzionale).
Non richiede di definire alcuna nozione di priorità, associatività e parentesi perché elimina a priori ogni ambiguità sull'ordine di esecuzione delle operazioni. **La visita PRE-ORDER dà luogo alla notazione PREFISSA.**



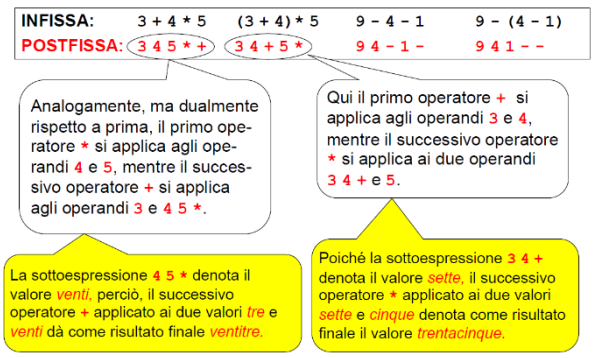
PRIORITÀ



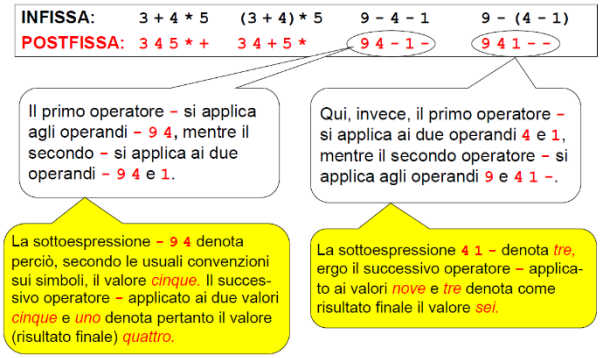
ASSOCIATIVITÀ

NOTAZIONE POSTFISSA

Nella **NOTAZIONE POSTFISSA** ho prima gli operandi, poi l'operatore. Non richiede di definire alcuna nozione di priorità, associatività e parentesi perché elimina a priori ogni ambiguità sull'ordine di esecuzione delle operazioni. **La visita POST-ORDER dà luogo alla notazione POSTFISSA**



PRIORITÀ



ASSOCIATIVITÀ

<u>La notazione postfissa è adattissima a un elaboratore</u> perché fornisce prima gli operandi che possono così essere caricati nei registri del processore o in altre zone opportune di memoria, pronti per l'ALU e solo dopo "comanda" l'esecuzione dell'operazione.

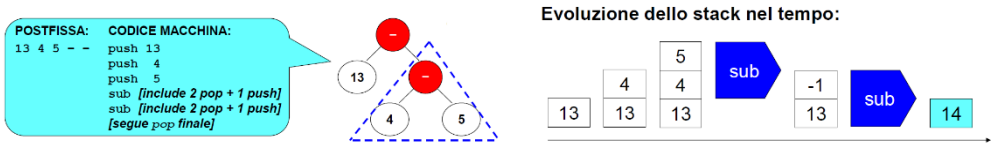
<u>Lo stesso principio è utilizzato anche nei compilatori:</u>

- Ogni nodo-operatore viene tradotto nell'operazione assembler
- Ogni nodo-valore viene tradotto nel caricamento di tale valore in un registro macchina

Se non vogliamo preoccuparci dei registri, si può anche usare una macchina a stack!

- **Ogni nodo-valore carica un valore** sullo stack (PUSH)

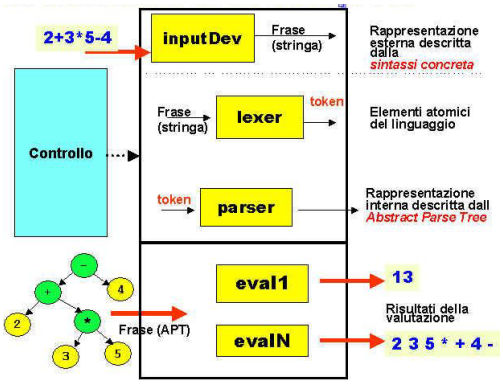
- **Ogni nodo-operatore causa il prelievo di due valori** dallo stack (POP) **e il collocamento** sullo stack **del risultato** (PUSH)
- **Alla fine si preleva il risultato** dallo stack (POP)



Es. Java Virtual Machine non si basa su un'architettura fisica, ma su una macchina virtuale. Però è fatto così, a stack. Nella linea del tempo viene mostrato come lavora la ALU.

Esperimenti con Parsing Emulator slide 116-134 pacchetto 06.

6.6 – VERSO IL VALUTATORE



6.6.1 – ARCHITETTURA DI UN INTERPRETE

Nell'architettura di un interprete **ci possono essere N valutatori** che leggono l'albero e mi danno risultati diversi, in base a quello che deve fare il mio sistema.
In questo esempio, abbiamo due valutatori in cui **ognuno valuta un AST in un dato dominio, secondo la sua funzione di interpretazione.**
Un valutatore che dà come risultato un linguaggio macchina, lo chiameremmo **COMPILATORE**.

6.6.2 – VALUTATORE

Il valutatore incorpora la funzione di valutazione:

- Deve visitare l'albero applicando in ogni nodo la semantica prevista per quel tipo di nodo, cioè deve discriminare che tipo di nodo sta visitando
- Scegliere strategia di visita
- Scegliere struttura software

Per implementare la **funzione di valutazione**, si può usare:

- METODOLOGIA FUNZIONALE: una **funzione tradizionale statica** → public static int eval (Exp e) con dentro catene orrende if…else.
  - Separa nettamente sintassi astratta e interpretazione
  - Si basa su una funzione di interpretazione esterna alle classi
- METODOLOGIA OBJECT ORIENTED: un **metodo eval** dell'**AST** → metto public abstract eval() dentro la classe Exp → interessante, potrei chiedere ad ogni nodo di valutarsi, poi io devo solo mettere insieme le cose. Svantaggio: f di valutazione è sparsa nell'AST.
  - Sfrutta **polimorfismo** dei linguaggi a oggetti → splittando la funzione in N pezzi, uno per ogni tipo di nodo, riesco a discriminarli. In questo modo definisco la funzione public int eval() tante volte ma con comportamenti diversi. **Pratico, efficiente e molto OOP** (Object-Oriented Programming)
  - Si basa su un metodo di interpretazione interno alle classi

	PRO	CONTRO
METODOLOGIA FUNZIONALE	Facilita introduzione nuove interpretazioni (basta scrivere nuova funzione)	Rende più oneroso introdurre una nuova produzione (modificare codice di tutte le f di interpretazione)
METODOLOGIA OBJECT ORIENTED	Facilita l'aggiunta di nuove produzioni (definisco nuova sottoclasse)	Rende più oneroso introdurre nuove interpretazioni (definisco nuovo metodo in ogni classe)

6.6.3 - COMPILATORE

Mentre l'interprete restituisce una valutazione della frase, il compilatore restituisce una meta-valutazione della frase, ossia una sua riscrittura in un codice intermedio. È quindi molto semplice passare da un interprete ad un compilatore.

Esempio – Arithmetic Stack Machine

- Senza registri, opera sullo stack
- Ha quattro operazioni aritmetiche: SUM, SUB, MUL, DIV
- Ha due operazioni di trasferimento da stack: PUSH, POP
- Per definizione, le operazioni aritmetiche prelevano dallo stack i due operandi (come se facessero due pop) e pongono sullo stack il risultato (come se facessero una push).

Tutte le classi della tassonomia vengono dotate della nuova operazione **emit(PrintWriter p)** che emette il codice macchina corrispondente alla meta-valutazione dell'espressione corrente sul dispositivo di uscita p.
Il **PrintWriter** è un'astrazione, poi dentro ha tutte le emissioni che ci pare. L'ordine con cui emetto il codice in un PrintWriter conta, non è una semplice stampa: prima emetto il figlio sinistro, poi il figlio destro, poi emetto la radice.
L'operazione di somma ha il corrispondente in assembler, quindi siamo a posto, ma se avessi un'operazione come l'elevamento a potenza che non ha un equivalente in assembler, bisognerebbe emettere una sfilza di codice macchina che di fatto causa una serie di moltiplicazioni secondo la semantica che abbiamo stabilito.

```
class PlusExp extends OpExp {
        ...
        public void emit(PrintWriter p) {
                left.emit(p); right.emit(p); p.println("SUM"); }
}
```

```
class NumExp extends Exp {
        …
        public void emit(PrintWriter p) {
                p.println("PUSH "+ val); }
}
```

6.6.4 – VISITOR

Di solito un linguaggio di programmazione ha una grammatica fissa, ma richiede molteplici interpretazioni delle frasi (analisi semantica, type checking, code generation, etc), perciò, la metodologia più opportuna sembra quella funzionale, che però non è object-oriented!
Bisognerebbe poter coniugare i plus dei due approcci:

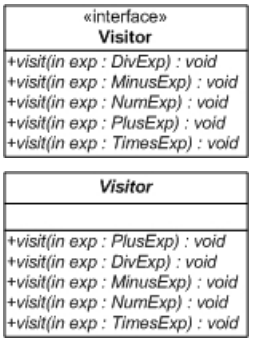
- **L'unitarietà concettuale e fisica della funzione di interpretazione**
- **La possibilità di porre in ogni classe la "parte di funzione di interpretazione" che le compete**, tipica dell'approccio a oggetti.

Occorre incapsulare la logica d'interpretazione in una entità di più alto livello che:

- **Sostituisca il costrutto funzione come "contenitore"** capace di assicurare l'unitarietà concettuale e fisica della funzione di interpretazione
- **Incorpori nei suoi metodi la business logic** della (ex) funzione di interpretazione
- **Supporti appieno il polimorfismo**, in modo da superare l'inguardabile pletora di instanceof della funzione "vecchia maniera"

Il **VISITOR** realizza la logica di interpretazione in modo coerente all'approccio a oggetti:

- **Cattura UNA logica di interpretazione**: si scrivono tanti visitor quante le interpretazioni richieste ed è possibile organizzarle in una **TASSONOMIA**
- **La logica di interpretazione è concentrata in UN unico luogo**, ma non ha più la forma di una "old style function": è suddivisa, all'interno del visitor, in **tante implementazioni del metodo visit** quante sono le classi della tassonomia (**overloading**)



Prevediamo allora un'**interfaccia visitor** che ha al suo interno il **metodo visit** in cui vengono separati i casi: gli argomenti sono DivExp, MinusExp, ecc. e sono gli stessi pezzi che c'erano nella grande funzione statica a suon di if if if if. Qui, invece, possono coesistere uno di fianco all'altro con il nome del metodo identico, ma l'argomento diverso, quindi utilizzando l'overloading.
Nel caso delle espressioni, potremmo avere una classe base astratta (o interfaccia) Visitor, poi un visitor che stampa le espressioni (ParExpVisitor), un visitor che calcola le espressioni (EvalVisitor) e così via.

```
Visitor visitor = new MyVisitor(); // visitor di valutazione
Result result = expression.accept(visitor);
```

Nell'approccio funzionale bisognava chiamare la funzione statica. Grado di oggettivizzazione zero.
Nell'approccio Object-Oriented inverto il punto di vista, ho previs to un metodo polimorfo con cui chiedo all'espressione di valutarsi. Non va tanto bene se penso che il mondo possa evolversi.
Nell'approccio a Visitor costruisco il mio visitor (che potrebbe essere quello che valuta, quello che stampa, quello che vuoi) e chiedo all'espressione di accettare la visita di quel visitor. Non devo più prevedere il metodo eval, emit e in futuro altri metodi (latec, pdf, ...): prevedo un unico metodo che non cambia mai che si chiama "accept". Sarà poi l'intruso che ti visita in base a quel che sta cercando. Quindi, **ciò che era argomento diventa target**: si passa da un'entità passiva ad un'entità attiva perché posso aggiungere visitor in qualunque momento (creando una nuova classe visitor) e l'impatto sull'esistente è zero perché non devo riaprire nessun altro file, né ricompilare tutto.
Il Visitor usa la **tecnica del DOUBLE DISPATCH**: si attiva la valutazione chiedendo all'espressione di accettare la visita di un certo visitor con expression.accept(visitor). L'espressione serve le richiesta rimpallando l'azione sul visitor e passando se stessa come oggetto da visitare, ossia invocando visitor.visit(this).
In questo "BOTTA & RISPOSTA" la risoluzione dell'overloading seleziona automaticamente la visit appropriata senza bisogno di instanceof, ottenendo quindi un **CAST DINAMICO.**

Esempi di Visitor nelle slide 164-170 pacchetto 06.

7 – VALUTAZIONE PROTOTIPALE DI ESPRESSIONI in JAVAFX e SWING

7.1 – JAVAFX

7.1.1 – TREEITEM

TreeItem è una classe JavaFX che modella alberi: pur essendo pensata per lavorare in tandem con il componente grafico TreeView, è in realtà indipendente dalla grafica. Questo è terribile, ma utile per esperimenti vari.

- **TreeItem<T> è un nodo che contiene un valore di tipo T** (è un wrapper sostanzialmente)
- L'albero si costruisce nodo per nodo e a tal fine, ogni nodo mantiene al suo interno una lista di figli.
- Il **metodo getChildren** consente di recuperare tale lista
- Col **metodo add**, vanno aggiunti gli altri nodi **alla lista**

Esempio

1. **Costruzione nodo radice**

   TreeItem<String> root = new TreeItem<>("Esseri viventi");

2. **Costruzione e aggancio nodi di primo livello**

   TreeItem<String> animali = new TreeItem<>("Animali");
   TreeItem<String> vegetali = new TreeItem<>("Vegetali");
   root.**getChildren().add**(animali);
   root.**getChildren().add**(vegetali);

3. **Aggancio nodi foglie (di secondo livello)**

   animali.**getChildren().add**(new TreeItem<>("Pesci"));
   animali.**getChildren().add**(new TreeItem<>("Mammiferi"));
   vegetali.**getChildren().add**(new TreeItem<>("Graminacee"));
   vegetali.**getChildren().add**(new TreeItem<>("Betullacee"));

Ipotesi: accumuliamo il risultato in uno `StringBuilder`

```
private static <T> void preorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
    if (root != null) {
       sb.append(root.getValue());
       List<TreeItem<T>> children = root.getChildren();
       if (children != null) {
         for (TreeItem<T> child : children) {
              sb.append(separator);
              preorder(child, sb, separator);
         }
       }
   }
```

Invocazione lato cliente:

```
StringBuilder sb = new StringBuilder();
preorder(root, sb, ", ");
System.out.println("preorder: " + sb);
```

[Visita in PRE-ORDER](): essendo solo TreeNode non hanno il concetto di visita. Quindi dobbiamo implementarlo noi. Si usa uno stringBuilder per mantenere i risultati da portarsi sempre dietro, quindi si usa una funzione statica che "preorder" che usa lo stringBuilder. Man mano che elaboro i nodi, cosa ci metto tra l'uno e l'altro? Usiamo un generale separator, e poi ci si mette in mezzo quello vuole l'utente.

Albero visto in top down: Esseri viventi, Animali, Pesci, Mammiferi, Vegetali, Graminacee, Betullacee

[Visita in POST-ORDER](): prima i figli e poi la radice.

Ipotesi: accumuliamo il risultato in uno `StringBuilder`

```
private static <T> void postorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
    if (root != null) {
       List<TreeItem<T>> children = root.getChildren();
       if (children != null) {
         for (TreeItem<T> child : children) {
              postorder(child, sb, separator);
              sb.append(separator);
         }
       }
       sb.append(root.getValue());
   }
}
```

```
postorder: Pesci, Mammiferi, Animali, Graminacee, Betullacee, Vegetali, Esseri viventi
```

[Visita in ORDER]()

Ipotesi: accumuliamo il risultato in uno `StringBuilder`

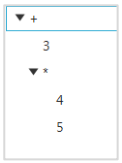
```
private static <T> void inorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
  if (root != null) {
   List<TreeItem<T>> children = root.getChildren();
   if (children != null && children.size()>2)
       throw new IllegalArgumentException("Not binary!");
   if (children != null && children.size()>0) {
       inorder(children.get(0), sb, separator);
   }
   sb.append(root.getValue()+ separator);
   if (children != null && children.size()>1) {
       inorder(children.get(1), sb, separator);
   }
  }
}
```

```
inorder: Pesci, Animali, Mammiferi, Esseri viventi, Graminacee, Vegetali, Betullacee,
```

Esempio - Espressione con TreeItem

Un albero per l'espressione 3+4*5 diventa:

```
TreeItem<String> root = new TreeItem<>("+");
TreeItem<String> l = new TreeItem<>("3");
TreeItem<String> r = new TreeItem<>("*");
TreeItem<String> rl = new TreeItem<>("4");
TreeItem<String> rr = new TreeItem<>("5");
r.getChildren().add(rl); r.getChildren().add(rr);
root.getChildren().add(l); root.getChildren().add(r);
TreeView<String> treeview = new TreeView<>(root);
```




```
preorder: +, 3, *, 4, 5
postorder: 3, 4, 5, *, +
inorder: 3, +, 4, *, 5,
```

La visita in-order non esprime il livello (asso-ciatività) degli operatori.

7.1.2 – VALUTAZIONE DELL'ALBERO

Elementi che ci servono:

- **La visita in post-order** → ce l'abbiamo, ma produce una stringa
- Occorre **adattarla per riempire una lista** anziché uno StringBuilder → nuova funzione **postorderEnumeration(root,list)** che riceve una List<TreeNode<String>> pronta da riempire
- **Uno stack di appoggio** → c'è nella JCF: **Stack<Integer> stack = new Stack<Integer>();**

Per valutare l'espressione basta quindi scandire la lista:

- **Quando si incontra un numero, lo si mette sullo stack** → push
- **Quando si incontra un operatore**, si svolge l'operazione:
  - **Prendendo i due dati dallo stack** → due pop
  - **Mettendo sullo stack il risultato** → push

Esempio

```
private static Integer calc(TreeItem<String> root) {
  Stack<Integer> stack = new Stack<Integer>();
  List<TreeItem<String>> list = new ArrayList<>();
  postorderEnumeration(root, list);
  for (TreeItem<String> item : list) {
    String value = item.getValue();
    try {
      Integer i = Integer.parseInt(value);
      stack.push(i);
    }
    catch (NumberFormatException e)  {
      // tutto ok, se non è un intero, value è l'operatore
      // qui va inserita la logica dell'operazione
    }
  }
  return stack.pop();
}
```

Non si può sapere se è un numero o un operatore
Ergo, si tenta e si vede come va a finire (orrendo!)

La logica dell'operazione:

```
catch (NumberFormatException e)  {
// tutto ok, se non è un intero, value è l'operatore
  Integer v2 = stack.pop();
  Integer v1 = stack.pop();
  switch (value) {
     case "+": stack.push(v1 + v2); break;
     case "-": stack.push(v1 - v2); break;
     case "*": stack.push(v1 * v2); break;
     case ":": stack.push(v1 / v2); break;
  }
}
```

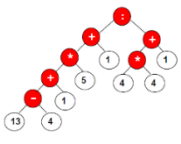
```
Il risultato di  3 4 5 * +  è  23
```

INFISSA: (((13-4)+1)*5)+1):(4*4+1)

POSTFISSA: 13 4 - 1 + 5 * 1 + 4 4 * 1 + :

Evoluzione dello stack:

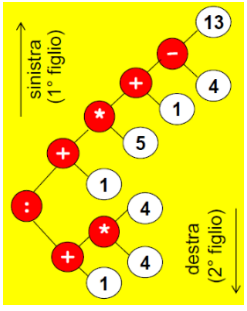
4	1		5		1		4	16	16	17		
13	9	9	10	10	50	50	51	51	51	51	51	3

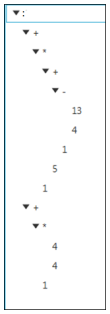


```
preorder: :, +, *, +, -, 13, 4, 1, 5, 1, +, *, 4, 4, 1
postorder: 13, 4, -, 1, +, 5, *, 1, +, 4, 4, *, 1, +, :
inorder: 13, -, 4, +, 1, *, 5, +, 1, :, 4, *, 4, +, 1,
espressione: (((((13)-(4))+(1))*(5))+(1)):(((4)*(4))+(1))
```

```
Il risultato di  13 4 - 1 + 5 * 1 + 4 4 * 1 + :  è  3
```

Nella valutazione con Stack **postfissa**, il layout sembra opposto perché si espande verso il basso e verso destra. In pratica basta ruotarlo di 90° per riconoscerlo.





7.2 – SWING

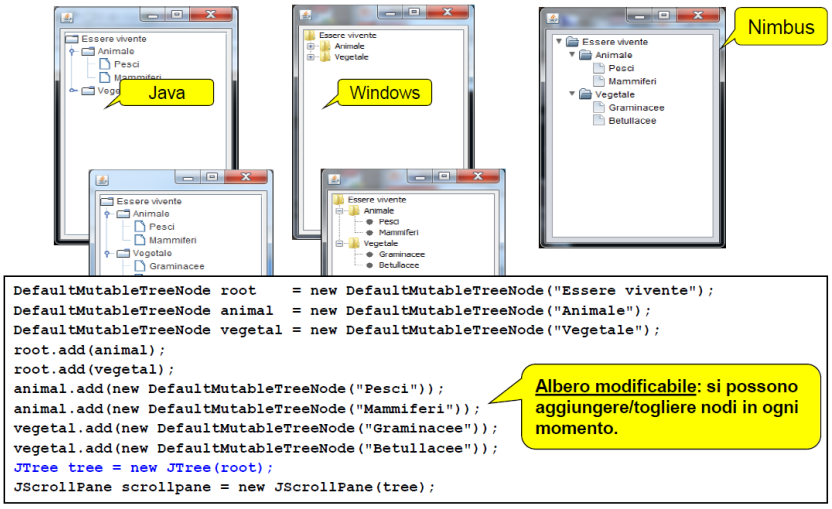
Swing offre un **componente per mostrare alberi: JTree:**

- Struttura gerarchica, facilmente espandibile / collassabile
- È possibile reagire all'evento "clic su un item"

Visualizza un albero inteso come struttura di TreeNode:

- **Approccio "C-like"**: l'albero coincide con il puntatore a un nodo, non esiste un concetto di "albero come oggetto"
- **È una interfaccia**, implementata da varie classi: l'implementazione più diffusa è **DefaultMutableTreeNode**
- Per visualizzare un **albero costruito da noi** occorre costruirsene un modello fatto di **DefaultMutableTreeNode**

7.2.1 – JTREE




```
DefaultMutableTreeNode root    = new DefaultMutableTreeNode("Essere vivente");
DefaultMutableTreeNode animal  = new DefaultMutableTreeNode("Animale");
DefaultMutableTreeNode vegetal = new DefaultMutableTreeNode("Vegetale");
root.add(animal);
root.add(vegetal);
animal.add(new DefaultMutableTreeNode("Pesci"));
animal.add(new DefaultMutableTreeNode("Mammiferi"));
vegetal.add(new DefaultMutableTreeNode("Graminacee"));
vegetal.add(new DefaultMutableTreeNode("Betullacee"));
JTree tree = new JTree(root);
JScrollPane scrollpane = new JScrollPane(tree);
```

Albero modificabile: si possono aggiungere/togliere nodi in ogni momento.

7.2.2 – TREENODE

Enumeration è una interfaccia che risale a Java2.
Antenata degli iteratori, è semplice ma molto limitata:

Modifier and Type	Method and Description
`void`	`insert(MutableTreeNode child, int index)` Adds `child` to the receiver at `index`.
`void`	`remove(int index)` Removes the child at `index` from the receiver.
`void`	`remove(MutableTreeNode node)` Removes `node` from the receiver.
`void`	`removeFromParent()` Removes the receiver from its parent.
`void`	`setParent(MutableTreeNode newParent)` Sets the parent of the receiver to `newParent`.
`void`	`setUserObject(Object object)` Resets the user object of the receiver to `object`.

Modifier and Type	Method and Description
`Enumeration`	`children()` Returns the children of the receiver as an `Enumeration`.
`boolean`	`getAllowsChildren()` Returns true if the receiver allows children.
`TreeNode`	`getChildAt(int childIndex)` Returns the child `TreeNode` at index `childIndex`.
`int`	`getChildCount()` Returns the number of children `TreeNode`s the receiver contains.
`int`	`getIndex(TreeNode node)` Returns the index of `node` in the receivers children.
`TreeNode`	`getParent()` Returns the parent `TreeNode` of the receiver.
`boolean`	`isLeaf()` Returns true if the receiver is a leaf.

permette di iterare sull'insieme una sola volta.
L'interfaccia MutableTreeNode estende TreeNode: è quella implementata da **Default Mutable Tree Node.**

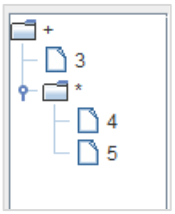
La classe DefaultMutableTreeNode fornisce decine di metodi utili – in particolare, le visite:

- **Enumeration breadthFirstEnumeration()** restituisce un'enumerazione che visita l'albero in **ampiezza**
- **Enumeration preorderEnumeration()** restituisce un'enumerazione che visita l'albero in **pre-order**
- **Enumeration postorderEnumeration()** restituisce un'enumerazione che visita l'albero in **post-order**
- **Enumeration children()** restituisce un'enumerazione che **elenca i figli** (da sx a dx)

Il **metodo getUserObject** recupera l'oggetto (un Object) eventualmente incapsulato nel nodo all'atto della costruzione.

Esempio – Espressione con JTree 3+4*5

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("+");
root.add(new DefaultMutableTreeNode(new Integer(3)));
DefaultMutableTreeNode node = new DefaultMutableTreeNode("*");
node.add(new DefaultMutableTreeNode(new Integer(4)));
node.add(new DefaultMutableTreeNode(new Integer(5)));
root.add(node);
JTree tree = new JTree(root);
JScrollPane scrollpane = new JScrollPane(tree);
scrollpane.setPreferredSize(new Dimension(100,400));
panel.add(scrollpane);
```



7.2.3 – VALUTAZIONE DELL'ALBERO

Per valutare un'espressione serve la visita in postorder, e DefaultMutableTreeNode la offre già pronta:
Enumeration<DefaultMutableTreeNode> e = root.postorderEnumeration();
Ora serve solo uno stack di appoggio → la JCF lo offre: Stack<Integer> stack = new Stack<Integer>();
Dunque, per valutare basta scandire l'enumerazione:

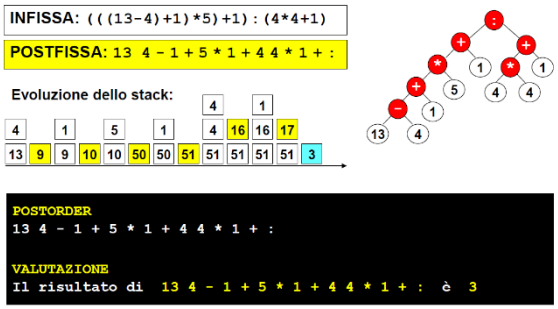
- Quando si incontra un numero, lo si mette sullo stack
- Quando si incontra un operatore, si svolge l'operazione
  - Prima, prendendo i due dati dallo stack
  - Poi, mettendo sullo stack il risultato

Esempio – Valutazione con Stack

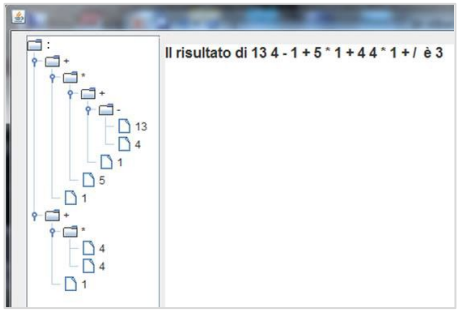
```
Stack<Integer> stack = new Stack<Integer>();
while (e.hasMoreElements()) {
    DefaultMutableTreeNode node = e.nextElement();
    Object obj = node.getUserObject(); // recupera oggetto interno
    if (obj instanceof Integer) stack.push((Integer)obj);
    else { // è un operatore
        String operator = (String) obj;
        Integer v2 = stack.pop();
        Integer v1 = stack.pop();
        switch (operator) {
              case "+": stack.push(v1 + v2); break;
              case "-": stack.push(v1 - v2); break;
              case "*": stack.push(v1 * v2); break;
              case ":": stack.push(v1 / v2); break;
        }
    }
}
```

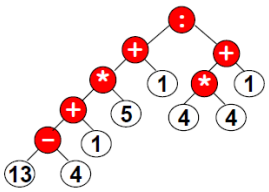
Auto-unboxing & auto-boxing

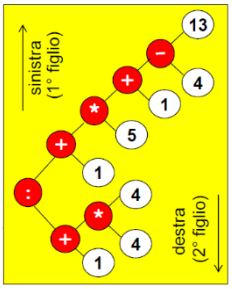
```
Il risultato di  3 4 5 * +  è  23
```



Se visualizziamo l'albero JTree e osserviamo quello generato nella figura precente, vediamo che anche qui il layout di JTree sembra "opposto" al nostro! JTree non disegna come ci aspetteremmo, ma in realtà basta ruotare il nostro albero di 90° per riconoscerlo, come prima.







8 – INTERPRETE ESTESO

8.1 – ASSEGNAMENTO

Tutti i linguaggi di programmazione introducono a un qualche livello le nozioni di variabile e assegnamento. Sebbene spesso si utilizzi l'operatore "=", l'assegnamento non è un'uguaglianza o equazione in senso matematico: non è simmetrico né riflessivo, infatti x=25 è molto diverso da 25=x.
Per questo, alcuni linguaggi come Pascal, utilizzano un simbolo di operatore "direzionale", che renda evidente la non riflessività (:=).
Questa difformità è anche una delle principali ragioni che rendono difficile l'apprendimento iniziale dell'informatica in senso tradizionale. Lo si vede molto bene nella scrittura x = x + 1 che come uguaglianza matematica sarebbe impossibile!
In matematica una volta che stabilisci il significato di una variabile, quella è, per sempre. Invece qui, nella stessa riga lo stesso simbolo cambia significato.

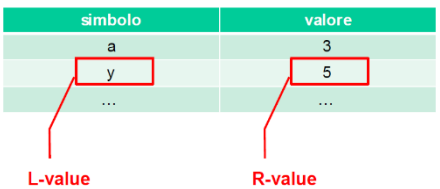
La peculiarità dell'assegnamento è che il simbolo di variabile ha significato diverso a destra e a sinistra dell'operatore =.
Il simbolo di variabile è **overloaded** → Il suo significato dipende dalla posizione rispetto all'operatore: **a sinistra indica il contenitore** (variabile in quanto tale), mentre **a destra indica il contenuto** (di quella variabile).

La **semantica informale** può quindi essere così riassunta:

- Prendere il valore della variabile a destra dell'operatore =
- Usarlo per valutare il valore dell'espressione a destra dell' = → **R-VALUE**: valore di destra
- Porre il risultato nella variabile specificata a sinistra dell' = → **L-VALUE**: valore di sinistra

Linguaggi diversi possono fare scelte diverse in merito a varie questioni chiave:

- Assegnamento distruttivo / non distruttivo: si può cambiare il valore associato in precedenza a un simbolo di variabile?
  - **I LINGUAGGI IMPERATIVI HANNO L'ASSEGNAMENTO DISTRUTTIVO**, perché distruggono la differenza tra variabile e contenuto. È difficile capire cosa fa un programma semplicemente leggendone il testo, bisogna simulare mentalmente l'evoluzione
  - **I LINGUAGGI LOGICI HANNO L'ASSEGNAMENTO NON DISTRUTTIVO** e quindi hanno **TRASPARENZA REFERENZIALE**: un simbolo ha sempre lo stesso significato ovunque, come nelle dimostrazioni matematiche
- È utile/opportuno distinguere sintatticamente L-value da R-value? Nei **linguaggi di shell** spesso si distingue: **x = $x+1** #bash



8.2 - ENVIRONMENT

Per esprimere la semantica dell'assegnamento occorre introdurre il **concetto di environment** inteso come **insieme di coppie (simbolo, valore)** → una tabella a due colonne (mappa). **L'assegnamento modifica l'environment** causando un effetto collaterale secondo la seguente semantica: **x = valore**
"3+5" è un alias per 8, ma non cambia l'ambiente perché posso ricalcolarlo altri 234958237 miliardi di volte. Invece non appena provo a fare "x+3", se lo calcolo adesso o se lo calcolo tra 10 minuti, come faccio a sapere se vale ancora come prima? Se il linguaggio mi garantisce l'assenza di assegnamento distruttivo, ok, altrimenti può cambiare valore istante per istante. C'è bisogno di mantenere da qualche parte il valore di x.

- Se non presente una coppia con primo elemento x, si inserisce nell'environment la coppia (x, valore)
- Se invece esiste già una coppia (x, v), vi sono due possibilità:
  - **Assegnamento distruttivo**: essa viene eliminata e sostituita dalla nuova coppia (x, valore)
  - **Singolo assegnamento**: essa viene mantenuta e il tentativo di nuovo assegnamento a x dà luogo a errore

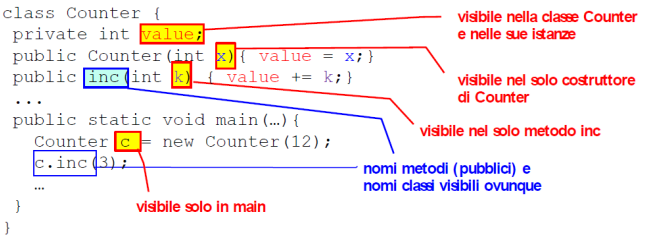
8.2.1 – ENVIRONMENT MULTIPLI

L'environment è spesso suddiviso in sotto-ambienti, di norma collegati al tempo di vita delle strutture run-time:

- **ENVIRONMENT GLOBALE**: contiene le coppie il cui tempo di vita è con l'intero programma
- **ENVIRONMENT LOCALI**: contengono coppie il cui tempo di vita non coincide con l'intero programma, sono tipicamente legati all'attivazione di funzioni o altre strutture run-time

Gli environment locali sono associati a **blocchi racchiusi tra graffe**, quindi ad esempio per ogni funzione in genere c'è un environment locale.

Ogni modello computazionale deve specificare il campo di visibilità dei suoi simboli (scope), ossia quali environment sono visibili in quali punti della struttura fisica del programma.



Per introdurre l'assegnamento in un linguaggio occorre **stabilire**:

- La **sintassi dei nomi** delle variabili
- **Se l'assegnamento sia distruttivo o meno**
- Se la scrittura di assegnamento x=valore sia un'istruzione o una espressione per supportare o meno l'**assegnamento multiplo**
  - **ISTRUZIONE**: **effettua un'azione ma non denota un valore**
  - **ESPRESSIONE**: **effettua un'azione e denota anche un valore** che costituisce il «risultato» dell'espressione

8.3 – ASSEGNAMENTO MULTIPLO

Di base lo scopo dell'assegnamento non è denotare un valore ma produrre un effetto collaterale nell'environment → **ha la natura di istruzione,** come in vari linguaggi (Pascal).
Tuttavia, tale approccio rende impossibile comporre assegnamenti singoli in un assegnamento multiplo x=y=z=valore perché se l'assegnamento è un'istruzione, gli assegnamenti "successivi " nella catena sono privi di significato.
Se ho x=y=2 e considero che y=2 è un'istruzione, cosa mi ritorna? Niente. Quindi dopo avrei che x è uguale a non si sa che cosa. In C, ad esempio, volevano propagare l'assegnamento, quindi y=… non è più un'istruzione, ma è un'espressione che ha un effetto collaterale: il risultato è sempre 2 ma nella tabella dell'environment si hanno ora più variabili che valgono 2.
Quindi in C e in Java e in tutti i linguaggi derivati, l'assegnamento è un'espressione e non un'istruzione.
Far finta che l'assegnamento sia un'espressione è il trucco che si è inventato il C per rendere possibile l'assegnamento multiplo.

L'operatore di assegnamento è necessariamente associativo a destra, perché il valore è l'ultimo elemento.
La frase precedente x = y = z = valore deve quindi essere interpretata come x = (y = (z = valore)).

Esempi

x = y = 3-2-1 → significato equivalente a x=(y=0), ovvero:

- y=0 denota il valore 0 e causa l'inserimento nell'environment della coppia (y,0)
- x=0 denota anch'essa il valore 0 (inutile) e causa l'inserimento nell'environment della nuova coppia (x,0)

k + π → corretta purché k e π siano definiti nell'ambiente al momento della valutazione

y=(x=y=2)+3 → significato equivalente alla sequenza:

- x=(y=2) che denota il valore 2 e causa l'inserimento nell'environment delle due coppie (y,2) e (x,2), in quest'ordine
- y=2+3 che denota il valore 5 e causa l'inseri-mento nell'environment della coppia (y,5) in sostituzione della precedente (y,2)

8.4 – ESTENSIONE DELL'INTERPRETE

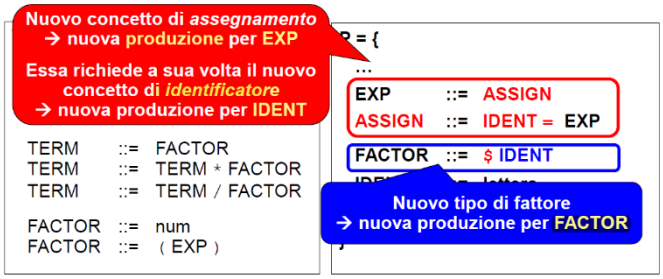
Riassunto: se cominciamo a mettere in pista le variabili dobbiamo distinguere il loro valore di sinistra con il loro valore di destra. Il nome di sinistra ha un significato completamente diverso da quello sul lato destro perché in un caso significa la scatola, nell'altro il contenuto della scatola e le due cose non coincidono.
Un altro elemento che dobbiamo valutare (che distingue i linguaggi imperativi dagli altri) è se l'assegnamento possa essere distruttivo o no, cioè se sovrascrive un precedente valore. La difficoltà dei linguaggi imperativi è proprio questa perché le variabili cambiano valore.
Abbiamo anche discusso come realizzare il concetto di variabile: concretamente si può fare in mille modi, ma concettualmente bisogna utilizzare delle tabelle in cui si associano dei simboli ai relativi valori. Ogni volta che nasce un nuovo simbolo immaginiamo che nasca una nuova riga della tabella. Se l'assegnamento non è distruttivo, il valore rimane sempre lo stesso nella tabella.
Abbiamo infine discusso se l'istruzione di assegnamento debba essere un qualcosa come il while, il for, ma poi non restituisce un valore (caratteristica delle istruzioni) oppure se debba essere un'espressione e restituisca un valore. L'obiettivo dell'assegnamento non è cambiare il mondo, creare effetti collaterali, quindi si può interpretare come un'istruzione. Un'istruzione lascia una traccia nel mondo, mentre un'espressione può essere valutata tante volte tanto non ha effetti collaterali sul mondo.
Però, nei linguaggi di derivazione C si usa l'assegnamento come espressione perché così è permesso l'assegnamento multiplo. In questo caso, l'operatore di assegnamento è necessariamente associativo a destra perché si parte da destra e il valore specificato nell'estrema destra è quello che poi ripercorre tutte le variabili alla sua sinistra. **Le espressioni dovrebbero essere idempotenti**, ma ce ne sono alcune che non funzionano così come il ++ e il -- → i++ != ++i

Per estendere il nostro interprete occorre aggiungere:

- L'espressione di assegnamento (operatore = o altro gradito) → "variabile = espressione"
- Il concetto di variabile nelle sue due interpretazioni di
    - **L-value**: il nome di variabile indica il contenitore (x = …)
    - **R-value**: il nome di variabile indica il contenuto (… = x+2)

Stessa sintassi destra/sinistra o sintassi differenziata? Si può usare una sintassi diversa per elemento a sinistra o a destra dell'operatore oppure (come in C e Java) l'utente può scrivere x sia a sinistra sia a destra e poi l'interprete capisce da solo di cosa si tratta. Se si fa la seconda scelta, serve un interprete LL(2).
Visto che l'uguale è simmetrico perché in matematica fa pensare all'uguaglianza, manteniamo lo stesso simbolo o lo cambiamo? Tipo in C "=" è diverso da "==". In Javascript è diverso ancora perché c'è "===".
Se scelgo sempre "=", devo inserire una nuova produzione che si chiamerà AssignExp.

Così facendo la grammatica è LL(2): 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 "="**
È fattibile, ma scomodo e meno efficiente.
Come alternativa si può differenziare sintatticamente L-Value da R-Value.
Nei parser veri (generati da strumenti automatici) si accetta la scomodità dell'LL(2), localmente. Nel nostro prototipo fatto a mano, invece, distingueremo aggiungendo un $ come in bash per R-value.



La grammatica così modificata è LL(1) perché IDENT "destro" è ora preceduto da $, ergo IDENT da solo identifica la ASSIGN.

Parimenti occorre estendere la sintassi astratta con tre nuovi tipi di nodo per l'AST, i.e. tre nuove classi Java:

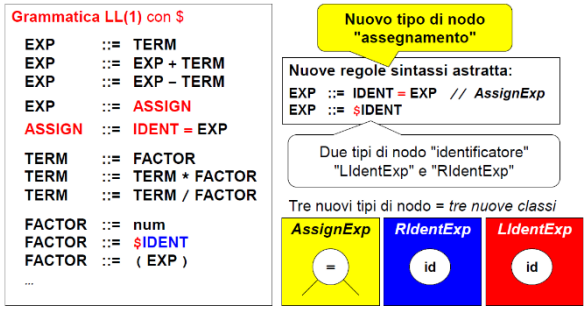
- Il nodo operazione di assegnamento (=)
- Il nodo L-value (IDENT a sinistra dell'=)
- Il nodo R-value (IDENT a destra dell'=)

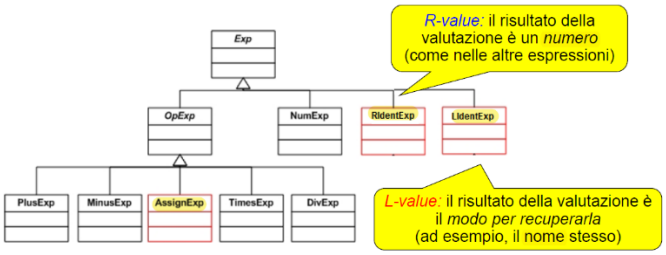
Per la valutazione degli identificatori si usa la seguente semantica informale:

- **Valutare nodo L-value**: ricavare un modo per accedere al contenitore avente nome = quel simbolo
- **Valutare nodo R-value**: ricavare il valore associato nell'environment a quel simbolo

Due classi distinte per rappresentare L-Value e R-Value:

- **LIdentExp** ha un **nome** che è usabile come "modo per accedere" → CHIAVE nella mappa
- **RIdentExp** ha un **nome** e un **valore associato** (nell'environment) → VALORE nella mappa





simbolo (chiave)	valore
a	3
y	5
…	…

L-value R-value

Esempio – Schema di Parser

- **La classe Token ha un nuovo metodo** di servizio **isIdentifier** che cattura la sintassi degli identificatori
- **parseExp cattura anche la nuova espressione Assign**, intercettando anche la sequenza IDENT = EXP
- **parseFactor cattura anche il nuovo fattore RIdent**, intercettando anche la sequenza $ IDENT

PRIMA

```
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;            // errore
   }
   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;            // errore
   }
   else return termSeq;
 } // end while
 return termSeq;
}
```

Aggiungere qui il nuovo caso *Assign*

```
public Exp parseFactor() {
  if (currentToken.equals("(")) {
     currentToken = scanner.getNextToken();
     Exp innerExp = parseExp();     // PDA: gestione self-embedding
     if (currentToken.equals(")")) {
          currentToken = scanner.getNextToken();
          return innerExp;          // le parentesi vengono omesse
     } else return null;            // errore
  }
  else    // dev'essere un numero
  if (currentToken.isNumber()) {
     int value = currentToken.getAsInt();
     currentToken = scanner.getNextToken();
     return new NumExp(value);
  }
  else                 // errore: non è un fattore
     return null;      // non si è costruito nulla, restituiamo null
}
```

Aggiungere qui il nuovo caso *$ Ident*

DOPO

```
public Exp parseExp() {
 Exp termSeq = parseTerm();
   ...
   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;            // errore
   } else if (currentToken.isIdentifier()) {
        String id = currentToken.toString();
        currentToken = scanner.getNextToken();
        if (!(currentToken.equals("="))) return null;
        currentToken = scanner.getNextToken();
        Exp rightExp = parseExp();
        return new AssignExp( new LIdentExp(id), rightExp);
   } else
        return termSeq; // next token non fa parte di L(Exp)
 } // end while
 return termSeq;    // next token è nullo -> stringa di input finita
}
```

```
public Exp parseFactor() {
  ...
    else // nuova regola: identificatori come R-Value
    if (currentToken.equals("$")) {
        currentToken = scanner.getNextToken();
        String id = currentToken.toString );
        currentToken = scanner.getNextToken();
        return new RIdentExp(id);
    }
  else                 // dev'essere un numero
  if (currentToken.isNumber()) {
    int value = currentToken.getAsInt();
    currentToken = scanner.getNextToken();
    return new NumExp(value);
  }
  else                 // errore: non è un fattore
    return null;       // non si è costruito nulla, restituiamo null
}
```

Esempio espressioni: x = 5 – 3; y = 4 + $x;
Se scambio l'ordine delle due funzioni, esplode tutto perché non posso fare 4+x se prima non ho assegnato ad x un valore.

8.4.1 - VISITOR

L'interfaccia (o classe astratta) ExpVisitor resta identica (altrimenti tutti i visitor già realizzati dovrebbero implementare i tre nuovi metodi): **si introduce una sua specializzazione ExpAssignVisitor.**

```
interface ExpAssignVisitor extends ExpVisitor {
  public abstract void visit( AssignExp e );
  public abstract void visit( LIdentExp e );
  public abstract void visit( RIdentExp e );
}
```

Refactoring dei visitor:

- Il visitor visualizzatore, ParExpVisitor, non ha problemi
- **Il visitor valutatore, EvalExpVisitor**, invece, non può più presumere che il risultato della valutazione sia sempre un numero perché nel caso di IdentExp non lo è → generalizzare il tipo di ritorno delle funzioni

Dentro la variabile lato destro ho sia il valore della variabile, sia il nome della variabile, quindi per comodità mi tengo entrambe le cose.

```
class AssignExp extends OpExp {
  public AssignExp ( Exp l, Exp r) { super(l,r); }
  public String myOp() { return "=" ; }
  public void accept( ExpVisitor v) { ((ExpAssignVisitor)v).visit(this);}
}
class LIdentExp extends Exp {
  String name;
  public LIdentExp(String v) { name = v; }
  public String toString() { return name; }
  public String getName() { return name; }
  public void accept( ExpVisitor v) { ((ExpAssignVisitor)v).visit(this); }
}
class RIdentExp extends Exp {
  String name;
  int value;
  public RIdentExp(String v) { name = v; }
  public String toString() { return name; }
  public String getName() { return name; }
  public int getValue() { return value; }
  public void accept( ExpVisitor v) { ((AssignExpVisitor)v).visit(this);}
}
```

Cast *necessario* perché i nuovi metodi esistono solo in ExpAssignVisitor

VISITOR VISUALIZZATORE

Se stampo l'albero con visita in pre-order:

VISITOR VALUTATORE

Il risultato della valutazione non è più sempre un numero perché LIdentExp denota un qualche "riferimento" alla variabile. Refactoring: le funzioni restituiscono un Object.

Mentre per il 99% delle espressioni abbiamo a che fare con un int, nella valutazione della variabile sinistra non ho più un int ma ho un riferimento alla variabile, a dove si trova. Per farla alla brutta si può fare tutto Object e un incapsulamento degli int in Integer, tanto boxing e unboxing sono automatici, ma qui è stato scritto perché i riferimenti sono tutti wrappati.

```
class ParExpVisitor implements ExpVisitor, ExpAssignVisitor {
 ...
 public void visit( LIdentExp e ) {
   curs = e.getName();
 }
 public void visit( RIdentExp e ) {
   curs = e.getName();
 }
 public void visit( AssignExp e ) { visitOpExp(e); }
}
```

Estensione per gestire le tre nuove espressioni

SCELTA: il lato sinistro e destro sono stampati in modo identico

```
class EvalExpVisitor implements ExpVisitor {
  Object value;                    // memorizzazione risultato
  public Object getEvaluation() { return value; }
  public void visit(PlusExp e) {
    e.left().accept(this);
    int arg1 = ((Integer)getEvaluation());
    e.right().accept(this);
    int arg2 = ((Integer)getEvaluation());
    value = new Integer(arg1 + arg2);
  }
  // analogamente per le altre categorie di espressioni
}
```

L'assegnamento richiede un environment e noi lo realizziamo tramite una Map<String,Integer> che memorizza le coppie (nomevariabile, valore).

Valutazioni:

- Valutare **LIdentExp** significa ricavare la **chiave** della mappa
- Valutare **RIdentExp** significa ricavare il **valore corrispondente** a una data chiave nella mappa
- Valutare **AssignExp** significa **inserire nella mappa** una nuova entry (eventualmente rimpiazzando quella esistente con la stessa chiave, ossia lo stesso nome di variabile → semantica di assegnamento distruttivo)

```
class EvalAssignExpVisitor
  extends EvalExpVisitor implements ExpAssignVisitor {
 Map<String,Integer> environment = new HashMap<>();
 public void visit(AssignExp e) {
   e.left().accept(this);    // a sinistra c'è una IdentExp
                             // → recuperiamo la CHIAVE
   String id = ((LIdentExp)e.left()).getName();
   e.right().accept(this);   // a destra c'è una Exp qualsiasi
   value = getEvaluation();  // → recuperiamo il VALORE
   // è anche il risultato dell'exp → assegnamento multiplo OK
   environment.put(id, value);       // inseriamo la entry
 }
 ...
```

Map garantisce che se esiste già una entry con la stessa chiave, viene rimpiazzata

```
...
public void visit(LIdentExp e) {
  // valutare IndentExp → restituire la chiave (il nome)
  value = e.getName();
}
public void visit(RIdentExp e) {
  // valutare IndentValExp → restituire il valore corrisp.
  String id = e.getName();
  Integer val = environment.get(id); // recuperiamo il valore
  if (val != null) value = val;    // memorizzazione risultato
  else { // non dovrebbe mai accadere!
    throw new RuntimeException("invalid identifier");
  }
 }
}
```

```
public class NewTest {
  // Creazione di espressioni di assegnamento
  // - a sinistra dell'operatore di assegnamento, un IDENT
  // - a destra di tale operatore, una espressione qualsiasi
  Exp s6 = new AssignExp( new LIdentExp("x") ,      // x = 5-3
            new MinusExp( new NumExp(5), new NumExp(3)) );
  Exp s7 = new AssignExp( new LIdentExp("y") ,      // y = 4+x
            new PlusExp( new NumExp(4), new RIdentExp("x")));
  Exp s8 = new AssignExp( new LIdentExp("y") ,      // y = -4*y
            new TimesExp( new NumExp(-4), new RIdentExp("y")));
  Exp s9 = new AssignExp( new LIdentExp("z") ,      // z = x = y
            new AssignExp( new LIdentExp("x") ,
             new RIdentExp("y")));
...
```

assegnamento multiplo (z=x=y)

L-value: il "valore" del nodo è la chiave

R-value: il "valore" del nodo è il valore della variabile

Visitor visualizzatore

```
...
ParExpVisitor visitor = new ParExpVisitor();
s6.accept(visitor);
System.out.println(s6 + "\t" + visitor.getVal() );
s7.accept(visitor);
System.out.println(s7 + "\t" + visitor.getVal() );
s8.accept(visitor);
System.out.println(s8 + "\t" + visitor.getVal() );
s9.accept(visitor);
System.out.println(s9 + "\t" + visitor.getVal() );
...
```

```
x=5-3   ( = x ( - 5 3 ) )
y=4+x   ( = y ( + 4 x ) )
y=-4*y  ( = y ( * -4 y ) )
z=x=y   ( = z ( = x y ) )
```

Visitor valutatore

```
...
EvalAssignExpVisitor evisitor = new EvalAssignExpVisitor();
s6.accept(evisitor);
System.out.println(s6 + "  =  " + evisitor.getEvaluation() );
s7.accept(evisitor);
System.out.println(s7 + "  =  " + evisitor.getEvaluation() );
s8.accept(evisitor);
System.out.println(s8 + "  =  " + evisitor.getEvaluation() );
s9.accept(evisitor);
System.out.println(s9 + "  =  " + evisitor.getEvaluation() );
}
```

```
x=5-3  =  2
y=4+x  =  6
y=-4*y  =  -24
z=x=y  =  -24
```

NOTA: ora `getEvaluation` restituisce un `Object`, ma la stampa avviene tramite `toString` che è polimorfa, ergo tutto funziona senza modifiche:
- la `toString` di `Integer` stampa il valore intero
- la `toString` di `String` stampa il nome variabile

8.4.2 – ESPRESSIONI SEQUENZA

Cominciamo ad andare verso un programma un po' più vero. Tipicamente si scrivono frasi più lunghe quindi immaginiamo di prendere più espressioni, una vicino all'altro. Il concetto di espressione a sequenza serve per creare un piccolo programma.
Se io scrivo 3+4, 3+4*5 valuto entrambe le espressioni, ma tengo solamente il risultato dell'ultima espressione. **Ha senso usare più espressioni in sequenza con la virgola solo se l'espressione precedente assegna ad una variabile un valore**, quindi come **vincolo** metto che **la prima operazione deve essere un assegnamento.**

Nuovo obiettivo: aggiungere espressioni-sequenza (sequenze di espressioni separate da virgola).

- IPOTESI 1: la prima espressione è sempre un assegnamento
- IPOTESI 2: il valore complessivo è quello dell'exp più a destra

Sintassi astratta: EXP ::= ASSIGN , EXP //SeqExp

Nuova grammatica di principio

```
EXP    ::= TERM
EXP    ::= EXP + TERM
EXP    ::= EXP – TERM
EXP    ::= ASSIGN
EXP    ::= ASSIGN , EXP
ASSIGN ::= IDENT = EXP
...
```

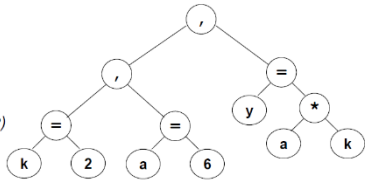
SEMANTICA INFORMALE:

- Si valuta il nodo **figlio sinistro** (produce effetti collaterali nell'environment)
- Si valuta il nodo **figlio destro** (produce effetti collaterali nell'environment)
- Il valore del nodo sequenza coincide col valore del figlio destro

Esempi

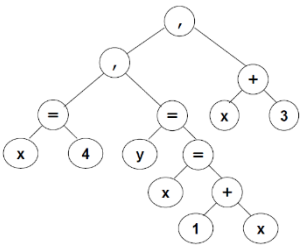
`k=2, a=6, y=a*k`

- Valuta le tre sub-espressioni, da sinistra a destra
- `k=2` causa l'inserimento nello environment della coppia *(k,2)* *[la sub-espressione in sé vale 2 ma tale valore non è utilizzato]*
- `a=6` causa l'inserimento nell'environment della coppia *(a,6)* *[la sub-espressione in sé vale 6 ma tale valore non è utilizzato]*
- `y=a*k` causa l'estrazione dall'environment del valore attuale di a e k, indi l'inserimento nell'environment della coppia *(y,12)*. Il risultato dell'espressione-sequenza in quanto tale è 12



`x=4, y=x=1+x, x+3`

- Valuta le tre sub-espressioni sequenza da sinistra a destra
- `x=4` causa l'inserimento nell'environment della coppia *(x,4)*
- `y=x=1+x` recupera dall'environment il valore attuale di x, indi inserisce nello environment la coppia *(x,5) [prima]* e la coppia *(y,5) [poi]*
- `x+3` recupera dall'environment il valore attuale di x e lo somma alla costante 3, ottenendo il risultato finale 8.



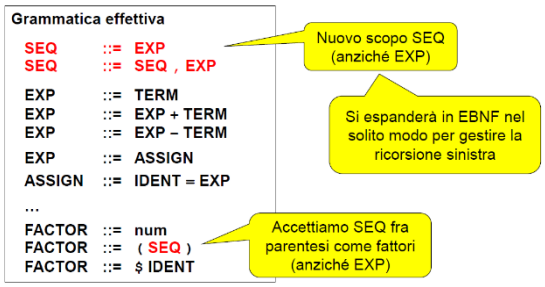
REVISIONE

Se seguiamo esattamente questa grammatica, **parseExp diventa lunga e poco leggibile** (anche perché non abbiamo previsto una parseAssign esplicita). Soprattutto, **ci tarpiamo le ali da soli se domani volessimo aggiungere nuovi tipi di espressioni atte a stare "anche davanti" alla virgola** (come Assign).

Perciò, revisioniamo leggermente la grammatica:

- **Aggiungendo un nuovo livello**, la sequenza di espressioni, che diventerà il nuovo scopo

- Stabilendo che la **sequenza sia semplicemente una sequenza di espressioni**, senza imporre vincoli sul primo elemento (in pratica sto facendo quello che fa il C)




```
public Exp parseFactor() {
   if (currentToken.equals("(")) {
      currentToken = scanner.getNextToken();
      Exp innerExp = parseSeq(); // SEQUENZA dentro parentesi
      if (currentToken.equals(")")) {
         currentToken = scanner.getNextToken();
         return innerExp;
      } else return null;
   }
   …
   …
}
```

```
public Exp parseSeq() {
   Exp expSequence = parseExp();
   while (currentToken != null) {
      if (currentToken.equals(",")) {
         currentToken = scanner.getNextToken();
         Exp nextExp = parseExp();
         if (nextExp != null)
            expSequence = new SeqExp(expSequence, nextExp);
         else
            return null;
      } else
         return expSequence;
   } // end while
   return expSequence;
}
```

```
public class TestParser {
   public static void main(String args[]) {
      String[] expressions = {
        ...
        "z = x = $ y",
        "x = 5 , y = $ x",
        "y = 4 + $ x , 3 - 5",
        "x = 7 , y = 4 + $ x",
        "x = 7 , y = 4 + $ x , w = $ y + 1",
        "( x = 7 , y = 4 + $ x ) , w = $ y + 1",
        "x = 7 , ( y = 4 + $ x , w = $ y + 1 )"
      };
      String expression = expressions[13];
      Scanner scanner = new MyStringScanner(expression);
      MyParser parser = new MyParser(scanner);
      Exp ast = parser.parseSeq();        // NUOVO SCOPO GRAMMATICA
      System.out.println(ast);            // da valutare poi
   }
}
```

Ad esempio, l'ultima espressione stampa `x=7,y=4+x,w=y+1`

ESTENSIONE DEI VISITOR VALUTATORI

Il nuovo ExpSeqVisitor specializza ExpAssignVisitor perché la sequenza, per come è definita, ha senso solo se esiste già la nozione di assegnamento. Questo è un esempio didattico in cui devo cambiare tutti i cast se no i tipi non tornano, ma se uno fa refactoring nella realtà, la fa per bene.

```
interface ExpSeqVisitor extends ExpAssignVisitor {
  public abstract void visit( SeqExp e );
}
```

```
class SeqExp extends OpExp {
  public SeqExp ( Exp l, Exp r) { super(l,r); }
  public String myOp() { return "," ; }
  public void accept( ExpVisitor v) {
    ((ExpSeqVisitor)v).visit(this);}
}
```

```
class ParExpVisitor implements
   ExpAssignVisitor, ExpSeqVisitor {
  ...
  public void visit( SeqExp e ) { visitOpExp(e); }
}
```

Visitor visualizzatore

```
class EvalSeqExpVisitor
  extends EvalAssignExpVisitor
  implements ExpSeqVisitor {
  public void visit( SeqExp e ) {
    e.left().accept(this);   // figlio sx → va solo valutato
                             // per causare i side effect
    e.right().accept(this);  // figlio dx → va valutato e
    value = getEvaluation(); // anche restituito
  }
}
```

Visitor valutatore

```
Exp[] expressions = {
 new SeqExp( // x=5, y=x
    new AssignExp(new LeftIdentExp("x"),new NumExp(5)),
    new AssignExp(new LeftIdentExp("y"),new RightIdentExp("x"))),
 new SeqExp( // y=4+x, 3-5
    new AssignExp( new LeftIdentExp("y"),
                   new PlusExp( new NumExp(4), new RightIdentExp("x"))),
    new MinusExp(new NumExp(3), new NumExp(5)) ),
 new SeqExp( // x=7, y=4+x
    new AssignExp( new LeftIdentExp("x"), new NumExp(7) ),
    new AssignExp( new LeftIdentExp("y") ,new PlusExp(
                                  new NumExp(4), new RightIdentExp("x")))),
 new SeqExp( // (x=7, y=4+x), w=y+1
    new SeqExp( // x=7, y=4+x
       new AssignExp( new LeftIdentExp("x"), new NumExp(7) ),
       new AssignExp( new LeftIdentExp("y") ,
                      new PlusExp( new NumExp(4), new RightIdentExp("x")))),
    new AssignExp( new LeftIdentExp("w"),
                   new PlusExp( new RightIdentExp("y"),new NumExp(1))) )
};
```

```
public class TestVisitors  {
  public static void main(String args[]) {
    Exp[] expressions = {
       ...
    };
    NicePrintExpVisitor printVisitor = new NicePrintExpVisitor();
    EvalExpVisitor evalVisitor = new EvalExpVisitor();
    for(Exp expression : expressions){
       expression.accept(printVisitor);
       expression.accept(evalVisitor);
       System.out.print("L'espressione " + printVisitor.getResult()
                        + " ha come risultato " + evalVisitor.getResult());
       System.out.println("\tEnvironment: " + evalVisitor.getEnv());
    }
  }
```

Una serie di Exp (cioè di AST)

L'espressione 3-5 ha come risultato -2 Environment: {}
L'espressione (2+(5*4))-1 ha come risultato 21 Environment: {}
L'espressione (3-5)-(1*5) ha come risultato -7 Environment: {}
L'espressione 5-(3-1) ha come risultato 3 Environment: {}

L'espressione (5-3)-1 ha come risultato 1	Environment: {}
L'espressione x=(5-3) ha come risultato 2	Environment: {x=2}
L'espressione y=(4+x) ha come risultato 6	Environment: {x=2, y=6}
L'espressione y=(-4*y) ha come risultato -24	Environment: {x=2, y=-24}
L'espressione z=(x=y) ha come risultato -24	Environment: {x=-24, y=-24, z=-24}
L'espressione (x=5),(y=x) ha come risultato 5	Environment: {x=5, y=5, z=-24}
L'espressione (y=(4+x)),(3-5) ha come risultato -2	Environment: {x=5, y=9, z=-24}
L'espressione (x=7),(y=(4+x)) ha come risultato 11	Environment: {x=7, y=11, z=-24}
L'espressione ((x=7),(y=(4+x))),(w=(y+1)) ha come risultato 12	Environment: {w=12, x=7, y=11, z=-24}

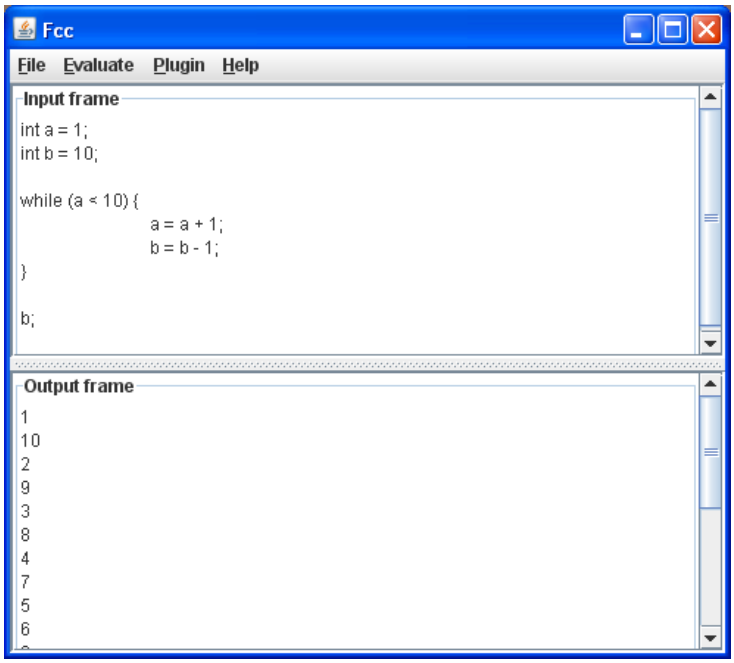
Riassumendo, il parser del "linguaggio espressioni" ora è completo poiché valuta espressioni su interi, con variabili e sequenze. Concettualmente non è molto distante da un "vero" linguaggio, ma mancano cose come le espressioni relazionali e le strutture for, while, if… **Se volessimo un compilatore basterebbe che un visitor emettesse codice, magari per una Extended Arithmetic Stack Machine con variabili e sequenze.**

9 – LAB con INTERPRETE per “Small C”

Esempio pratico di realizzazione di una sottospecie di “C” realizzato da uno studente.

- Un solo **tipo** di dati: **int** (inclusi boolean)
- **Operatori** aritmetici, boolean, relazionali e assegnamento
- **Istruzioni di selezione (if) e di iterazione (while)**
- **Blocchi innestati con (gerarchia di) environment**
- **Environment multipli**
    - **Un environment globale + tanti environment locali ai blocchi**
    - Tempo di vita di un environment pari al corrispondente blocco
    - La ricerca di un simbolo avviene a partire dall'environment corrente
    - Limite inferiore della catena: l'environment globale

La grammatica usa EBNF, self-embedding.
Per le espressioni aritmetiche e booleane ha realizzato sia la grammatica LL(1) che non.
Ha usato l’analisi ricorsiva discendente con tassonomia.
Ha usato il metodo visit() per valutare il linguaggio ed avere il risultato di alcune espressioni.



10 – GENERAZIONE AUTOMATICA DI RICONOSCITORI LL

10.1 – PARSER GENERATOR

I **PARSER GENERATOR** (o compiler compiler), sono strumenti che, **data una grammatica** (annotata), **producono automaticamente il riconoscitore in un linguaggio prescelto.**

Alcuni seguono l'approccio LL, altri le tecniche LR:

- **L'approccio LL è efficiente e facilita l'aggiunta di azioni semantiche**, messaggi d'errore, quickfix, etc.
  - I più diffusi sono **ANTLR** (per Java, C# e molti altri), **JavaCC** (per Java)
- **L'approccio LR è più potente** (in particolare, gestisce senza problemi la ricorsione sinistra), **ma più complesso**
  - I più diffusi sono **YACC & Lex**, **CUP & JLex**, **SableCC**, **GOLDParser**

Tuttavia, oggi gli strumenti per essere "veramente usati" devono integrarsi nei cicli di sviluppo e nei framework più diffusi: non basta più "generare il riconoscitore" (o l'interprete), ma **occorre saper offrire una soluzione completa, con:**

- **Editor** (con syntax highlighting)
- **Integrazione con strumenti di sviluppo** (Eclipse, IntelliJ, etc.)
  - Validazione
  - Segnalazione errori, content assist, quick fix, vista outline…
- **Velocità di realizzazione e prototipazione**

Il parser generator diviene quindi solo un componente di un workflow molto più completo (e complesso).

10.2 – DOMAIN-SPECIFIC LANGUAGES (DSL)

Un'area di recente sviluppo dei linguaggi è quella dei **Domain-Specific Languages (DSL)**: al contrario dei classici linguaggi di uso generale (General-Purpose Languages), **i DSL sono progettati per un particolare dominio applicativo** – anche molto ristretto.
Possono essere linguaggi di programmazione o di specifica, e non mirano a risolvere ogni problema (come un GPL), ma ad **offrire una soluzione efficace in uno specifico ambito di applicazione**. Casi tipici di DSL sono **HTML, SQL**, etc. (non è un tema nuovo: XML, JSON hanno obiettivi simili)

La sfida è quindi quella di avere linguaggi con meno "rumore sintattico", più pratici da usare proprio perché specifici e non generali.
Usare un DSL implica però definirlo e implementarlo → sintassi, semantica, compilatore, AST, …
Nonché, appunto, predisporre tutto il framework di supporto: editor, integrazione nei framework, ...
Il nuovo obiettivo diventa quindi avere strumenti per la generazione automatica di DSL completi di tutto il supporto.

XTEXT è uno dei **framework più diffusi di questo nuovo tipo:**

- Copre tutti gli aspetti della infrastruttura linguistica di un DSL
- Integrazione con tutti i framework più diffusi (tra cui Eclipse)
- Internamente usa ANTLR come parser generator

Esiste anche un pacchetto già pronto con Eclipse + Xtext & Xtend → Eclipse IDE for Java and DSL Developers

10.3 – ANTLR v4

ANTLR (ANother Tool for Language Recognition): strumento completo per la **generazione di parser LL(k):**

- Licenza BSD, incluso in distribuzioni Linux e altri tool (licenza gratis)
- 5000+ download/mese, ampia community, attivo e manutenuto
- Integrabile con Visual Studio Code (con rispettive estensioni)
- Disponibile come plugin per Eclipse, Netbeans, IntelliJ, jEDIT
- Disponibile anche un IDE stand alone (ANTLRworks 2)

L'originale algoritmo Adaptive LL(*) va oltre il puro LL(k)

- **Accetta ricorsione sinistra diretta** (ristrutturata internamente)
- **Molto efficiente**, prestazioni adattative nel runtime

Può generare codice per molti linguaggi come Java, C#, C, C++, Python, JavaScript, Go, Swift.

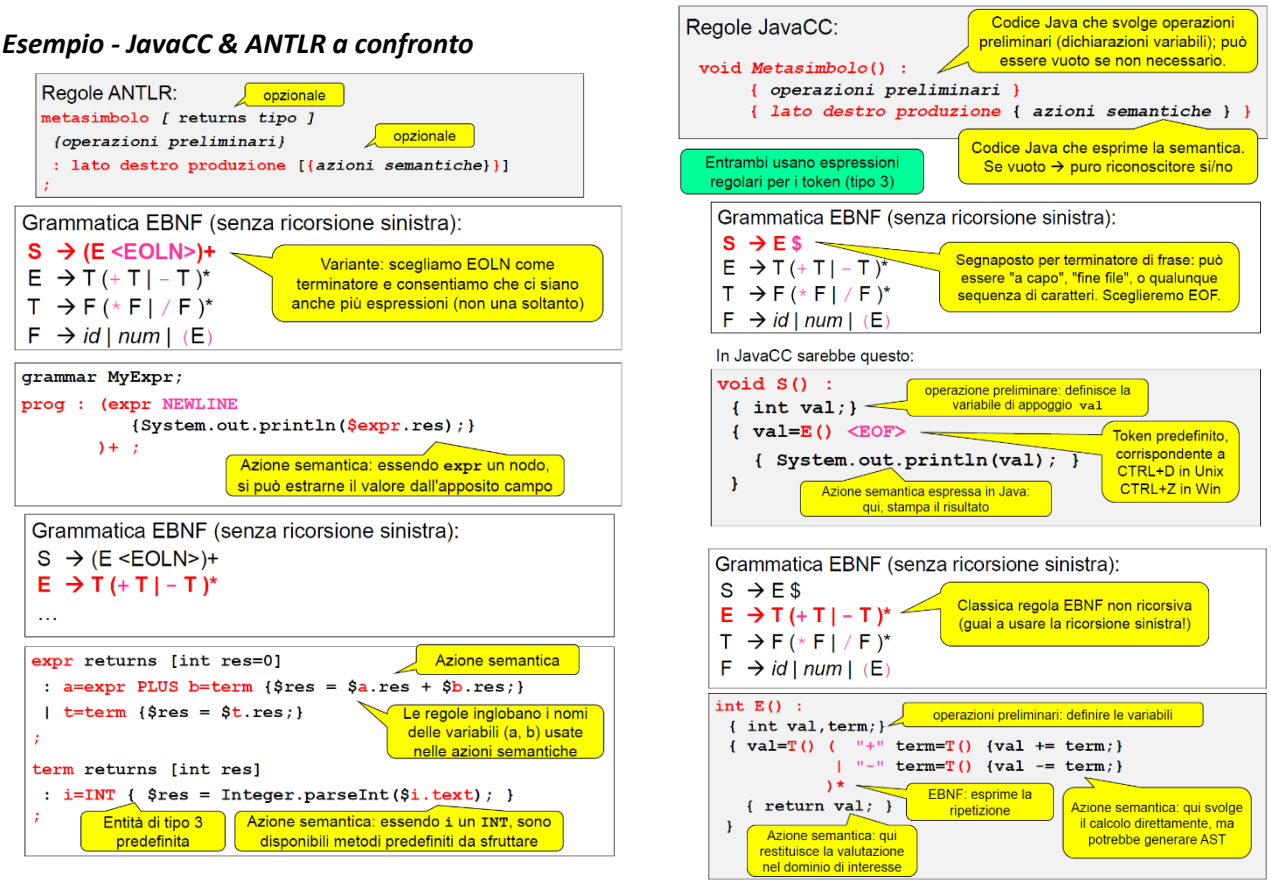
Al di là della specifica sintassi, **gli strumenti come ANTLR condividono le stesse idee di fondo:**

- La **sintassi** si scrive "circa" in **EBNF**
- Le **regole** possono essere **annotate con azioni semantiche**
- Il **generatore produce codice nel linguaggio target prescelto**, che deve poi essere compilato e/o integrato nel progetto normalmente

Le **differenze** riguardano:

- La **specifica sintassi per esprimere regole e azioni** semantiche
- Il **linguaggio target** (solo Java / Java e molti altri in ANTLR)
- La **disponibilità o meno di IDE grafici** (stand alone o plugin) per Eclipse, Visual Studio, IntelliJ…

Esempio - JavaCC & ANTLR a confronto



10.4 – DA ANTLR A XTEXT

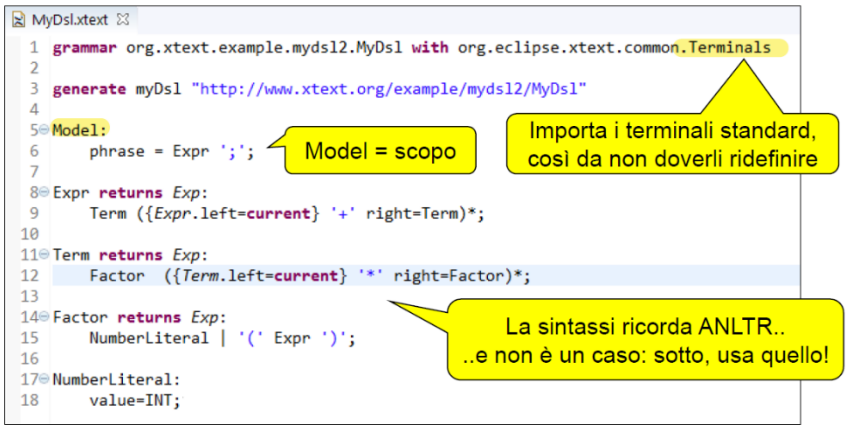
A differenza dei parser generator classici, **Xtext è model-based** e adotta l'**approccio a trasformazione di modelli:**

- La sintassi si scrive sostanzialmente come quella di ANTLR
- **Le azioni semantiche sono in componenti separati**, che vengono poi integrati nel workflow di Eclipse
- **Il generatore Xtext produce una infrastruttura completa** (parser, typechecker, compiler, editor per Eclipse, IDEA, browser web…)

Questo cambia completamente il modo di lavorare perché **dalla grammatica si generano di Xtext Artifacts**, che vengono poi **fatti girare in una nuova (separata) istanza di Eclipse**. Tale nuova istanza **ingloba il supporto per il nuovo linguaggio** (editor, segnalazione errori, quickfix, tooltip… tutti inclusi).

In pratica io specifico la grammatica e lui la elabora in un primo passo producendo nuove cose in cui c'è di tutto e di più chiamati "artefatti" (entità fatta ad arte per catturare certi elementi eterogenei).
Questi poi fanno girare una nuova istanza di Eclipse (quindi si hanno due istanze che girano contemporaneamente) che produce come effetto quello di arricchire Eclipse stesso.
Nel file di testo ".xtext" si specifica la grammatica, bisogna dire come generarla e dov'è il linguaggio che intendiamo. Lo scopo della grammatica è il Model e poi ci sono le regole.
Tutta questa cosa viene salvata e generata, i.e. vengono generati gli artefatti che producono una bella trafila di file e li si fa girare per conto loro: è proprio una nuova applicazione.
Ora abbiamo che Eclipse conosce il mio linguaggio e posso usarlo proprio come userei Java. Quindi, se creo una nuova classe con il mio nuovo linguaggio, lui riconosce già gli identificativi, come se si scrivesse ad esempio "Public" che io però ho chiamato "Hello".

Esempio
Xtext è model-based, quindi lo scopo è un Model, però la sintassi di base rimane analoga.
Guardando NumberLiteral vediamo che nei terminali c'è già il concetto di int che prende anche i negativi.



11 – LAB con DSL e XText

Xtext è un **framework** che copre tutti gli aspetti della infrastruttura linguistica di un DSL (internamente, usa ANTLR → **approccio LL***). Dalla specifica, **Xtext genera:**

- Lexer, parser, classi per il modello AST, l'AST specifico
- Editor con syntax highlighting e tutto quel che serve per una piena integrazione con l'ambiente Eclipse (o IDEA)
- Errori, content assist, quick fix, outline…

Da slide 5 a slide 24 del pacchetto 11 è spiegato il procedimento da seguire per poter installare ed utilizzare XText. In generale:

4. Creare un nuovo Progetto Xtext
5. Scrivere la grammatica del DSL
6. Generare gli Xtext artifacts
7. Lanciare il sistema appena creato
8. Creare un nuovo progetto del tipo dsl appena definito

Esempio con le espressioni slide 25-38.

Una volta è creata l'infrastruttura sintattica di base, bisogna fare **altri due passi:**

- **Validazione**
  - Specifica i vincoli semantici non mappati sulla grammatica
  - Sfruttata dall'IDE per generare error marker e quickfix
  - Opera continuamente in background, mentre l'utente scrive
- **Generazione**
  - Applica la semantica voluta alle frasi del DSL
  - Tipicamente generazione di codice, ma anche generazione di descrizioni, frasi, file…

Ad esempio i controlli di tipo vengono fatti in fase di valutazione. Noi non ne abbiamo mai parlato, ma sono controlli di tipo semantico. È una fase preliminare alla vera generazione del codice. Che poi la validazione e la generazione vengano fatte contemporaneamente, è una questione organizzativa.
Qui praticamente ci sono tutti i vincoli che non si possono esprimere nella grammatica ma che da qualche parte devono stare. Vengono generati anche i quick fix che appaiono durante la scrittura del codice.
Il validatore è quello che opera in background che fa apparire i tooltip e che ti fa apparire le "biscioline" quando scrivi delle cagate.

11.1 – ECLIPSE METAMODEL FRAMEWORK (EMF)

EMF specifica come governare la generazione di codice sulla base di un meta-modello, che può essere descritto in vari modi: Ecore (~UML) ma anche Xcore (scritto in Xtext stesso).

Xtext sfrutta EMF per generare l'AST della frase DSL: a partire dalla nostra grammatica, Xtext deduce il metamodello EMF del nostro linguaggio. **Tipicamente, per ogni regola grammaticale viene creata**, in src-gen, **una interfaccia Java con la corrispondente implementazione.**

Per usarlo occorre conoscere alcune **convenzioni**:

- Le classi sono sempre create con factory (no costruttori espliciti)
- Si accede a campi/proprietà con metodi getter/setter

- Collections implementate come EList (estende java.util.List)

Dall'AST si può generare ciò che si vuole: codice Java, file XML, altri formati…
L'AST è espresso in EMF: si potrebbe manipolarlo in Java, ma tipicamente si usa un linguaggio ad hoc che è Xtend: pensato per lavorare insieme a Xtext, ma usabile in generale. Ha molte idee Scala-like.

Java è molto tipizzato e siccome i tipi sono davanti, diventa difficile eliminarle. In tutti i linguaggi più nuovi invece, si usa il tipo del risultato in fondo. Scala e anche Xtend usano def per definire le funzioni, ma altri possono usare "fun" o "func"… Insomma, quelle sono le funzioni.
Ogni volta che si può dedurre il tipo di risultato, si mettono i due punti con il tipo in seguito. Quindi dove è semplice dedurre il tipo di ritorno, si può evitare di scriverlo.

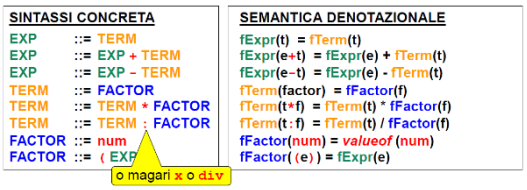
12 – LAB con ANTLR4, XText e XTend

Esempio di implementazione degli esempi di grammatica delle espressioni visti precedentemente.
Esempio di implementazione di “SmallJava”.

13 – LAB con MULTI-PARADIGM PROGRAMMING (Prolog)

La semantica denotazionale può essere tradotta pari pari in regole Prolog, ottenendo:

- Un puro riconoscitore, nel caso minimo
- Un interprete completo, con una semplice estensione



13.1 – RIPRENDIAMO IL PROLOG

Per i concetti principali di Prolog, vedi 3.1.

Gli argomenti della testa possono essere anche termini strutturati p(a, m(X,Y), B) :- …
In particolare, invece di m(X,Y) si possono usare strutture sintattiche con operatori infissi noti:

p(a, X+Y, B) :- …

Scrivere X+Y come argomento è esattamente come scrivere +(X,Y). Quindi se la m della regola precedente significa somma, le due cose sono equivalenti. Inoltre facendo così, il Prolog farà già il lavoro dello scanner perché sarà lui a trovare regole della forma X+Y e sarà lui a separare i token X e Y: ho la tokenizzazione già fatta.

Durante una query, le regole vengono selezionate tramite una forma evoluta di pattern matching detta **UNIFICAZIONE:** più regole con lo stesso funtore e ugual numero di argomenti, ma lista argomenti diversa, vengono selezionate in modo associativo in base agli argomenti presenti nella query (**PATTERN MATCHING**). Le variabili della query vengono fatte corrispondere a quelle della regola (UNIFICAZIONE delle variabili), sia in input che in output.

13.2 – RICONOSCITORE PROLOG

Abbiamo che **il motore Prolog contiene un suo scanner e un suo parser**, che usa per interpretare la sua sintassi di regole. Se riusciamo a mappare la nostra sintassi sulla sua, li riusiamo gratis!
3 su 4 degli operatori che ci interessano (+, -, *,) sono operatori infissi leciti anche per il Prolog, quindi possiamo riusarli.
Ci mancano:

- **L'operatore di divisione** che abbiamo scelto (:), che è diverso perché Prolog usa come operatore di divisione la classica barra (/)
- **Le espressioni usano parentesi tonde, ma Prolog le usa per la sua sintassi di top-level** → il suo parser le intercetta prima di noi ☹

13.2.1 – L'OPERATORE DI DIVISIONE

Per definire un nuovo operatore infisso in Prolog basta porre all'inizio della teoria logica la dichiarazione:

:- op(livelloprioritàà, associatività, operatore).

Dove:

- **livelloprioritàà** = numero che esprime la priorità dall'operatore (↑num ↓priorità)
- **associatività** = atomo che esprime se l'operatore è in/pre/postfisso, e se è associativo a sx/dx
- **operatore** = il nuovo operatore da dichiarare

Qui ha senso porre ":" allo stesso livello della moltiplicazione e dell'operatore di divisione "barra" già esistente, che è 500. **xfy** = "operatore infisso associativo a dx", **yfx** = "operatore infisso associativo a sx"

:- op(500,yfx,':').

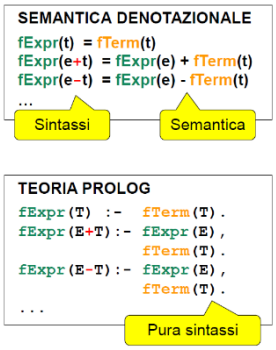
Così, Prolog accetterà di trovare scritti termini come 3:2, che verranno considerati scorciatoie per la notazione a funzione ':'(3,2).
In assenza di definizione esplicita di operatore infisso, avremmo potuto usare il ":" solo con tale notazione a funzione: scomodo! Ciò non significa aver dato a ":" il significato di divisione, né alcun altro significato! Abbiamo semplicemente autorizzato una notazione infissa in luogo di quella standard prefissa.
In definitiva: possiamo scrivere strutture sintattiche come 3+2, 14-2, X-Y+12, X*Y-3, e anche X:Y-3, 14:2, etc. come se l'operatore ":" fosse sempre esistito! 😊

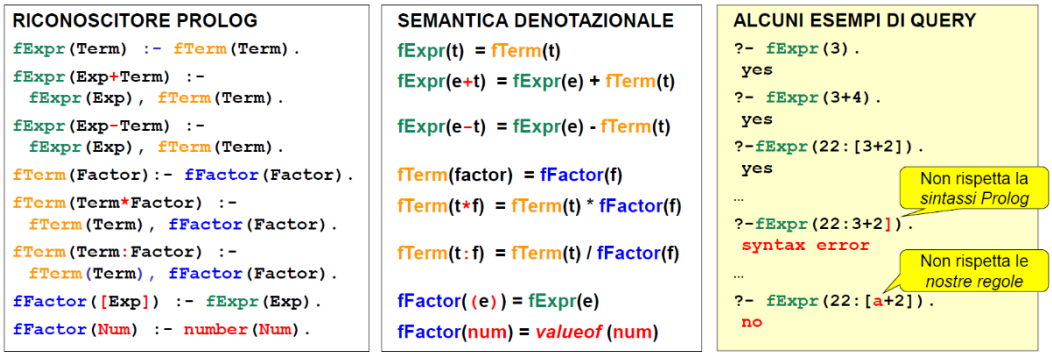
13.2.2 – LE PARENTESI TONDE

Le espressioni usano parentesi tonde, che però anche Prolog usa per la sua sintassi di top-level, per esprimere regole e termini, quindi non possiamo usarle perché il suo parser le intercetta subito.
Potremmo inventarci soluzioni complicate, ma non ne vale la pena! Per non complicarci la vita, le sostituiamo con le **parentesi quadre**, che Prolog conosce perché le usa per certi termini, come le liste.
In questo modo il parser Prolog le accetta e ne controlla già il bilanciamento. Che per lui siano liste è irrilevante: le usiamo come puro costrutto sintattico, l'importante è che il suo parser le lasci giungere a noi senza "mangiarsele".
Il prezzo è dover cambiare la sintassi per l'utente Prolog → anziché (3+4)*5 dovremo scrivere [3+4]*5.

Guarda caso, **la semantica denotazionale esprimeva già i vari casi con regole "pattern oriented"**. Se si riesce a mappare ogni regola della semantica in una regola Prolog, è (quasi) fatta!

- Gli operatori infissi della nostra grammatica sono operatori leciti per il parser Prolog → non servono mapping speciali, si possono riusare così come sono
- **Le regole Prolog non contengono semantica, sono pura sintassi,** quindi il significato delle operazioni dovrà essere espresso a parte





13.3 – VALUTATORE PROLOG

Per ottenere un valutatore occorre:

- **Estendere la testa delle regole** con un argomento per restituire il risultato, detto parametro di uscita
- **Estendere il corpo delle regole** in modo che
  - Ricavi i valori dei due argomenti
  - Li combini nel modo specificato dalla semantica

VALUTATORE PROLOG	SEMANTICA DENOTAZIONALE
evalExpr(Term, Value) :- evalTerm(Term, Value). evalExpr(Exp+Term, Value) :- evalExpr(Exp, Value1), evalTerm(Term, Value2), Value is Value1 + Value2. evalExpr(Exp-Term, Value) :- evalExpr(Exp, Value1), evalTerm(Term, Value2), Value is Value1 - Value2.	fExpr(t) = fTerm(t) fExpr(e+t) = fExpr(e) + fTerm(t) fExpr(e-t) = fExpr(e) - fTerm(t)
evalTerm(Factor, Value) :- evalFactor(Factor, Value). evalTerm(Term*Factor, Value) :- evalTerm(Term, Value1), evalFactor(Factor, Value2), Value is Value1 * Value2. evalTerm(Term:Factor, Value) :- evalTerm(Term, Value1), evalFactor(Factor, Value2), Value is Value1 / Value2.	fTerm(factor) = fFactor(f) fTerm(t*f) = fTerm(t) * fFactor(f) fTerm(t:f) = fTerm(t) / fFactor(f)
evalFactor([Exp], Value) :- evalExpr(Exp, Value). evalFactor(Num, Num) :- number(Num).	fFactor((e)) = fExpr(e) fFactor(num) = valueof (num)

Il sotto-problema del **controllo della sintassi dei numeri può essere delegato al parser Prolog** purché ci vadano bene la sua sintassi e semantica: **Prolog concepisce solo numeri reali** (non ha la nozione di numero intero), quindi per calcolare il valore di un'espressione, Prolog usa il predicato speciale "is" che applica agli argomenti la semantica dei numeri reali.

Il predicato **number** non svolge alcuna conversione atomo/numero: in effetti, la funzione di interpretazione valueof è implicita nel predicato standard is.

13.4 – tuProlog

Se volessimo un'applicazione completa, con Java potremmo gestire I/O grafico e le stringhe facilmente.
L'interprete però è molto più semplice scriverlo in Prolog perché abbiamo 8 regole contro decine e decine di linee di codice Java!
Ecco allora che possiamo sfruttare il **MULTI-PARADIGM SOFTWARE ENGINEERING**. Ha senso pensare a una applicazione multi-paradigma che adotti più paradigmi (e linguaggi) di programmazione contemporaneamente usandoli ciascuno per «ciò che sa fare meglio».
tuProlog (tuprolog.unibo.it) è un interprete Prolog scritto in Java, leggero, scalabile e interfacciabile con Java appunto per supportare facilmente la costruzione di applicazioni multi-paradigma.

Passi fondamentali:

1. **Creare un motore Prolog**: Prolog engine = new Prolog();
2. **Creare una teoria logica** (insieme di regole): Theory t = new Theory(…) dove il parametro è o una stringa che contiene la teoria, o un FileInputStream da cui leggerla
3. **Caricare la teoria logica nel motore**: engine.setTheory(teoria)

Le entità Prolog sono rappresentate da una tassonomia di classi:

- **Term (astratta)**: la radice della tassonomia dei termini
- **Struct, Var, Int, …**: sottoclassi concrete di Term

Per chiedere al motore Prolog di **risolvere una query** si usa: SolveInfo info = engine.solve(struttura-query);
La soluzione è incapsulata nell'**oggetto SolveInfo** restituito:

- **isSuccess**() è vero se la query ha almeno una soluzione
- **getSolution**() recupera la soluzione trovata (un Term)

- **getTerm(String varName)** recupera il valore, sotto forma di oggetto Term, di una delle variabili della query iniziale

In una possibile architettura per un parser ibrido, **Java si occupa di:**

- Visualizzare una finestra di dialogo per chiedere all'utente la stringa di input (con parentesi TONDE, così risolviamo anche questa)
- Sostituire le parentesi tonde con le parentesi QUADRE richieste dal valutatore Prolog (che quindi rimangono nascoste sotto)
- Creare un motore Prolog, configurarlo con l'opportuna teoria, e interrogarlo sottoponendogli l'espressione
- Recuperare il risultato della valutazione e visualizzarlo all'utente

tuProlog invece si occupa di interpretare e valutare l'espressione data e calcolare il risultato in accordo alla semantica denotazionale specificata. **L'interprete Prolog è in un file di testo, quindi si può cambiare senza neanche ricompilare la parte Java!**

Esempio di codice slide 31-35 pacchetto 13.

Esempio di generalizzazione con i numeri romani slide 36-45

Esempio di calcolo delle derivate slide 46-52

14 – RICONOSCITORI LR

14.1 – LR vs LL

Il problema dell'analisi LL è che è troppo semplice: costruendo l'albero **top-down**, deve poter identificare la produzione giusta da usare avendo visto solo i primi k simboli della parte destra. In pratica parte dal simbolo iniziale S e poi scende cercando di coprire la frase.
LL(1) è quasi l'unico caso interessante perché LL(0) è puramente astratto, quindi inutile nella pratica, e LL(2) è utile solo in certe situazioni, come nell'assegnamento.

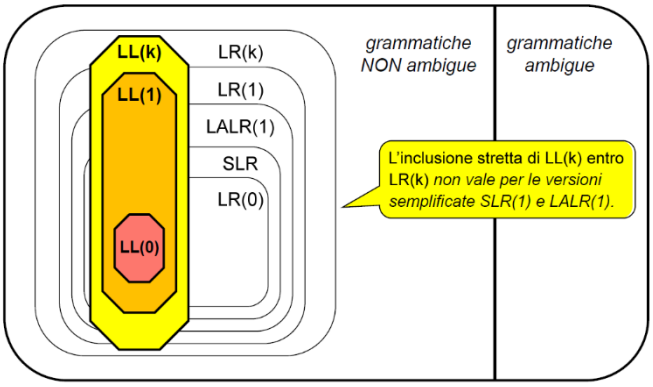
L'analisi LR invece costruisce l'albero **bottom-up**: partendo dalle foglie, aspetta di sapere abbastanza per decidere dove metterle. Parte quindi dalla frase da riconoscere e cerca di ridurla allo scopo S. Ad ogni passo deve quindi decidere, grazie alle **informazioni di contesto**, se:

- Proseguire la lettura da input, senza fare altro (**SHIFT**)
- Costruire un pezzo di albero senza leggere da input (**REDUCE**)

È meno naturale dell'analisi LL, ma è **superiore dal punto di vista teorico** perché il **parser LR è la macchina più potente per le grammatiche di tipo 2**. Infatti, **ogni grammatica LL(K) è anche LR(K)**.
Questa analisi è molto **complessa**: anche il caso più semplice LR(1) spesso è ingestibile. Per questo esistono delle **TECNICHE SEMPLIFICATE** che riducono la complessità:

- **SLR** (Simple LR)
- **LALR(1)** (Look-Ahead LR)

Si usano comunque sempre strumenti automatici.
La famiglia del parsing LR è composta come in figura.



14.2 – ARCHITETTURA PARSER LR

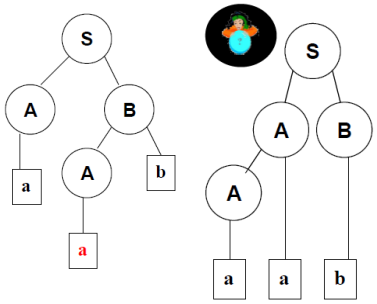
Il parser LR richiede concettualmente la presenza al suo interno di un ORACOLO: un componente che, in base al contesto corrente, gli dica in ogni istante se proseguire con la lettura da input o costruire un pezzo di albero senza leggere nulla.

Un parser LR è costituito da:

- Un **oracolo** che gli dice se fare SHIFT o REDUCE
- Uno **stack** in cui conservare l'input e l'albero
- Un **controller** orchestratore che governa il tutto

La **sequenza di riduzioni** non è casuale, ma è una **DERIVAZIONE CANONICA DESTRA USATA A ROVESCIO**, per risalire dalla frase allo scopo.

14.2.1 – DA TOP-DOWN A BOTTOM-UP



In un'analisi top-down la frase "aab" potrebbe essere analizzata secondo una certa grammatica GL secondo il processo rappresentato dall'albero in figura. L'albero viene costruire top-down, partendo dalla radice (lo scopo S). Ciò corrisponde ad applicare una derivazione canonica sinistra: S → AB → aB → aAb → aab

In un'analisi bottom-up, invece, la frase "aab" sarebbe invece riconosciuta a rovescio usando la derivazione canonica destra al contrario: aab → Aab → Ab → AB → S. L'albero viene quindi costruire bottom-up, partendo dalle foglie e risalendo.

14.2.2 – INFORMAZIONE DI CONTESTO

L'oracolo sfrutta delle informazioni di contesto per decidere con cognizione di causa se fare SHIFT o REDUCE. Esso è infatti un **RICONOSCITORE DI CONTESTI**: calcola la informazioni di contesto di ogni produzione e quando il contesto attuale è un contesto di riduzione, ordina la REDUCE giusta per costruire a colpo sicuro quel certo pezzo di albero.

Esempio
Grammatica G ricorsiva a sx: S → aAb | aaBba A→ Aa | b B → ε
La ricorsione sinistra non è un problema per l'analisi LR. Poiché il simbolo b compare nella parte destra di varie produzioni, sapere che b compare nella stringa data non basta per decidere quale riduzione applicare fra A → b, S → aAb o S → aaBba, perché appunto dipende dal contesto.
Se la stringa di input è "abb", l'azione giusta per la prima occorrenza di b è la riduzione A → b in modo da ottenere la forma di frase aAb da cui si può proseguire, ma per le occorrenze successive l'azione giusta sarebbe diversa.
Si dice allora che la riduzione A → b è adatta alla stringa di input nel contesto ab, ossia se la b da ridurre è preceduta dal prefisso a.

14.3 – ANALISI LR(0)

Nell'analisi LL siamo partiti dal caso LL(1) in quanto LL(0) è iper banale e non lo abbiamo considerato: LL(0) significa poter scegliere la mossa giusta da fare senza dover mai guardare il prossimo simbolo di input e questo è possibile solo se non ci sono mai alternative nelle regole.
Nell'analisi LR conviene invece studiare prima LR(0) perché, anche se **non si deve mai guardare il prossimo simbolo di input**, le **informazioni di contesto servono alla macchina per sapere qualcosa sul passato**.

Passi da fare per l'analisi LR(0):

- Calcolare il contesto LR(0) di ciascuna produzione
- Controllare collisioni fra contesti di produzioni diverse dove **COLLISIONE** = **stringa di un contesto è un PREFISSO PROPRIO di una stringa di un altro** (è prefisso di un'altra e ciò che segue è un terminale)
    - Se ci sono collisioni, i contesti LR(0) non bastano per avere un parser deterministico
    - **Se non ci sono collisioni, la grammatica è LR(0)**

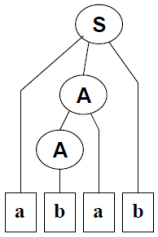
Esempio
La grammatica G ricorsiva a sx di prima è LR(0):
S → aAb → contest LR(0): {aAb} S → aaBba → contest LR(0): {aaBba}

A→ Aa → contest LR(0): {aAa} A→ b → contest LR(0): {ab}
B → ε → contest LR(0): {aa}
Non c'è collisione fra 2° e 5° contesto perché "aa", pur essendo prefisso di "aaBba", non è un prefisso proprio. Infatti ciò che segue "aa" è "B", che è un non-terminale.

Questo non è per niente un linguaggio semplice perché ha una ricorsione sinistra quindi è infinito, non accetta solo due parole e ciao. Ragionando bottom-up si ha molta più potenza: ecco perché LR è più potente!

La frase "**abab**" è analizzata come segue:

- Si legge "a" (SHIFT) ma ciò non identifica un contesto. L'oracolo ordina di proseguire
- Si legge "b" (SHIFT). "ab" fa parte del contesto A → b, quindi l'oracolo ordina di ridurre b ad A
- La forma di frase "aA" non fa parte di nessun contesto. L'oracolo ordina SHIFT
- Si legge "a", ottenendo "aAa" che è nel contesto A → Aa: l'oracolo ordina di ridurre Aa ad A
- La forma di frase è ancora "aA", quindi SHIFT
- Si legge "b", la forma "aAb" si riduce ad S → REDUCE

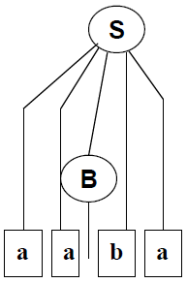


Stack	Input	Action
0	abab$	Shift
0 1	bab$	Shift
0 1 3	ab$	Reduce A->b
0 1 2	ab$	Shift
0 1 2 5	b$	Reduce A->Aa
0 1 2	b$	Shift
0 1 2 4	$	Accept

14.3.1 – UN CASO CRITICO

I calcoli connessi alle tecniche LR possono dare una informazione fuorviante **se S è riusato nella parte destra di qualche regola**. Per ovviare, bisogna introdurre una nuova produzione di top level Z → S con **Z nuovo scopo**.

Esempio
Con la stessa grammatica G ricorsiva a sx di prima:
S → aAb → contest LR(0): {aAb} S → aaBba → contest LR(0): {aaBba}
A→ Aa → contest LR(0): {aAa} A→ b → contest LR(0): {ab}
B → ε → contest LR(0): {aa}
Se analizzo la frase "aaba":



- SHIFT: leggo "a", nessun contesto
- SHIFT: leggo "a", ottengo "aa", quindi sia contesto B → ε sia contesto S → aaBba. Siccome dopo "aa" c'è B che è non terminale, non vi è collissione
- REDUCE: ε si riduce a B e ottengo stringa "aaB"
- SHIFT: leggo "b", ottengo "aaBb", nessun contesto
- SHIFT: leggo "a", ottengo "aaBba", contesto S → aaBba
- REDUCE a S

Esempio
S → Sa ha contesto LR(0) = { Sa } S → a ha contesto LR(0) = { a }
I contesti sono diversi, ma siccome S può essere anche in forme di frase intermedie, ridursi a S è ambiguo perché potremmo non aver finito la frase. Ecco che con la produzione di top level Z→S si risolve l'ambiguità.
Se voglio riconoscere la frase "aaa":

- Leggo "a", riduco a ad S
- Poi cosa faccio? Leggo di nuovo o basta?

Se non ci sono altre a, ci si è già ridotti allo scopo S, se invece la stringa continua con ulteriori a, occorre applicare ulteriori riduzioni, ma per sapere se la stringa è finita occorre guardare avanti di un simbolo e quindi la grammatica non è più LR(0).
Riformulo la grammatica con: Z→S = {S} S → Sa = {Sa} S → a = {a}
I primi due contesti collidono e adesso si vede chiaramente che S è un prefisso proprio di Sa.

14.3.2 – CALCOLO DEI CONTESTI LR(0)

Proprietà essenziale:
la grammatica che *definisce* i contesti LR(0) è sempre REGOLARE (a sinistra).

Il riconoscimento del contesto corrente, data la proprietà essenziale, può essere svolto da un automa a stati finiti, quindi **il nostro potentissimo oracolo è semplicemente un ASF!**

Per ottenere il riconoscitore di contesti ci sono due approcci:

- **APPROCCIO 1:** Definire in modo formale cosa sia il contesto, calcolarlo per ogni produzione e progettare il RSF che riconosca le frasi di tali contesti
- **APPROCCIO 2:** **Definire in modo formale cosa sia il contesto, ma poi applicare un procedimento operativo pratica per costruirli iterativamente evitando la parte di progettazione dell'RSF**. In questo caso occorre la produzione di top-level Z→ S

Riassumendo: per ogni token che mi arriva, si chiede all'oracolo cosa si deve fare con quel simbolo: trasformarlo in un metasimbolo oppure aspettare. L'oracolo quindi riconosce in quale contesto si trova.
L'oracolo magico non fa altro che guardare in ogni momento se la situazione in cui ci troviamo è riducibile ad una situazione nota oppure no: nel caso sia no, allora bisogna aspettare ad avere più informazioni.

DEFINIZIONE CONTESTI LR(0)

Il contesto LR(0) di una produzione A → α è:

$$\textbf{LR(0)ctx(A}\rightarrow \alpha\textbf{)} = \{\gamma \mid \gamma = \beta\alpha, \quad Z \overset{*}{\Rightarrow} \beta A w \Rightarrow \beta\alpha w, \quad w \in VT^* \}$$

In questo caso ci serve determinare β, mentre α è noto → Il Contesto Sinistro è quindi solo quello che riguarda β, è la sua fetta di sinistra, sono i prefissi. Appiccicando α alla fine, avremo il Contesto Complessivo.

Il contesto LR(0) della produzione A → α è l'insieme di tutti i prefissi $\beta\alpha$ di una forma di frase che usa la produzione all'ultimo passo (quindi $\beta A w \Rightarrow \beta\alpha w$) di una derivazione canonica destra.
Di conseguenza, tutte le stringhe del contesto LR(0) della produzione A → α hanno la forma $\beta\alpha$ e differiscono solo per il prefisso β, che dipende solo da A.

Contesto sinistro di un non-terminale A:
$$\text{leftctx(A)} = \{ \beta \mid Z \overset{*}{\Rightarrow} \beta A w,\ w \in VT^* \}$$
da cui il contesto LR(0) della produzione $A \rightarrow \alpha$:
$$\text{LR(0)ctx}(A \rightarrow \alpha) = \text{leftctx(A)} \bullet \{\alpha\}$$

Di conseguenza, si può esprimere il contesto LR(0) come concatenazione fra l'insieme dei β (Contesto sinistro di A) e il suffisso α.
Esiste un algoritmo che sfrutta questa definizione per calcolare costruttivamente i contesti LR(0).

In pratica, tutte le volte in cui mi troverò il simbolo A, guarderò cosa c'era davanti a lui: ciò che ho visto prima è proprio l'informazione di contesto che vado cercando. **Cerco quindi tutti i prefissi**: ci possono essere situazioni in cui A può essere riscritto come γ.

CALCOLO DEI CONTESTI SINISTRI

Determinare leftctx(A) implica investigare tutti i modi in cui può apparire il metasimbolo A in una forma di frase. A tale scopo si definiscono i seguenti due **POSTULATI**:

- Poiché lo scopo Z per definizione non compare mai nella parte destra di alcuna produzione, **leftctx(Z) = {ε}**.
- Data una produzione B → $\gamma A \delta$, i prefissi che possono esserci davanti ad A sono quelli che potevano esserci davanti a B seguiti dalla stringa γ. (**B → $\gamma A \delta$ ⇒ leftctx(A) ⊇ leftctx(B)·{γ}**)

Quindi, un primo contributo a leftctx(A) è leftctx(B)·{γ}. Si parla di primo contributo perché altri contributi potranno emergere analizzando le altre regole di produzione.
Nel complesso, leftctx(A) si ottiene perciò unendo tutti i contributi di tutte le produzioni in cui A compaia nella parte destra.

Con B → $\gamma A \delta$ è come se potessi dividere la mia frase in: (qualcosa) B = (qualcosa) γ A δ in cui il "qualcosa" di sinistra è il contesto sinistro di B e l'altro "qualcosa" è il contesto di destra di A.
L'unica cosa che so in questo caso è che il contesto di sinistra di A è più grande del contesto di sinistra di B.

Esempio
Calcoliamo i contesti di sinistra di questa grammatica mettendo in pratica i due postulati:

- leftctx(Z) = {ε}
- B → $\gamma A \delta$ ⇒ leftctx(A) ⊇ leftctx(B)·{γ} (A contenuto in B)

$Z \rightarrow S$
$S \rightarrow a\,S\,A\,B \mid B\,A$
$A \rightarrow a\,A \mid B$
$B \rightarrow b$

Regola 1) Z → S:
- **leftctx(Z) = {ε}**
- leftctx(S) ⊇ leftctx(Z) · {ε} = leftctx(Z)

Regola 2): S → aSAB | BA:
- leftctx(S) ⊇ leftctx(S) · {a} dove S → aSδ
- leftctx(A) ⊇ leftctx(S) · {aS} dove S → aSAδ
- leftctx(B) ⊇ leftctx(S) · {aSA} dove S → aSAB
- leftctx(B) ⊇ leftctx(S) · {ε} dove S → Bδ
- leftctx(A) ⊇ leftctx(S) · {B} dove S → BA

Regola 3) A → aA | B:
- leftctx(A) ⊇ leftctx(A) · {a}
- leftctx(B) ⊇ leftctx(A) · {ε}

leftctx(S) = leftctx(Z) · ~~{ε}~~ ∪ leftctx(S) · {a}
leftctx(A) = leftctx(S) · {aS} ∪
leftctx(S) · {B} ∪
leftctx(A) · {a}
leftctx(B) = leftctx(S) · {aSA} ∪
leftctx(S) · ~~{ε}~~ ∪
leftctx(A) · ~~{ε}~~

Possiamo adesso passare dai contesti alla grammatica e ci accorgeremo che abbiamo una grammatica regolare sinistra riconoscibile da una macchina a stati deterministica:

leftctx(Z) = {ε} <LctxZ> → ε
leftctx(S) = leftctx(Z) ∪ leftctx(S) · {a} <LctxS> → <LctxZ> | <LctxS> **a**
leftctx(A) = leftctx(S) · {aS} ∪ leftctx(S) · {B} ∪ leftctx(A) · {a} <LctxA> → <LctxS>**aS** | <LctxS>**B** | <LctxA>**a**
leftctx(B) = leftctx(S) · {aSA} ∪ leftctx(S) ∪ leftctx(A) <LctxB> → <LctxS>**aSA** | <LctxS> | <LctxA>

I simboli, terminali e non, della grammatica originale sono allora i simboli terminali di questa grammatica.

Dalla grammatica possiamo poi ottenere anche le espressioni regolari facilmente:

<LctxZ> → ε leftctx(Z) = {ε}
<LctxS> → <LctxZ> | <LctxS> **a** leftctx(S) = {a*}
<LctxA> → <LctxS>**aS** | <LctxS>**B** | <LctxA>**a** leftctx(A) = { (a*aS + a*B) a* }
<LctxB> → <LctxS>**aSA** | <LctxS> | <LctxA> leftctx(B) = { a*aSA + a* + (a*aS + a*B) a*}

Concatenando poi i contesti sinistri con i rispettivi suffissi, passando per le espressioni regolari semplificate, otteniamo i contesti LR(0):

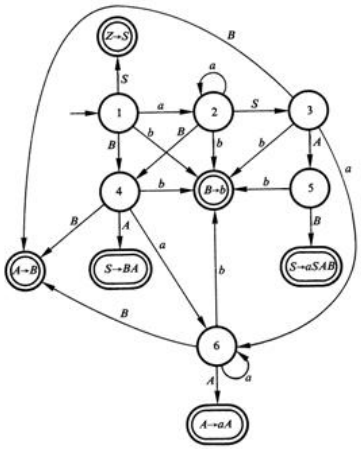
leftctx(Z) = {ε} LR(0)ctx(Z→S)= {ε}S = {S}
leftctx(S) = {a*} LR(0)ctx(S→aSAB) = {a*aSAB}
leftctx(A) = { (a*aS + a*B) a* } LR(0)ctx(S→BA) = {a*BA}
leftctx(B) = { a*aSA + a* + (a*aS + a*B) a*} LR(0)ctx(A→aA) = {a* (aS+B) a* aA}
LR(0)ctx(A→B) = {a* (aS+B) a* B}
LR(0)ctx(B→b) = {a* (aSA + ε) b + a* (aS + B) a*b}

Ora si costruire l'RSF che riconosce l'unione di questi linguaggi e se è deterministico, il gioco è fatto.

Questo RSF è la macchina caratteristica della grammatica iniziale e ogni stato finale è etichettato con una produzione. Si riparte ogni volta dallo stato iniziale e si percorre l'ASF per sapere che mossa fare.
Questa volta è andata bene ed è deterministico perché è un esempio costruito a tavolino, ma normalmente non va così bene.
Vediamo ora il parsing LR(0) all'opera:

- **SHIFT**: automa caratteristico raggiunge uno <u>stato non finale</u>, quindi non possiamo ancora decidere
- **REDUCE**: automa caratteristico raggiunge uno <u>stato finale</u>, quindi applico la produzione e riduco

Dopo ogni riduzione, l'automa ricomincia da capo.



Come dicevamo, la fatica più grossa è progettare la macchina, ma poi non è niente di impossibile perché abbiamo uno stack, un controller che fa le domande e un oracolo che dà le risposte.
È una macchina che va veloce, è efficiente ed è interessante perché ha sempre una risposta e fa benissimo il suo lavoro: mangia anche la ricorsione sinistra.
Il problema è che la macchina la devi progettare a mano: questo è il tipico comportamento meccanico che va bene da far fare ad una macchina perché farlo a mano è noioso e lungo.

La forma di frase corrente è mantenuta su uno stack, che è normale perché stiamo parlando di un PDA:

- **SHIFT** → mette in cima allo stack il nuovo simbolo terminale letto
- **REDUCE** → toglie tanti elementi quanti corrispondono alla parte destra della riduzione da applicare, poi mette in cima il simbolo non terminale corrispondente alla parte sinistra di tale riduzione

Parsing bottom-up di "**abbabb**":

- simbolo: a stato: 1 stato futuro: 2
  simbolo: b stato: 2 stato <u>finale</u>: B → b
  riduzione: abbabb → aBbabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: b stato: 4 stato <u>finale</u>: B → b
  riduzione: aBbabb → aBBabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: B stato: 4 stato <u>finale</u>: A → B
  riduzione: aBBabb → aBAabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: A stato: 4 stato <u>finale</u>: S → BA
  riduzione: aBAabb → aSabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: b stato: 6 stato <u>finale</u>: B → b
  riduzione: aSabb → aSaBb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: B stato: 6 stato <u>finale</u>: A → B
  riduzione: aSaBb → aSaAb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: A stato: 6 stato <u>finale</u>: A → aA
  riduzione: aSaAb → aSAb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: A stato: 3 stato futuro: 5
  simbolo: b stato: 5 stato <u>finale</u>: B → b
  riduzione: aSAb → aSAB
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: A stato: 3 stato futuro: 5
  simbolo: B stato: 5 stato <u>finale</u>: S → aSAB
  riduzione: aSAB → S
- simbolo: S stato: 1 stato <u>finale</u>: Z → S
  riduzione: S → Z

Frase ridotta allo scopo: OK!

Ho "ab", sostituisco b con B. Con aB cosa posso fare? Nulla, quindi continuo.
Siccome ho "aBB", trasformo A → B. Ottengo "aBA", trasformo con S e ottengo "aS". Cosa posso fare con aS? Niente, allora proseguo.

Guardando l'evoluzione dello stack:
Molto spesso io cambio solo l'ultima lettera nello stack, quindi i passaggi precedenti sono uguali a prima: questo mi fa venire un'idea.

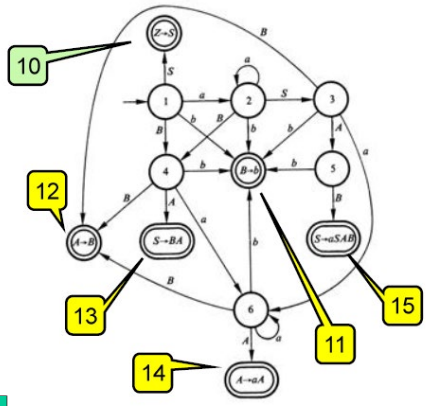
							b	B	A		b	B		
			b	B	A		a	a	a	A	A	A		
	b	B	B	B	B	S	S	S	S	S	S	S		
a	a	a	a	a	a	a	a	a	a	a	a	a	S	Z

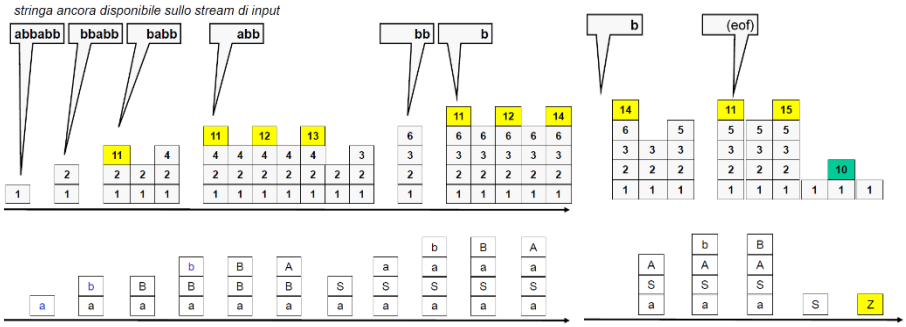
Se quell'oracolo è fatto di centinaia di stati e per arrivare ad una riduzione sono passato da 96 stati, non è molto smart rifarlo tante volte: allora ci mettiamo una **cache che ha la forma di una <u>stack ausiliario</u>**.
In questo modo mi segno in che stato sono, così la volta dopo non riparto dallo stato 1, ma riparto dagli stati precedenti.

Lo stesso esempio con un parsing ottimizzato diventa quindi:

- stack degli stati: [1] (stato iniziale)
  simbolo: a stato: 1 stato futuro: 2
  simbolo: b stato: 2 stato finale: B → b
  stack degli stati: [1, 2, B→b]
  riduzione: abbabb → aBbabb
  stack degli stati: [1, 2] (top: 2)
- simbolo: B stato: 2 stato futuro: 4
  simbolo: b stato: 4 stato finale: B → b
  stack degli stati: [1, 2, 4, B→b]
  riduzione: aBbabb → aBBabb
  stack degli stati: [1, 2, 4] (top: 4)
- simbolo: B stato: 4 stato finale: A → B
  stack degli stati: [1, 2, 4, A→B]
  riduzione: aBBabb → aBAabb
  stack degli stati: [1, 2, 4] (top: 4)
- stack degli stati: [1, 2,4] (top: 4)
  simbolo: A stato: 4 stato finale: S → BA
  stack degli stati: [1, 2, 4, S→BA]
  riduzione: aBAabb → aSabb
  stack degli stati: [1, 2] (top: 2)
- simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: b stato: 6 stato finale: B → b
  stack degli stati: [1, 2, 3, 6, B→b]
  riduzione: aSabb → aSaBb
  stack degli stati: [1, 2, 3, 6] (top: 6)
- simbolo: B stato: 6 stato finale: A → B
  stack degli stati: [1, 2, 3, 6, A→B]
  riduzione: aSaBb → aSaAb
  stack degli stati: [1, 2, 3, 6] (top: 6)
- stack degli stati: [1, 2, 3, 6] (top: 6)
  simbolo: A stato: 6 stato finale: A → aA
  stack degli stati: [1, 2, 3, 6, A→aA]
  riduzione: aSaAb → aSAb
  stack degli stati: [1, 2, 3] (top: 3)
- simbolo: A stato: 3 stato futuro: 5
  simbolo: b stato: 5 stato finale: B → b
  stack degli stati: [1, 2, 3, 5, B→b]
  riduzione: aSAb → aSAB
  stack degli stati: [1, 2, 3, 5] (top: 5)
- simbolo: B stato: 5 stato finale: S → aSAB
  stack degli stati: [1, 2, 3, 5, S→aSAB]
  riduzione: aSAB → S
  stack degli stati: [1] (top: 1)
- simbolo: S stato: 1 stato finale: Z → S
  stack degli stati: [1, Z→S]
  riduzione: S → Z
  stack degli stati: [1] (situaz. iniziale: OK)

All'inizio lo stack degli stati contiene solo lo stato iniziale 1, poi, **dopo ogni lettura dell'input**, si pone sullo stack degli stati lo stato corrispondente alla transizione e si pone sullo stack di input il simbolo appena letto. **Ad ogni passo di riduzione**, invece, si estraggono dallo stack degli stati tanti stati quanti i simboli coinvolti nella parte destra della riduzione applicata e si pone sullo stack lo stato risultante della riduzione. Allo stesso tempo si estraggono dallo stack di input i simboli coinvolti sempre nella parte destra della riduzione applicata e si pone su tale stack il metasimbolo risultante dalla riduzione.





14.3.3 – CONDIZIONE LR(0)

Condizione *sufficiente* perché una grammatica sia LR(0) è che, date le due produzioni A → α e B → ω

- **se** θ ∈ LR(0)ctx(A → α) θ ∈ (VT ∪ VN)*
- **e** θw ∈ LR(0)ctx(B → ω) w ∈ VT*

risulti:

w = ε, A = B, α = ω

Una grammatica è analizzabile mediante analisi LR(0) con la condizione sufficiente in figura. A parole, **ogni stato di riduzione dell'automa ausiliario deve essere etichettato da una produzione unica e non deve avere archi di uscita etichettati da terminali.**
Supponiamo di avere θ a cui equivalgono due contesti: allora ho due scelte quindi non posso mai avere una macchina deterministica. L'unico caso in cui ci cavo i piedi è quando i due contesti sono gli stessi.

Guardando l'automa dell'esempio precedente, esso è LR(0) perché verifica la condizione sufficiente LR(0) in quanto nello stato finale non ci sono archi uscenti. Se ci fossero, dovrebbero essere etichettati da un meta simbolo.

Teorema: una grammatica LR(0) non è mai ambigua.

14.3.4 – PROCEDIMENTO OPERATIVO

Il procedimento visto fino ad ora per costruire l'automa caratteristico è stato lungo e complesso (era l'approccio 1): serve un procedimento più snello. Infatti, l'automa può essere ottenuto senza calcolare esplicitamente né i contesti sinistri né i contesti LR(0), ma semplicemente usando un procedimento operativo meccanizzabile (approccio 2). **In questo secondo approccio:**

- **Non calcoliamo i contesti** per sintetizzare l'automa

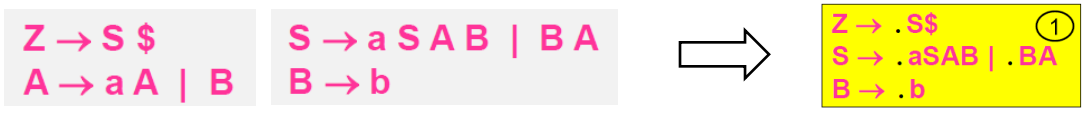
- **Partiamo dalla regola di top level Z → S$ e analizziamo via via le situazioni che si presentano**
  - Scriviamo ogni situazione in un diverso rettangolo detto LR TERM
  - Quando una regola ne usa un'altra, la includiamo nel rettangolo
  - Introduciamo l'astrazione cursore "." per indicare dove siamo all'interno di ogni regola (all'inizio saremo davanti e scriveremo Z → .S$)
- **Studiamo le evoluzioni possibili di ogni rettangolo e costruiamo direttamente l'automa dal basso, caso per caso**

Esempio

Alcune convenzioni:

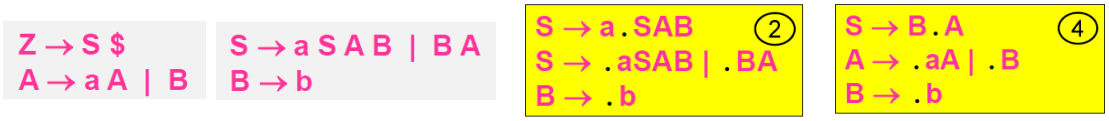
- Ogni frase lecita è esplicitamente terminata dal terminatore $, quindi la produzione di top level aggiunta è Z→S$
- Data una forma di frase, il cursore "." denota il confine tra la parte già analizzata a sinistra e quella ancora da analizzare a destra (Es. A → a.B significa che il prossimo input è B)
- Input = forma di frase eventualmente già sottoposta a passi di riduzione costituita sia da simboli terminali che non
- L'automa si appoggia ad uno stack di input in cui i simboli a sinistra del cursore sono già stati estratti, mentre quelli a destra no
- Prossimo input = simbolo al top dello stack

All'inizio tutto l'input è ancora da leggere e ogni frase lecita deriva dallo scopo Z, quindi dalla grammatica solita, otteniamo la prima regola del riquadro giallo. Le altre regole sono ottenute perché per descrivere compiutamente il primo stato occorre anche aggiungere tutte le produzioni di S (seconda regola) e tutte le produzioni di B (terza regola) – perché una delle produzioni di S può iniziare per B.



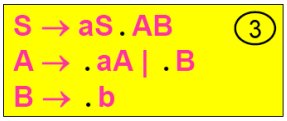
Abbiamo ottenuto allora lo stato 1, il primo stato dell'automa caratteristico che rappresenta tutte le situazioni iniziali possibili. Ora dobbiamo capire le possibili evoluzioni spostando il cursore a destra di una posizione in tutti i modi possibili:

- **Prossimo input = S** con Z → S.$ cioè stato finale **(F)**
- **Prossimo input = b** con B → b cioè stato finale di riduzione **(R1)**
- **Prossimo input = a** con S → a.SAB **(2)**
  - Input deve iniziare per S → considero produzioni di S
  - Una produzione di S comincia per B → considero produzioni B
- **Prossimo input = B** con S → B.A **(4)**
  - Input inizia per A → considero produzioni A
  - Una produzione di A può iniziare per B → considero produzioni B



Ora dallo stato **(2)** vediamo le possibili evoluzioni:

- Prossimo input = S con S → aS.AB **(3)**
  - Input può iniziare per A → produzioni di A
  - Una produzione di A comincia per B → produzioni di B
- Prossimo input = a con S → a.SAB **(2)** → auto anello
- Prossimo input = B con S → B.A **(4)**
- Prossimo input = b con B → b. **(R1)**



Dallo stato **(4)** invece le evoluzioni sono:

- Prossimo input = A con S → BA. **(R3)**
- Prossimo input = a con A → a.A **(6)**
  - Input può iniziare per A → produzioni di A
  - Una produzione di A comincia per B → produzioni di B
- Prossimo input = B con A → B. **(R2)**
- Prossimo input = b con B → b. **(R1)**

$A \rightarrow a\,.\,A$ (6)
$A \rightarrow\ .\,aA \mid .\,B$
$B \rightarrow\ .\,b$

Ora, dallo stato (3) le evoluzioni sono:

- Prossimo input = A con S → aSA.B **(5)**
  - Input può iniziare con B → produzioni B
- Prossimo input = a con A → a.A **(6)**
- Prossimo input = B con A → B. **(R2)**
- Prossimo input = b con B → b. **(R1)**

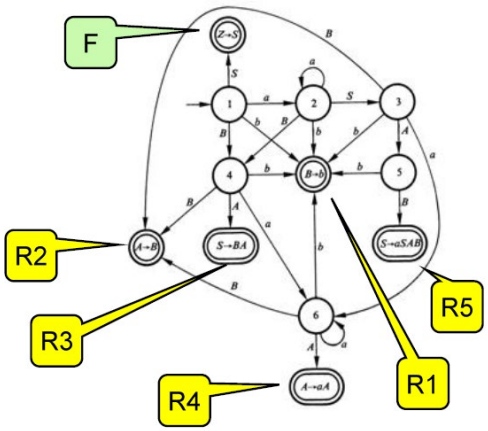
$S \rightarrow aSA\,.\,B$ (5)
$B \rightarrow\ .\,b$

Dallo stato (6) le evoluzioni sono:

- Prossimo input = A con A → aA. **(R4)**
- Prossimo input = a con A → a.A **(6)** → auto anello
- Prossimo input = B con A → B. **(R2)**
- Prossimo input = b con B → b. **(R1)**

Dallo stato (5) le evoluziono sono:

- Prossimo input = B con S → aSAB. **(R5)**
- Prossimo input = b con B → b. **(R1)**



14.3.5 – PARSER LR(0)

Tecnicamente, le mosse sono 4 e non 2:

- Ogni arco corrisponde ad un'azione SHIFT
  - **SHIFT** è l'arco corrispondente ad un terminale → vera lettura da input
  - Se l'arco corrisponde ad un metasimbolo, azione è **GOTO** → lettura interna dallo stack
- Ogni stato finale corrisponde ad un'azione **REDUCE**
  - Detta **ACCEPT** nel caso finale che riduce allo scopo e completa l'albero

Si costruisce allora la **tabella di parsing LR** in cui in ogni cella c'è l'indicazione dell'azione da compiere + altre indicazioni accessorie. Di solito si usa la **notazione**:

- s/5 = shift (leggi input) e vai nello stato 5
- g/5 = goto stato 5 e consuma un metasimbolo
- r/4 = riduci usando la 4° produzione e termina
- a = accetta e termina

Omettendo le situazioni di errore, la tabella di parsing sarà **sparsa**, cioè alcune posizioni saranno vuote.

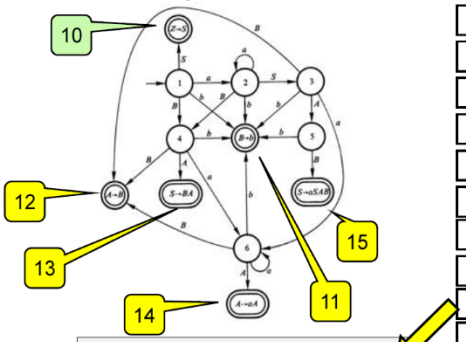
Algoritmo di costruzione della *tabella di parsing*

- per ogni arco S1 → S2 con input il simbolo terminale *a*, si inserisce in tabella alla posizione (S1, *a*) l'azione *shift to S2*
- per ogni arco S1 → S2 con "input" (da stack) il metasimbolo *X*, si inserisce in tabella alla posizione (S1, *X*) l'azione *goto S2*
- per ogni stato S_i associato alla regola R-esima, $A \rightarrow \alpha\,.$ si inserisce in tabella l'azione *reduce R* in tutta la riga corrispondente allo stato S_i
- per ogni stato S_i contenente la produzione $Z \rightarrow S\,.\,\$$ si inserisce in tabella alla posizione (S_i , \$) l'azione *accept*

Esempio

Un parser LR(0) deve decidere se fare shift o reduce guardando solo lo stato attuale. Solo se ha deciso di shiftare, sfrutta il simbolo letto per decidere lo stato futuro.
In corrispondenza di uno stato di riduzione, l'azione è per forza la stessa in tutte le colonne dei simboli terminali perché l'input **non** è stato letto!

L'automa con gli stati rinumerati:



$B \to b$ (1) $A \to B$ (2)
$S \to B\,A$ (3) $A \to a\,A$ (4)
$S \to a\,S\,A\,B$ (5)

	a	b	$	S	A	B
1	s2	s11		g10		g4
2	s2	s11		g3		g4
3	s6	s11			g5	g12
4	s6	s11			g13	g12
5		s11				g15
6	s6				g14	g12
10			a			
11	r1	r1	r1			
12	r2	r2	r2			
13	r3	r3	r3			
14	r4	r4	r4			
15	r5	r5	r5			

Notiamo che qui è tutto perfetto: nelle grammatiche reali potrebbero esserci più scelte. si potrebbe ridurre con due regole diverse oppure si potrebbe ridurre o shiftare: c'è spesso conflitto. Nella realtà c'è solo una cura: passare ad LR(1).

LR(0): sono davanti ad una porta e non so cosa c'è dopo, potrebbe esserci quello che voglio o potrebbe esserci una tigre: se c'è la tigre, sfiga. **LR(1):** sono davanti ad una porta ma posso guardare dallo spioncino per vedere se dopo c'è quello che voglio o la tigre.

Contro-esempio

$Z \to E$ $E \to T+E \mid T$ $T \to a$

Se consideriamo la grammatica a destra, notiamo che ha la ricorsione destra. Non è LL(1) perché gli Starter Set sono tutti identici (= {a}) e si può vedere calcolando i FIRST. Per vedere se è LR(0) si può:

- Calcolare i contesti e verificare che rispettino la condizione sufficiente
- Applicare il procedimento operativo e verificare a posteriori se l'automa è deterministico o no

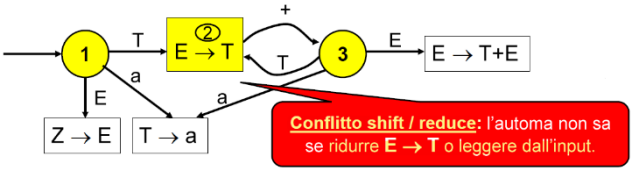
Calcolo dei contesti: lo stato 2 di riduzione ha anche un arco uscente etichettato con simbolo terminale ☹

leftctx(Z) = { ε }		*postulato*
leftctx(E) ⊇ leftctx(Z) • { ε } = leftctx(Z)		$Z \to E$
leftctx(T) ⊇ leftctx(E) • { ε } = leftctx(E)		$E \to \boldsymbol{T}+E$
leftctx(E) ⊇ leftctx(E) • { T+ }		$E \to T+\boldsymbol{E}$
leftctx(T) ⊇ leftctx(E) • { ε } = leftctx(E)		$E \to \boldsymbol{T}$
Al solito, le produzioni che generano simboli terminali non si considerano.		
Unendo i vari contributi:		
leftctx(Z) = { ε }	→	⟨LctxZ⟩ → ε
leftctx(E) = { ε } ∪ leftctx(E) • { T+ }	→	⟨LctxE⟩ → ε | ⟨LctxE⟩ T+
leftctx(T) = leftctx(E)	→	⟨LctxT⟩ → ⟨LctxE⟩
leftctx(T) = leftctx(E) = (T+)*		

Dunque, dal fatto che leftctx(T) = leftctx(E) = (T+)* segue che:

LR(0)ctx(Z → E) = leftctx(Z) • { E } = **E**
LR(0)ctx(E → T+E) = leftctx(E) • { T+E } = (T+)* **T+E**
LR(0)ctx(E → T) = leftctx(E) • { T } = (T+)* **T**
LR(0)ctx(T → a) = leftctx(T) • { a } = (T+)* **a**

Conflitto

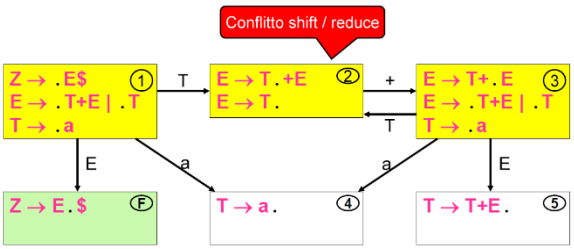


Conflitto shift / reduce: l'automa non sa se ridurre **E → T** o leggere dall'input.

Procedimento operativo: ☹

Se potessimo vedere il prossimo carattere, sapremmo che scelta fare.

$Z \to E$ $E \to T+E \mid T$ $T \to a$



14.4 – ANALISI LR(1)

Con LR(0) i conflitti emergono quasi sempre perché è troppo semplicistico, nella realtà è inutilizzabile. **L'analisi LR(K) opera analogamente all'analisi LR(0), ma guarda avanti di k simboli (Lookahead symbols):**

- Le riduzioni sono ritardate di k simboli
- Le definizioni di contesto e contesto sinistro sono estese considerando i k simboli successivi
- Il procedimento operativo viene esteso considerando i k simboli successivi

Purtroppo, **la complessità di questa tecnica fa sì che LR(1) sia spesso ingestibile e richieda semplificazioni,** mentre LR(2) e successivi sono di fatto neanche pensabili.

14.4.1 – CONTESTI LR(K)

Formalmente il contesto LR(K) di una produzione A → α è così definito:

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

Tutte le stringhe del contesto LR(k) della produzione A → α hanno la forma $\beta\alpha u$ e differiscono per il prefisso β e per la stringa **u**.

La stringa u appartiene all'insieme FOLLOWk(A) delle stringhe di lunghezza k che possono seguire il simbolo non terminale A:

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

Dove FOLLOW1(A) non è altro che l'insieme FOLLOW(A) dei Following Symbols già introdotto nella definizione dei Director Symbols nell'analisi LL (vedi DIRECTOR SYMBOLS SET).

Da qui, il contesto sinistro di un non-terminale A è:

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

Dove il contesto LR(K) della produzione A → α diventa:

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

Anche i due **POSTULATI** si estendono analogamente:

- **leftctx(Z, ε) = { ε }**
- Se c'è una produzione P → $\gamma Q\delta$, posto $v \in FOLLOW_k(P)$ e $u = FIRST(L(\delta) \cdot v, k)$, abbiamo che **leftctx(Q, u)** ⊇ **leftctx(P, v)** · **{γ}** che implica una grammatica regolare sinistra

14.4.2 – AUTOMA CARATTERISTICO LR(1)

- Una grammatica G con *n* metasimboli e *t* terminali comporta una grammatica dei contesti sinistri con potenzialmente (*n*-1) * t^k + 1 metasimboli (nel caso LR(0) sarebbero stati al più *n*)
- Per un *tipico linguaggio con 50–100 terminali*, ciò significa una grammatica dei contesti sinistri *50-100 volte più grande* del caso LR(0), ossia *praticamente intrattabile.*
- Per questo, l'approccio LR(1) completo è spesso sostituito da *versioni semplificate, approssimate ma meno onerose.*

Si procede analogamente al caso LR(0):

- Calcolo espressioni regolari per contesti LR(K)
- Le uso per costruire l'automa caratteristico

Questo approccio, già non banale per k=0, diventa ancora più lungo e complesso per k>0.

CALCOLO DEI CONTESTI LR(1)

Secondo i postulati, in pratica, presa la produzione P → $\gamma Q\delta$, si opera in questo modo:

- **I possibili valori di v** sono i simboli terminali che possono seguire P
- Si considera il linguaggio L(δ) generato da δ e gli si appende in coda v (questo ha influenza solo se L(δ) comprende ε)

- Si prendono le **<u>iniziali delle stringhe così ottenute</u>**: sono i **possibili valori di u**, ognuno dei quali dà luogo ad un diverso contesto sinistro di Q

$Z \rightarrow S\$$
$B \rightarrow C \mid Db$
$S \rightarrow CbBA$
$C \rightarrow a$
$A \rightarrow ab \mid Aab$
$D \rightarrow a$

Esempio – Approccio dei contesti

Le regole che contengono solo simboli terminali, come sempre non si considerano nel calcolo dei contesti sinistri.

Considerando la grammatica a lato e i postulati, il contesto sinistro di S - quando dopo c'è dollaro - contiene il ctx sx di Z quando dopo non c'è nulla, compreso il caso in cui dopo non ci sia nulla.

La "seccatura" è che dobbiamo guardare anche nelle altre regole perché dobbiamo vedere cosa c'è dopo. Es. nella produzione di S, dobbiamo andare nella regola Z.

Z → S$:	leftctx(Z, ε) = { ε }			
	leftctx(S, $) ⊇ leftctx(Z, ε) · {ε}	con δ=$,	$\gamma=\varepsilon$	
S → C.bBA	leftctx(C, b) ⊇ leftctx(S, $) · {$\varepsilon$}	con δ=bBA,	$\gamma=\varepsilon$,	u=b
S → Cb.BA	leftctx(B, a) ⊇ leftctx(S, $) · {Cb}	con δ=A,	γ=Cb,	u=a
S → CbB.A	leftctx(A, $) ⊇ leftctx(S, $) · {CbB}	con δ=$,	γ=CbB,	u=$

Nella produzione B→C, poiché nulla precede C, si ha che $\gamma=\varepsilon$. Per dedurre δ, bisogna guardare dove è usato B: poiché esso è usato in S→CbBA, dove è seguito da A, δ=A, e poiché A genera a, u=a.

B → C	leftctx(C, a) ⊇ leftctx(B, a) · {ε}	con δ=A,	$\gamma=\varepsilon$,	u=a

Nella produzione B → Db, poiché nulla precede D, si ha che $\gamma=\varepsilon$, inoltre, ovviamente, δ=b.

B → Db	leftctx(D, b) ⊇ leftctx(B, a) · {ε}	con δ=b,	$\gamma=\varepsilon$

Nella produzione A → Aab, ovviamente $\gamma=\varepsilon$. **<u>u può essere "a", ma non solo</u>**: giacché A viene usato anche in S → CbBA, dove è seguito da ε, questo **fa entrare in gioco v, ossia i simboli che possono seguire A**, tra cui $, introdotto dalla regola Z → S$.

A → Aab	leftctx(A, a) ⊇ leftctx(A, $) · {$\varepsilon$}	e	leftctx(A, a) ⊇ leftctx(A, a) · {ε}

Unendo tutto insieme:

leftctx(Z, ε) = { ε }
leftctx(D, b) ⊇ leftctx(B, a) · {ε}
leftctx(C, b) ⊇ leftctx(S, $) · {$\varepsilon$}
leftctx(A, $) ⊇ leftctx(S, $) · {CbB}

leftctx(S, $) ⊇ leftctx(Z, ε) · {ε}
leftctx(B, a) ⊇ leftctx(S, $) · {Cb}
leftctx(C, a) ⊇ leftctx(B, a) · {ε}
leftctx(A, a) ⊇ leftctx(A, $) · {$\varepsilon$}

eliminato perché inutile
~~leftctx(A, a) ⊇ leftctx(A, a) · {ε}~~

Ora, concatenando I suffissi, si ottengono gli LR(1) ctx:

LR(1)ctx(Z → S$) = leftctx(Z, ε) · S$ · { u=FOLLOW(Z) } = **S$**

LR(1)ctx(S → CbBA) = leftctx(S, $) · CbBA · { u=FOLLOW(S) } = **CbBA$**

LR(1)ctx(A → Aab) = leftctx(A, a) · Aab · { u∈FOLLOW(A), u=a } + leftctx(A, $) · Aab · { u∈FOLLOW(A), u=$ } =
= **CbB Aaba + CbB Aab$**

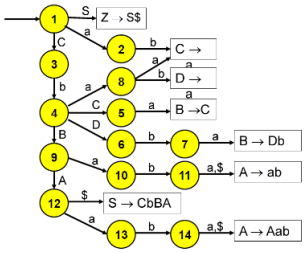
LR(1)ctx(A → ab) = leftctx(A, a) · ab · { u∈FOLLOW(A), u=a } + leftctx(A, $) · ab · { u∈FOLLOW(A), u=$ } =
= **CbB aba + CbB ab$**

LR(1)ctx(B → C) = leftctx(B, a) · C · { u∈FOLLOW(C), u=a } = **CbCa**

LR(1)ctx(B → Db) = leftctx(B, a) · Db · { u∈FOLLOW(C), u=a } = **CbDba**

LR(1)ctx(C → a) = leftctx(C, a) · a · { u∈FOLLOW(C), u=a } + leftctx(C, b) · ab · { u∈FOLLOW(C), u=b } =
= **Cbaa + ab**

LR(1)ctx(D → a) = leftctx(D, b) · a · { u∈FOLLOW(D), u=b } = **Cbab**



L'automa caratteristico LR(1) risultante è perfettamente deterministico, a conferma che la grammatica data è LR(1).
Invece, i contesti LR(0) avrebbero dei conflitti.

$Z \rightarrow S\$$
$B \rightarrow C \mid Db$
$S \rightarrow CbBA$
$C \rightarrow a$
$A \rightarrow ab \mid Aab$
$D \rightarrow a$

Esempio – approccio operativo

Dobbiamo tenere conto in ogni stato anche del simbolo successivo (lookahead symbol). Come convenzione, a fianco di ogni regola specificheremo il lookahead set, cioè i possibili simboli di input che rendono valida l'azione. Anything = ? o *

La costruzione degli stati diventa più complessa del caso LR(0) perché ora per ogni regola bisogna computare anche il nuovo lookahead set, svolgendo in pratica gli stessi passi che portano ai contesti LR(1).

Nella regola di top level, per ipotesi il lookahead set è sempre Anything perché non si andrà mai oltre il terminatore $. Z → S$: ?

Consideriamo ora la regola S → .CbBA. Dopo il cursore c'è C, quindi considero anche la produzione C → .a

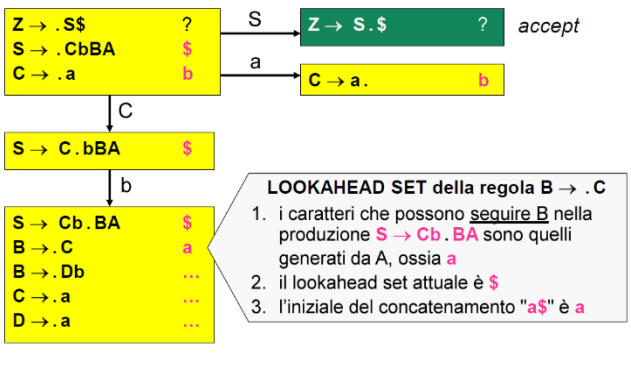
A differenza del caso LR(0), ora devo calcolare anche i lookahead set per ogni regola.

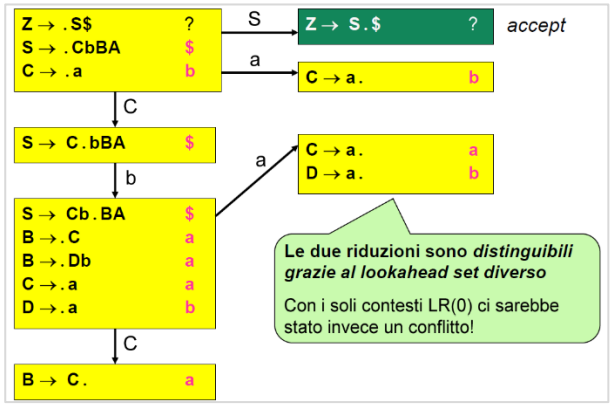
Lookahead set per S → CbBA:

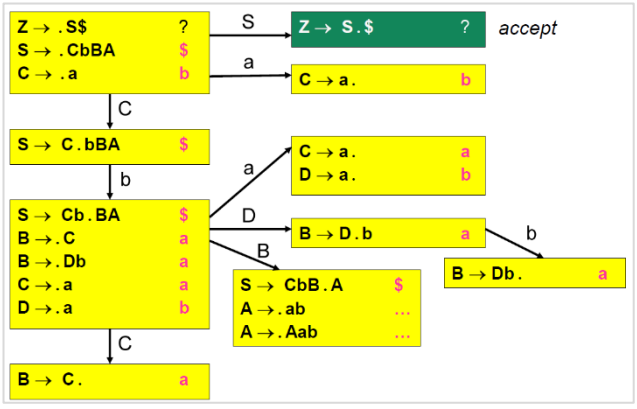
1. Guardo caratteri che seguono S nella produzione di livello superiore Z –Z .S$, cioè $
2. Concateno $ con il lookahead set attuale ?, ottenendo $?
3. K=1, quindi prendo l'iniziale della stringa appena ottenuta, cioè $

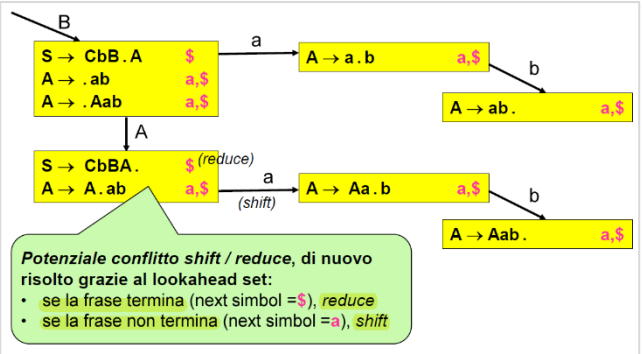
Lookahead set per C → .a:

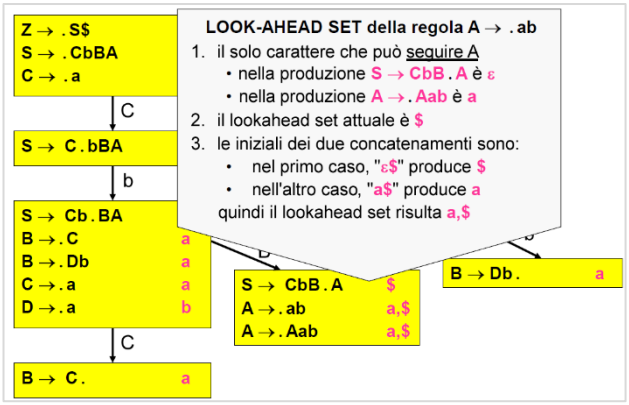
1. Caratteri che seguono C nella produzione di livello superiore S → .CbBA, cioè b
2. Concateno b con il lookahead attuale $, ottengo b$
3. Tengo solo l'iniziale perché k=1, quindi b

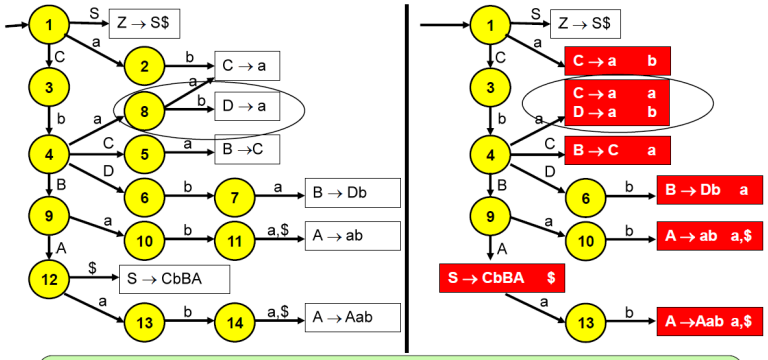












Sono di fatto identici: *il secondo ha meno stati*, perché nasce pre-ottimizzato: quando entra in una successione di stati il cui risultato è certo, effettua l'azione appena possibile, senza ritardarla inutilmente.

14.5 – SLR

È praticamente impossibile costruire un parser LR(1) per un vero linguaggio perché le sue dimensioni sarebbero intrattabili. Per questo esistono delle versioni semplificate, non potenti come il vero LR(1), ma adatte a molti casi concreti, che accorpano gli stati simili, riducendo il numero totale di stati dell'automa:

- SLR (Simple LR)
- LALR (Look-Ahead LR)

14.5.1 – CONTESTI SLR(K)

LR(0) è fregato perché non sa cosa c'è dopo, ma cosa frena LR(1)? Lo vuole fare in maniera pignola: io sono a, se dopo di me c'è b, oppure se c'è d bla bla bla. Serve una via di mezzo.
Partiamo dal contesto LR(0), che è facile da calcolare, ma non basta: mettiamo in pista i simboli che seguono, senza stare a specificare tutti i vincoli. Guardo cosa potrebbe seguire "a", non in questa precisa regola, ma in generale guardando tutta la grammatica. Usiamo un ragionamento "a grana grossa".
Da notare che PRIMA provi con LR(0), quindi se funziona bene, ma se non funziona non buttiamo via il calcolo: lo completiamo applicandoci in fondo il calcolo dei Follow. Se così facendo gli insiemi si disgiungono, noi abbiamo ottenuto con meno fatica un risultato decente!
METAFORA DELLE TAGLIE: con LR(0) ho la taglia unica, con LR(1) ho il sarto che cuce i vestiti su misura. Con questo SRL invece è come avere le taglie S, M, L, XL… sono statisticamente adatte a tutti, anche se non andranno perfette a tutti i corpi.

Il contesto SLR(K) di una produzione A→α è definito:

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

Cioè, può essere calcolato facilmente a partire dal contesto LR(0), che è molto più semplice.
L'idea è quindi quella di provare a calcolare i contesti SLR e vedere se presentano conflitti: se non ce ne sono, si può usare SLR al posto di LR(1) completo.

È possibile dimostrare che:

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

ovvero *il contesto SLR è un po' più grande*, e quindi più esposto a potenziali conflitti, *del contesto LR completo.*

Esempio – non SLR(1)

Riprendiamo la "solita" grammatica:

Z → S$	S → CbBA	A → ab | Aab
B → C | Db	C → a	D → a

Come primo passo dobbiamo calcolare i contesti sinistri LR(0):

leftctx(Z) = { ε }	*postulato*
leftctx(S) ⊇ leftctx(Z) • { ε }	Z → S $
leftctx(C) ⊇ leftctx(S) • { ε }	S → C bBA
leftctx(B) ⊇ leftctx(S) • { Cb }	S → Cb B A
leftctx(A) ⊇ leftctx(S) • { CbB }	S → CbB A
leftctx(C) ⊇ leftctx(B) • { ε }	B → C
leftctx(D) ⊇ leftctx(B) • { ε }	B → D b
leftctx(A) ⊇ leftctx(A) • { ε }	A → A ab
leftctx(A) ⊇ leftctx(A) • { ε }	A → A ab

Ora uniamo i vari contributi...

leftctx(Z) = { ε }
leftctx(S) = { ε }
leftctx(B) = { Cb }
leftctx(A) = { CbB }
leftctx(C) = { ε + Cb }
leftctx(D) = { Cb }

... e da lì estraiamo i contesti LR(0):

LR(0)ctx(Z → S$) = { ε } • { S$ }
LR(0)ctx(S → CbBA) = { ε } • { CbBA }
LR(0)ctx(B → C) = { Cb } • { C }
LR(0)ctx(B → Db) = { Cb } • { Db }
LR(0)ctx(A → ab) = { CbB } • { ab }
LR(0)ctx(A → Aab) = { CbB } • { Aab }
LR(0)ctx(C → a) = { ε + Cb } • { a }
LR(0)ctx(D → a) = { Cb } • { a }

Infine, concateniamo a costoro gli insiemi FOLLOW corrispondenti:

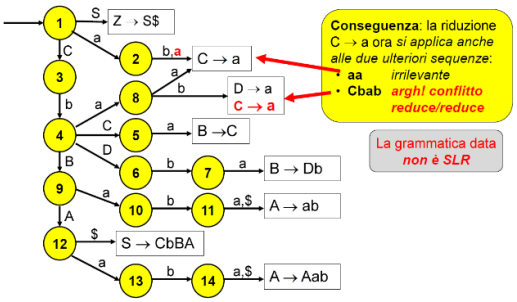
SLR(1)ctx(Z → S$)	= LR(0)ctx(...) • FOLLOW(Z) = { S$ } • { ε }
SLR(1)ctx(S → CbBA)	= LR(0)ctx(...) • FOLLOW(S) = { CbBA } • { $ }
SLR(1)ctx(B → C)	= LR(0)ctx(...) • FOLLOW(B) = { CbC } • { a }
SLR(1)ctx(B → Db)	= LR(0)ctx(...) • FOLLOW(B) = { CbDb } • { a }
SLR(1)ctx(A → ab)	= LR(0)ctx(...) • FOLLOW(A) = { CbBab } • { a,$ }
SLR(1)ctx(A → Aab)	= LR(0)ctx(...) • FOLLOW(A) = { CbBAab } • { a,$ }
SLR(1)ctx(C → a)	= LR(0)ctx(...) • FOLLOW(C) = { a+Cba } • { a,b }
SLR(1)ctx(D → a)	= LR(0)ctx(...) • FOLLOW(D) = { Cba } • {b }

SLR(1)ctx(Z → S$)	= S$
SLR(1)ctx(S → CbBA)	= CbBA$
SLR(1)ctx(A → ab)	= CbBaba + CbBab$
SLR(1)ctx(A → Aab)	= CbBAaba + CbBAab$
SLR(1)ctx(B → C)	= CbCa
SLR(1)ctx(B → Db)	= CbDba
SLR(1)ctx(C → a)	= aa + Cbaa + ab + Cbab
SLR(1)ctx(D → a)	= Cbab

Sono quasi identici ai contesti LR(1)... **tranne questo che è più grande!**

LR(1)ctx(Z → S$)	= S$
LR(1)ctx(S → CbBA)	= CbBA$
LR(1)ctx(A → ab)	= CbBaba + CbBab$
LR(1)ctx(A → Aab)	= CbBAaba + CbBAab$
LR(1)ctx(B → C)	= CbCa
LR(1)ctx(B → Db)	= CbDba
LR(1)ctx(C → a)	= Cbaa + ab
LR(1)ctx(D → a)	= Cbab

Guardando il risultato finale e confrontando SLR(1) con LR(1), l'informazione spuria evidenziata ha inquinato il resto? Ha creato un conflitto? "aa" c'è da un'altra parte? No. "Cbab" c'era da un'altra parte? No.
Quindi, quell'informazione spuria non crea problemi perché dovrei usarla in due situazioni che non si verificheranno mai.
Purtroppo, guardando l'automa caratteristico SLR(1) vediamo che con la stringa "aa", non si hanno effettivamente problemi, mentre con la "Cbab" sì perché si ha un punto di conflitto.
Se nello stato 8 l'automa riceve "b", deve ridurre a C o a D?



Esempio

Riconsideriamo questa grammatica, che ***sappiamo non essere LR(0)***

Z → E $ E → T+E | T T → a

Avevamo ottenuto:

LR(0)ctx(Z → E)	= leftctx(Z) • { E }	= E
LR(0)ctx(E → T+E)	= leftctx(E) • { T+E }	= (T+)* T+E
LR(0)ctx(E → T)	= leftctx(E) • { T }	= (T+)* T
LR(0)ctx(T → a)	= leftctx(T) • { a }	= (T+)* a

Conflitto

Calcoliamo i contesti SLR. Il simbolo $ ovviamente ora va evidenziato, perché il calcolo dei contesti SLR mette in gioco *anche i simboli che seguono*, non più solo i prefissi come accadeva nell'LR(0).

SLR(1)ctx(Z → E)	= E • FOLLOW(Z)	= E
SLR(1)ctx(E → T+E)	= (T+)* T+E • FOLLOW(E)	= (T+)* T+E $
SLR(1)ctx(E → T)	= (T+)* T • FOLLOW(E)	= (T+)* T $
SLR(1)ctx(T → a)	= (T+)* a • FOLLOW(T)	= (T+)* a ($ | +)

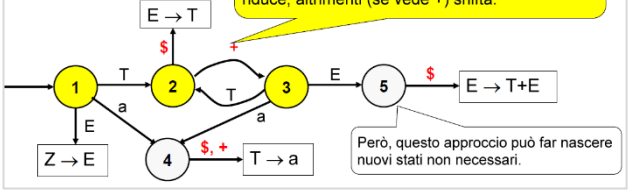
diversi!

SLR(1)ctx(Z → E)	= E
SLR(1)ctx(E → T+E)	= (T+)* T+E $
SLR(1)ctx(E → T)	= (T+)* T $
SLR(1)ctx(T → a)	= (T+)* a ($ | +)

Non più in conflitto (il secondo non è un prefisso del primo)

La grammatica è *SLR(1)*

***Il parser SLR sfrutta il prossimo simbolo* per decidere se ridurre o shiftare:** se vede il $ riduce, altrimenti (se vede +) shifta.



In questo caso, ad esempio, anche se la grammatica che abbiamo non è LR(0), vediamo che è SLR(1) perché viene fuori un parser SLR deterministico che riesce a risolvere il conflitto.

PROCEDIMENTO OPERATIVO

Anche in questo caso c'è un modo più immediato per ottenere il parser SLR: si prende l'automa LR(0) e si tolgono certe riduzioni incompatibili con il lookahead set: **una riduzione A→α va inserita nella tabella di parsing SOLO SE il prossimo simbolo appartiene a FOLLOW(A).**

Quindi, in pratica:

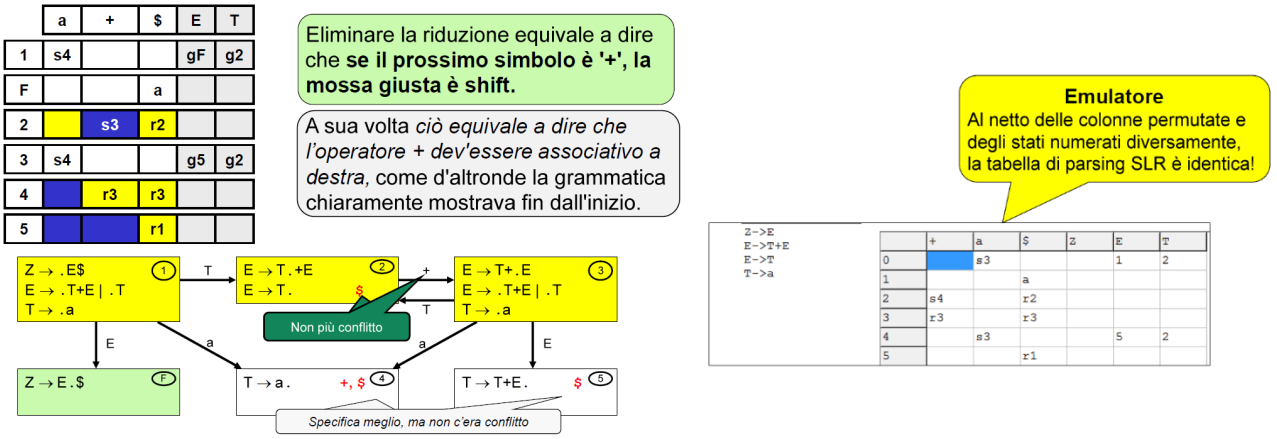
- **Nel parser LR(0) le riduzioni sono presenti per tutti i terminali** perché se si sceglie di ridurre non si legge da input e quindi non si può distinguere in base ad esso (avrò RN in tutta la riga)
- **Nel parser SLR** invece si può spiare avanti e usare il prossimo simbolo per discriminare, cioè **la riduzione sarà presente SOLO nelle colonne compatibili con il lookahead set**

Esempio

	a	+	$	E	T
1	s4			gF	g2
F			a		
2	r2	s3,r2	r2		
3	s4			g5	g2
4	r3	r3	r3		
5	r1	r1	r1		

	a	+	$	E	T
1	s4			gF	g2
F			a		
2		s3	r2		
3	s4			g5	g2
4		r3	r3		
5			r1		

In pratica, cancelliamo delle caselle.



	+	a	$	Z	E	T
0		s3			1	2
1			a			
2	s4		r2			
3	r3		r3			
4		s3			5	2
5			r1			

Esempio variante

La grammatica considerata adesso è analoga alla precedente ma è ricorsiva a sinistra e ha una produzione in più che include l'uso delle parentesi.

Z → E $ **E → E+T | T** **T → a | (E)**

leftctx(Z) = { ε } *postulato*
leftctx(E) ⊇ leftctx(Z) • { ε } = leftctx(Z) *Z → E*
leftctx(T) ⊇ leftctx(E) • { E+ } = leftctx(E) *E → E +T*
leftctx(T) ⊇ leftctx(E) • { ε } = leftctx(E) *E → T*
leftctx(E) ⊇ leftctx(T) • { (} *T → (E)*

Al solito, le produzioni che generano simboli terminali non si considerano.
Unendo i vari contributi e passando alla notazione grammaticale:

leftctx(Z) = { ε } → ⟨LctxZ⟩ → ε
leftctx(E) = { ε } ∪ leftctx(T) • { (} → ⟨LctxE⟩ → ε | ⟨LctxT⟩ (
leftctx(T) = leftctx(E) • ({ ε } ∪ {E+}) → ⟨LctxT⟩ → ⟨LctxE⟩(ε |E+)

Sostituendo:
⟨LctxZ⟩ → ε
⟨LctxT⟩ → ⟨LctxE⟩(ε |E+)
⟨LctxE⟩ → ε | ⟨LctxT⟩ (→ ε | ⟨LctxE⟩(ε |E+) (
Risolvendo:
⟨LctxE⟩ = ((ε |E+) ( )*
⟨LctxT⟩ = ((ε |E+) ( )* (ε |E+)

Non è LR(0) [era da dire..]
Conflitto fra E ed E+T

Da cui i contesti sinistri LR(0):
LR(0)ctx(Z → E) = leftctx(Z) • { E } = **E**
LR(0)ctx(E → E+T) = leftctx(E) • { E+T } = **((ε |E+) ( )* E+T**
LR(0)ctx(E → T) = leftctx(E) • { T } = **((ε |E+) ( )* T**
LR(0)ctx(T → a) = leftctx(T) • { a } = **((ε |E+) ( )* (ε |E+) a**
LR(0)ctx(T → (E)) = leftctx(T) • { (E) } = **((ε |E+) ( )* (ε |E+) (E)**

Concatenandoli ai contesti sinistri LR(0), e introducendo il terminatore $ nella produzione di top-level (come sempre quando si passa a LR(1)):

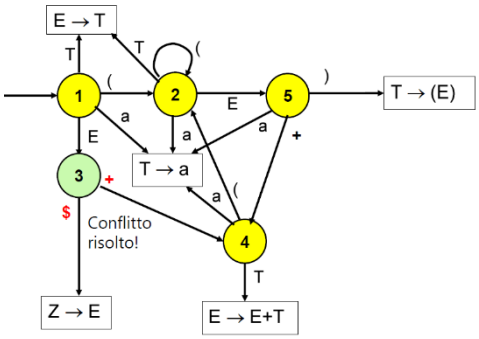
SLR(1)ctx(Z → E$) = **E$** • { ε }
SLR(1)ctx(E → E+T) = ((ε |E+) ( )* **E+T** • { $, +,) } }
SLR(1)ctx(E → T) = ((ε |E+) ( )* **T** • { $, +,) } }
SLR(1)ctx(T → a) = ((ε |E+) ( )* (ε |E+) **a** • { $, +,) }
SLR(1)ctx(T → (E)) = ((ε |E+) ( )* (ε |E+) **(E)** • { $, +,) }

Conflitto risolto

Dunque, la grammatica data non è LR(0).

Per capire se possa essere SLR occorre vedere se l'aggiunta dei simboli FOLLOW riesce differenziare i contesti – in particolare se riesce a differenziare E da E+T.

FOLLOW(Z) = { ε }
FOLLOW(E) = { $, +,) }
FOLLOW(T) = { $, +,) }



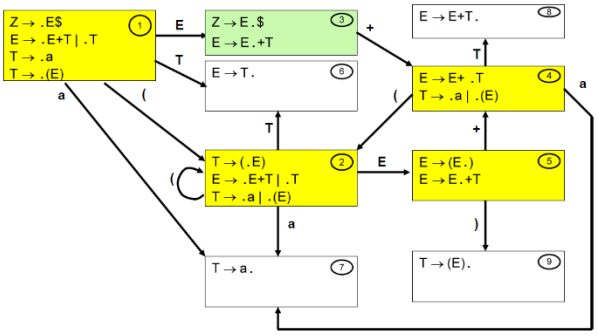


Tabella di parsing SLR(1)

	(	)	+	a	$	E	T
1	s2			s7		g3	g6
2	s2			s7		g5	g6
3			s4		a		
4	s2			s7			g8
5		s9	s4				
6		r2	r2		r2		
7		r3	r3		r3		
8		r1	r1		r1		
9		r4	r4		r4		

14.6 – LALR

Abbiamo visto come funziona il parser LR (dal basso verso l'alto).
Passando da LR(0) a LR(1) si ha troppa complessità, allora sono state inventate delle forme intermedie dette LR(0.5) per esprimere una complessità inferiore.

SLR cerca di guardare i simboli che possono seguire il carattere corrente, senza vederli davvero, ma individua i caratteri che potrebbero esserci. Si usano i simboli FOLLOW, quindi si hanno degli insiemi.
Questo implica però che **si hanno anche dei falsi positivi** perché i contesti che saltano fuori sono un po' più grandi del necessario: il che implica che potrebbero esserci più conflitti. In realtà, statisticamente, spesso funziona e non si hanno conflitti.

Esiste però un'altra possibilità per semplificare: ammettiamo di aver fatto il calcolo del LR(1) e di avere una marea di stati. Se li guardo da vicino molti stati sono molto simili e differiscono solamente per epsilon: si possono allora usare delle approssimazioni intelligenti dando una pulita e considerando uguali cose che non sono proprio uguali ma molto simili. Ovviamente bisogna prendere lo stato più grande dei due.

Formalmente, un approccio alternativo ad SLR consiste nel collassare in un solo stato quegli stati che, nel parser LR(1) completo, sono identici a meno dei lookahead set. Si parla allora di LookAhead LR, cioè **LALR**.

- **PRO**: trasformazione sempre possibile, spesso conveniente: parser LALR ha molti meno stati dell'LR
- **CONTRO**: possono apparire conflitti reduce/reduce, tipicamente gestibili

Questo approccio si implementa calcolando prima LR(0) e poi aggiungendo i lookahead set, calcolati separatamente.

Esempio

Questa non è una grammatica SLR né LR(0), è invece LALR. Per dimostrarlo:

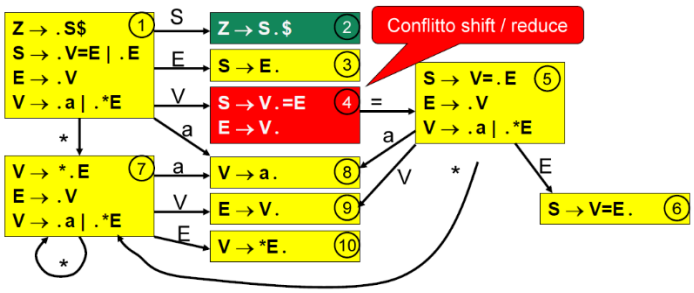
Consideriamo questa grammatica (sintassi astratta di un subset del C):

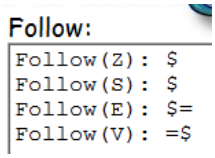
0) Z → S $
1) S → V=E
2) S → E
3) E → V
4) V → a
5) V → *E

- Calcoliamo il parser LR(0) → conflitti
- Tentiamo l'approccio SLR → conflitti
- Calcoliamo il parser LR(1) → no conflitti
- Riduciamo gli stati tramite LALR

(Nella realtà ovviamente non si fa così ma ci si arriva in modo più diretto)

Nello stato 4, la variabile può essere che debba essere ridotta a espressione (right value, voglio leggere il valore) oppure può essere il riferimento del contenuto (left value, è il nome della variabile). C'è conflitto perché non posso vedere il carattere dopo e quindi non so se dopo c'è l'uguale o meno.





con FOLLOW(A). In questo esempio FOLLOW(E) = {$, =}, ma è proprio il simbolo "=" ad aprire anche l'alternativa shift → il conflitto è ineliminabile, quindi l'approccio SLR non funziona.

Passare al parser SLR(1) implica eliminare dalla parsing table le riduzioni A→α incompatibili

Nella parsing table SLR(1) vengono cancellate le regole incompatibili con quelle che seguono, però tra quelle cancellate non c'è quella rossa che rompeva le scatole.

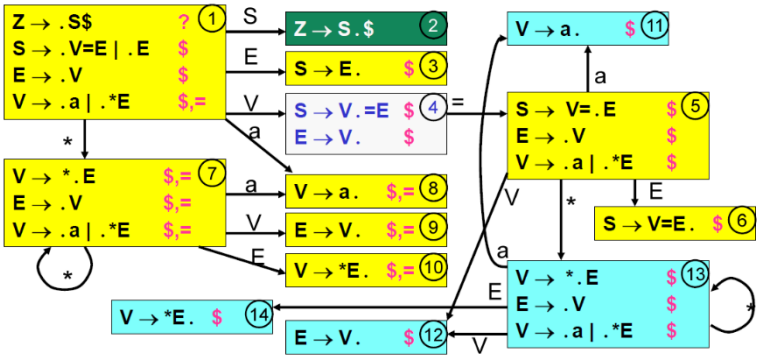
	a	*	=	$	S	E	V
1	s8	s7			g2	g3	g4
2				a			
3	r2	r2	r2	r2			
4	r3	r3	r3,s5	r3			
5	s8	s7				g6	g4
6	r1	r1	r1	r1			
7	s8	s7				g10	g9
8	r4	r4	r4	r4			
9	r3	r3	r3	r3			
10	r5	r5	r5	r5			

LR(0)

	a	*	=	$	S	E	V
1	s8	s7			g2	g3	g4
2				a			
3			r2	r2			
4			r3,s5	r3			
5	s8	s7				g6	g9
6			r1	r1			
7	s8	s7				g10	g9
8			r4	r4			
9			r3	r3			
10			r5	r5			

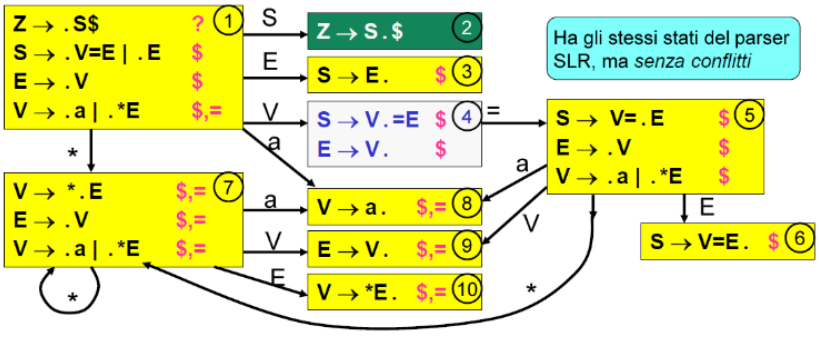
SLR(1)

Non poter cancellare riduzioni dalla parsing table equivale a dire che se il prossimo simbolo è "=", la mossa giusta potrebbe comunque essere sia shift sia reduce, a seconda delle circostanze.
Per constatarlo, se proviamo a seguire il parser nel riconoscimento della frase "a=a", vedremo che in momenti successivi il parser dovrà shiftare e poi ridurre: servono entrambe le mosse!



Proviamo allora a calcolarci LR(1) e otterremo un parser che non ha più conflitti, però ovviamente gli stati sono aumentati e in particolare sono 14 (e stiamo giocando con una grammatica cagarina: se avessimo tutti i caratteri ASCII? Addio…).

Siccome il LALR prevede di collassare in un solo stato quegli stati che sono identici a meno dei lookahead set, vediamo che in questo caso 7≡13, 8≡11, 9≡12, 10≡14. Otteniamo allora il **parser LALR:**



	a	*	=	$	S	E	V
1	s8	s7			g2	g3	g4
2				a			
3				r2			
4			s5	r3			
5	s8	s7				g6	g9
6				r1			
7	s8	s7				g10	g9
8			r4	r4			
9			r3	r3			
10			r5	r5			

15 – LAB per GENERAZIONE AUTOMATICA RICONOSCITORI LR

Nato per Unix, portato poi su altre piattaforme: **YACC (Yet Another Compiler Compiler)** a partire dalla descrizione della grammatica (context free), genera l'analizzatore sintattico LR(1), SLR, o LALR(1).
Usa **LEX (Lexical Analyser Generator)** per ottenere il prossimo token: basandosi su espressioni regolari, LEX analizza lo stream di input, cerca i token ed esegue le azioni specificate. Il riconoscimento delle espressioni regolari si basa su un RSF deterministico.
Dopo YACC, sono nati tanti altri generatori, come:

- **CUP** LALR Parser Generator di Java, che aveva anche un'interfaccia grafica
- **GOLD PARSER**: aveva un motore universale in cui iniettare la tabella. Si poteva compilare il linguaggio e includere il runtime per avere il macchinino funzionante. Aveva anche un bel supporto grafico
- **SABLEcc**: generatore di parser LALR(1) in Java con supporto grammatiche EBNF.
    - **PRO**: rileva e riporta conflitti shift/reduce e genera automaticamente le classi relative all'albero inserendo le relative azioni
    - **CONTRO**: non ha direttive di precedenza

15.1 – YACC (Yet Another Compiler Compiler)

Funzioni fondamentali (versione originale in C):

- **yylex** per il lessico
- **yyerror** per gli errori
- **yyparse** per la grammatica context-free

Ha **direttive di precedenza** per la gestione automatica delle priorità e delle associatività degli operatori:

- Servono a risolvere conflitti shift/reduce
- Ogni operatore può essere associativo (a sinistra, a destra) o non associativo – left, right, nonassoc. Quando ci sono dei conflitti, si usano queste informazioni aggiuntive per dire che azione preferire
- L'ordine con cui sono definiti gli operatori va dal meno prioritario (primo citato) al più prioritario (ultimo citato)

15.2 – SABLEcc

- LALR(1)
- Produzioni nella forma: **assegnamento = id uguale exp puntoevirgola ;** (Terminali)
- **Non ha direttive di precedenza**
- Non consente di agganciare azioni alle produzioni (come invece consentiva JavaCC)
- Genera automaticamente le classi dell'APT di cui il parser farà uso
- Riporta **sia i conflitti shift/reduce, sia reduce/reduce** emettendo messaggi d'errore

Esempio del Dangling Else Problem slide 22-32 (if annidati non hanno "else" obbligatorio)

15.3 - GOLDPARSER

- LALR(1)
- Architettura modulare:
    - BUILDER: genera la parsing table a partire da un file di testo nel formato EBNF
    - ENGINE: interpreta la parsing table, non semanticamente. La semantica è demandata ad altri componenti

PRO: la grammatica non deve essere direttamente inclusa nel codice sorgente e favorisce lo sviluppo si software object-based.

16 – PROCESSI COMPUTAZIONALI (ITERAZIONE e RICORSIONE)

È ben noto dai corsi di Fondamenti cosa siano un ciclo o una funzione ricorsiva, ma spesso si confonde il costrutto linguistico con il processo computazionale sottostante.
OBIETTIVO: concentrarsi sul modello computazionale indipendentemente dalla sintassi, dal linguaggio e dal costrutto linguistico usati.

Cosa caratterizza un processo computazionale "iterativo" rispetto a uno "ricorsivo"?

16.1 – PROCESSI COMPUTAZIONALI ITERATIVI

Nei linguaggi imperativi, il costrutto linguistico che esprime un processo computazionale iterativo è tipicamente il **CICLO**. Al di là della specifica sintassi, **in tale schema:**

- **C'è sempre** una variabile che funge da **ACCUMULATORE**
- **Inizializzata prima** e **modificata durante** il ciclo
- Che, **al termine** del ciclo, **contiene il risultato**

Di conseguenza, al passo k, l'accumulatore contiene il risultato parziale k-esimo: **il processo computazionale computa IN AVANTI.**

```
int fact = 1          ← Accumulatore (valore iniziale: elemento neutro della moltiplicazione)
for(int i=1; i<=n; i++){
    fact = fact * i;  ← L'accumulatore porta avanti il risultato parziale
}
print(fact)           ← Al termine, il valore dell'accumulatore rappresenta il risultato
```

Esempio

Un processo come questo è **fondato sulla distruzione sistematica** di quello che è successo prima: cancello la storia. Non posso tornare indietro e vedere dov'è stato l'errore: devo ricominciare da capo e fare debug con step, step, step...

16.2 – PROCESSI COMPUTAZIONALI RICORSIVI

Al contrario, un processo computazionale ricorsivo è **espresso da una FUNZIONE RICORSIVA.** Al di là della specifica sintassi, in tale schema:

- **Non c'è alcun accumulatore**
- La chiamata ricorsiva ottiene il risultato (n-1)esimo
- Il corpo della funzione opera sul risultato (n-1)-esimo per sintetizzare il risultato (n)-esimo desiderato

Durante le chiamate ricorsive **non c'è alcun risultato parziale**: si svolge solo il problema. **Il risultato è sintetizzato mentre le chiamate si chiudono: il processo computa ALL'INDIETRO.**

```
int fact(int n) {
  return n==0 ? 1 : fact(n-1) * n;
}
print(fact(3))
```

POI lo si elabora per sintetizzare il risultato n-esimo

PRIMA ci si procura il risultato del «caso precedente» (n-1)esimo

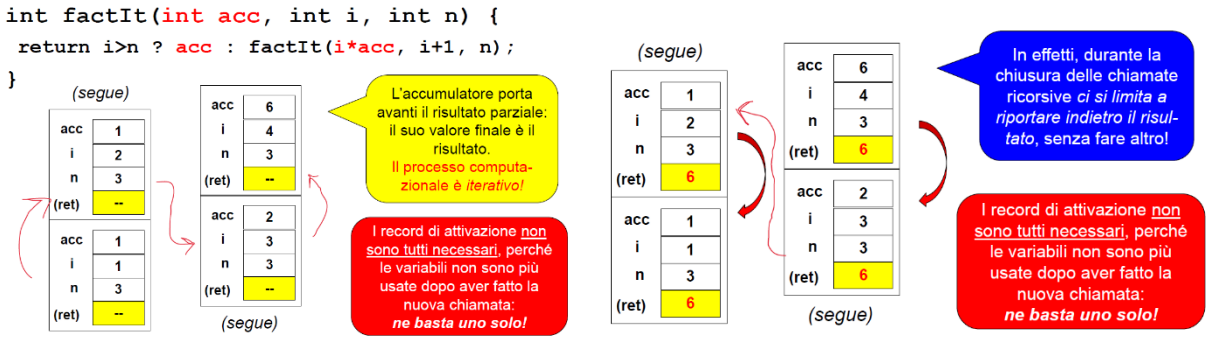
Esempio

È un processo computazionale basato sulla **sintesi di nuovi valori che non sovrascrivono i precedenti.** Se si interrompe l'iterazione al passo k, non si ha in mano niente: il risultato finale viene sintetizzato durante la chiusura delle chiamate.

16.3 – TAIL RECURSION

Un processo computazionale ricorsivo è espresso da una funzione ricorsiva, ma **non è detto che una sintassi ricorsiva dia luogo a un processo computazionale che opera all'indietro**.
Infatti, un costrutto sintatticamente e formalmente ricorsivo può dar luogo a un processo computazionale iterativo, che, cioè, computa in avanti. Ciò accade con la **RICORSIONE IN CODA (TAIL RECURSION)**: la chiamata ricorsiva è l'ultima istruzione della funzione e il risultato parziale k-esimo viene "passato avanti" come argomento.

Esempio
Come nel caso del ciclo, se si interrompe l'iterazione al passo k, l'accumulatore contiene il risultato k-esimo. È un processo computazionale basato sulla sintesi di nuovi valori che però possono sovrascrivere i precedenti perché, computando in avanti, questi ultimi non sono più necessari quando si fa la nuova chiamata.



La ricorsione in coda (tail recursion) è quindi un costrutto sintatticamente ricorsivo che dà luogo a un processo computazionale iterativo, che computa in avanti.
Per questo motivo, può essere usata in alternativa ai cicli per esprimere l'iterazione:

- **Linguaggi funzionali e logici** tendono a seguire questo approccio, ottimizzandone il runtime
- **Linguaggi imperativi**, invece, avendo i cicli, di norma non ottimizzano la tail-recursion

16.3.1 – TRO (TAIL RECURSION OPTIMIZATION)

La Tail Recursion può essere ottimizzata stabilendo di allocare il nuovo record di attivazione sopra il precedente. In tal modo, l'occupazione di risorse è identica al caso ciclico, all'insegna del motto:

stesso processo ↔ stessa immagine a runtime

Iterazione e tail-recursion sono solo forme diverse dello stesso processo computazionale. Tipicamente, la **TRO** è ottimizzata nei linguaggi che non offrono una sintassi specifica per l'iterazione (cicli).
In pratica la tail-recursion è tale e quale ad un ciclo.

- Nei **linguaggi imperativi** tradizionali (C, Java, C#), la **tail recursion non è ottimizzata** perché tali linguaggi offrono già una sintassi specifica per l'iterazione (cicli while e for), quindi la forma ricorsiva è praticamente usata solo per processi computazionali realmente ricorsivi.
- Nei **linguaggi logici** (Prolog) e **funzionali** (Lisp, Scheme), la **tail recursion** invece **è da sempre ottimizzata** perché questi linguaggi non offrono tipicamente una sintassi specifica per l'iterazione, quindi la tail-recursion è l'unico modo per esprimere processi computazionali iterativi.
- Sono sempre più i casi di **linguaggi blended** (Scala, Kotlin) che sono linguaggi di base imperativi a oggetti che prendono idee e approcci dal mondo funzionale → hanno la **tail recursion ottimizzata**

SCALA

Scala (Odersky et al) è un linguaggio blended (ibrido) che unisce gli approcci imperativo e funzionale, anche se, in verità, fa di tutto per spingere l'utilizzatore verso il paradigma funzionale.
Scala **compila in bytecode e gira su JVM,** infatti può anche interoperare con Java.
In Scala, la tail recursion è ottimizzata (limitatamente alla ricorsione diretta): ciò permette di fare confronti (molto istruttivi) fra codice ottimizzato e non ottimizzato.

Esempio

JAVA

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

SCALA

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

In Scala si scrive prima il tipo e poi la variabile: è un'idea del pascal, di 60 anni fa, ma è tornata perché ci si è resi conto che in questo modo è più utile. "def" per definire delle funzioni come in Javascript si usa "function".
Il compilatore **scalac** permette di disabilitare la TRO con l'opzione -g:notailcalls. Usando il disassemblatore javap -c possiamo quindi confrontare i due bytecode prodotti.

Con TRO abilitata, si salvano solo i dati, non si chiama nessuno. Al contrario, con TRO disabilitato si fa come in java e si fa la chiamata attraverso la tabella dei metodi virtuali.

Con TRO abilitata

```
 0: iload_2
 1: iload_3
 2: if_icmple 7
 5: iload_1
 6: ireturn
 7: iload_2
 8: iload_1
 9: imul
10: iload_2
11: iconst_1
12: iadd
13: iload_3
14: istore_3
15: istore_2
16: istore_1
17: goto 0
```

La chiamata alla funzione è scomparsa!

Con TRO disabilitata

```
 0: iload_2
 1: iload_3
 2: if_icmple 9
 5: iload_1
 6: goto 20 // ireturn
 9: aload_0
10: iload_2
11: iload_1
12: imul
13: iload_2
14: iconst_1
15: iadd
16: iload_3
17: invokevirtual #16
    // Method factIt2
20: ireturn
```

Nel caso n=30, inserendo nel codice delle chiamate al metodo Java System.nanoTime() per misurare il tempo di esecuzione, si ottiene:

- **7186 ns**, con TRO disabilitata (-g:notailcalls)
- **6159 ns**, con TRO abilitata con un guadagno di circa il 15% nel caso in questione

Inserendo invece nel codice una chiamata al metodo Java Thread.currentThread().getStackTrace().length che recupera la dimensione dello stack si ottiene:

- **51**, con TRO disabilitata (-g:notailcalls)
- **21,** con TRO abilitata → guadagno fondamentale in termini di risorse stack occupate (–60%)

17 – BASI DI 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":

- Spesso confinati nell'accademia, o in gruppi di "affezionati utenti"
- Spesso visti "con sospetto" dai "veri programmatori"

Nell'ultimo decennio, **l'aumento esponenziale della complessità dei sistemi ne ha fatto riscoprire la validità:**

- Spogliati dalla sintassi (poco user-friendly) del passato, gli approcci funzionali hanno molto da dire e da dare a una sana ingegnerizzazione dei sistemi software
- **Linguaggi blended**: idee dal paradigma funzionale re-innestate e adattate ai moderni linguaggi "main stream"

Il **PARADIGMA "FUNCTIONAL PROGRAMMING"** porta con sé idee che mantengono, e se possibile rinforzano, la propria validità in un moderno contesto blended a oggetti:

- **Distinzione variabili / valori** (var vs val): in java c'è "final", ma in quanti lo abbiamo usato? Tanto se non lo metto funziona lo stesso, quindi praticamente nessuno l'ha messo…
- **Abolizione dei tipi primitivi**: "everything is an object": se quando c'è stato il passaggio C-Java avessero tolto tutti i tipi, nessuno sarebbe migrato ai nuovi linguaggi, ma poi, noi lo sappiamo, abbiamo tutto in doppia gestione (es. int e Integer)
- **Costrutti come espressioni**: "everything is an expression": è tutto uniforme
- **Abolizione delle parti statiche** in favore di **singleton companion**
- **Collezioni e oggetti immodificabili**: "compute by synthesis". Potremmo cominciare a pensare che le collezioni siano immodificabili come le stringhe, e poi quando ho bisogno di lavorare su una lista, ne creo una nuova.
- **FUNZIONI COME FIRST-CLASS ENTITIES:** idea di funzioni come veri **cittadini**! In Java se la funzione fosse un vero oggetto, potrei restituire una funzione come tipo di ritorno ad una funzione, ma non si può fare. Non posso creare un generatore che restituisce funzioni, quindi in realtà le funzioni sono cittadini di serie B. **Nuovo approccio**: ora anche le funzioni sono cittadini veri. Ecco perché si chiama mondo **funzionale: le funzioni sono oggetti come tutti gli altri**
  - Chiusure, lambda expressions
  - Lazy evaluation vs eager evaluation
- E inoltre: concisione, operatori come funzioni…

17.1 – DISTINZIONE VARIABILI/VALORI

Basato sulla modifica di valori, il paradigma imperativo genera programmi difficili da leggere e manutenere perché i simboli cambiano continuamente valore. **Il paradigma funzionale si basa invece sull'idea che i simboli mantengano il valore, come in matematica.**

I moderni **linguaggi blended** uniscono i due mondi:

- La parola chiave **var** denota **variabili** (che possono cambiare valore)
- La parola chiave **val** denotare **valori** (che invece non lo faranno)

Pur supportando l'approccio imperativo, si promuove uno stile più funzionale:

- In certi contesti, l'uso di val è semplificato (o obbligato)
- Proprietà introdotte da val hanno solo getter (non setter)

17.2 – UNIFORMITÀ

17.2.1 – TUTTO È OGGETTO

I linguaggi a oggetti tradizionali, pur chiamandosi a oggetti, si basano su tipi primitivi che oggetti non sono! Ciò causa disuniformità e necessità di trattamento ad hoc:

- Tipi primitivi passati **per valore** vs oggetti passati **per riferimento**
- Array e collezioni di tipi primitivi o gestiti ad hoc (es. array Java) o non ammessi (collezioni Java) → necessità di **classi wrapper**

I linguaggi blended ripuliscono lo spazio concettuale, cancellando i tipi primitivi: EVERYTHING IS AN OBJECT! In Scala, ad esempio, gli interi sono di tipo Int che è una classe come le altre.

17.2.2 – DA STATICO A SINGLETON

I linguaggi a oggetti tradizionali, pur chiamandosi a oggetti, si basano sulle classi, che non sono oggetti. Ciò causa disorientamento e necessità di design pattern:

- Perché si chiama OOP se come prima cosa mi introduci le classi?
- Quando una funzione dev'essere statica e quando un metodo?
- Meglio una funzione statica o un oggetto singleton?

I linguaggi blended cancellano le parti statiche: il loro ruolo è svolto da un **singleton**, il **COMPANION OBJECT**:

- In Scala, il companion object ha lo stesso nome della classe
- In Kotlin, può anche avere nome diverso, ma nello stesso file

17.3 – COSTRUTTI COME ESPRESSIONI

I linguaggi a oggetti tradizionali distinguono comunque, come i loro progenitori non a oggetti, fra istruzioni ed espressioni, ad esempio “for, while, if, throw” sono istruzioni (statement), non espressioni: denotano azioni, non valori.
La presenza di istruzioni induce a esprimere computazioni mettendole in sequenza, il che implica variabili di appoggio, ma le variabili modificabili rendono illeggibili i programmi…
I linguaggi blended cancellano le istruzioni in favore di espressioni, secondo il motto "**EVERYTHING IS AN EXPRESSION**": in Scala e kotlin, ad esempio, i costrutti “for, while, if, throw” diventano espressioni che producono valori e sono quindi combinabili in cascata.

17.4 – COLLEZIONI E OGGETTI IMMODIFICABILI

I linguaggi a oggetti tradizionali offrono **collezioni** (es. JCF) che tipicamente sono **totalmente modificabili:**

- Difficoltà con concorrenza e parallelismo
- Promozione di uno stile imperativo basato sulla continua modifica di oggetti e valori

L'approccio funzionale suggerisce invece di **computare per sintesi di nuovi oggetti**, senza modificare l'esistente (una manna per concorrenza e parallelismo).
I linguaggi blended offrono tipicamente entrambi i tipi di collezioni, ma spesso facilitando quelle immodificabili: in Scala, ad esempio, le collezioni immodificabili sono importate di default e sono quindi più dirette da usare, mentre per usare quelle modificabili bisogna fare uno sforzo maggiore e farlo "volutamente".

17.5 – FUNZIONI COME FIRST-CLASS ENTITIES

Tutti i linguaggi di programmazione supportano costrutti per esprimere funzioni, ma ben di rado come first-class entities. **Nei linguaggi imperativi tradizionali**, invece, una funzione tipicamente è solo un costrutto che incapsula codice. Una **funzione** così intesa non è manipolabile come ogni altro tipo di dato:

- **Non può essere assegnata a variabili.** Una approssimazione fatta sono i puntatori a funzione del C, che però contengono solo l'indirizzo del codice, non una vera "entità funzione"..
- **Non può essere passata come argomento a un'altra funzione** perché gli argomenti devono essere valori di un qualche tipo, mentre le funzioni in quei linguaggi non lo sono
- **Non può essere restituita da un'altra funzione** perché non si può "sintetizzare comportamento" come fosse un valore (e le funzioni esprimono "comportamenti"…). Al più, si può restituire l'indirizzo di una funzione già esistente
- **Non può essere definita e usata "al volo"**: le funzioni devono essere definite nel codice, staticamente

Una funzione che sia first-class entity deve essere manipolabile come ogni altro tipo di dato e quindi:

- Deve poter essere **assegnata a variabili** di tipo "funzione"
- Deve poter essere **passata come argomento** a un'altra funzione
- Deve poter essere **restituita da un'altra funzione** perché sintetizzare un valore funzione non è così diverso da sintetizzare un valore di un altro tipo
- Deve poter essere **definita e usata "al volo"** come ogni altro valore (literal) di ogni altro tipo
- Magari anche senza avere per forza un nome…

Molti linguaggi a oggetti, nati su base imperativa, **stanno incorporando il concetto di funzione come first-class entity** (o qualcosa che le somigli):

- **Javascript**: da sempre ha il costruttore Function e la keyword function per specificare funzioni "anonime"
- **C#**: delegati e costruttore Func<…>. Per le chiusure offre lambda expression e metodi anonimi da vario tempo, esistono dei veri e propri “tipi funzione”
- **C++**: dallo standard 11, supporto per funzioni "anonime" e possibilità di passarle
- **Java**: supporto parziale introdotto solo in Java 8. Esistono funzioni anonime come **lambda expressions** si possono definire come literal e passare come argomenti, ma non sono vere first class entities, perché non definiscono un tipo. Tipo della lambda expression = la sua interfaccia funzionale

Passare una funzione come argomento a un'altra significa poter definire **FUNZIONI DI ORDINE SUPERIORE**:

- **Funzioni di secondo ordine**: manipolano funzioni
- **Funzioni di terz'ordine**: manipolano funzioni di secondo ordine
- Ecc. ecc.

Assegnare e restituire funzioni implica disporre di TIPI-FUNZIONE (o di qualcosa di simile che ne faccia le veci):

- Le funzioni sono valori (istanze, oggetti) di quel tipo/i → funzioni come dati manipolabili dinamicamente
- In più, sono eseguibili → incapsulano comportamento. Mentre prima pensavamo alla divisione tra segmento dati e segmento codice, ora stiamo mescolando le due cose perché abbiamo un dato funzione che è anche eseguibile quindi è un dato e allo stesso tempo incapsula un pezzo di codice!
- Si potrebbe perfino "sintetizzare un nuovo comportamento" ed eseguirlo subito dopo

Esempi in Javascript

Le funzioni hanno tipo <function>:
La keyword function definisce oggetti-funzioni *var f = function (z) { return z*z; }*
Qui non c'è il nome, è una function e basta. Dopo basterebbe fare f() e viene fatto quello che c'è dentro le {}

Una funzione può riceverne un'altra come argomento *var ff = function (f, x) { return f(x); }*
Si può passare una funzione come argomento a un'altra *loop(function(){ i++ }, 10)();*
Si può può restituire un risultato-funzione: *function ff(){return function(r){return r+10}}*
Qui nascono le chiusure di cui parleremo più avanti: gli argomenti a chi appartengono? e quanto vivono?

Si può perfino creare una nuova funzione a partire da testo *square = new Function("x", "return x*x")*
Questo è un alias. Function è un costruttore a n parametri: l'ultimo è il body della funzione, quello che metterebbe tra {}. Praticamente monta pezzi di stringhe, li interpreta e li rende codice eseguibile.
È fantastico essere aperti all'esterno, ma c'è totale insicurezza: se io scrivessi istruzioni strane, potrei ritrovarmi con il disco formattato o cose di questo tipo. Bisogna proteggere il proprio fortino, servono comunque dei confini.

17.6 – VARIABILI LIBERE E CHIUSURE

Se un linguaggio ammette la definizione di funzioni con variabili libere (cioè non definite localmente), tali funzioni dipendono dal contesto circostante. La necessità di criteri per dare significato alle variabili libere sfocia nel confronto tra **CHIUSURA LESSICALE vs CHIUSURA DINAMICA**.
Ci sono casi analoghi anche in linguaggi tradizionali:

- In funzioni C, variabili libere esistono solo con riferimento a variabili globali
- In funzioni Pascal, possono esistere anche funzioni innestate → necessità di distinguere fra catena statica e catena dinamica

Tuttavia, in entrambi i casi non ci sono grandi conseguenze perché le funzioni non sono first-class entities!

Una vera funzione nel senso matematico del tempo non dipende da nessun argomento, sa che ne deve avere 5 e basta. Se poi al suo interno ne usa 6 perché usa anche z che non è insieme agli altri 5 ma è esterno, allora c'è una sua definizione in quell'ambiente. Quella funzione così com'è finché non la attacco a qualcuno non può funzionare, è un componente open e va chiuso rispetto all'ambiente perché ha variabili libere: è una chiusura.

E se non avessi un solo z? Se io nella mia chiusura ho una z ma vengo chiamato da una funzione che sta da un'altra parte (che ha a sua volta ha una sua z), di quale "z" stiamo parlando?
Potremmo dire "la più vicina": ma vicina in che senso? Vicina temporalmente? L'ultima che mi ha chiamato? O vicina in tempo spaziale? Questi sono due modelli computazionali completamente legittimi e funzionano in modo completamente diverso e sono la chiusura lessicale e la chiusura dinamica.

Se il linguaggio supporta funzioni come first-class entities, la presenza di **variabili libere** determina la nascita della **nozione di chiusura**:

- **Un «oggetto funzione»** ottenuto chiudendo rispetto a un certo contesto una definizione di funzione che aveva variabili libere
- **Il prodotto finale dell'atto di chiudere le variabili libere** rispetto a un certo contesto d'uso

Esempio Javascript

```
function ff(f, x) { // crea una chiusura
  return function(r){ return f(x)+r; }
}
```

Invocando ff ottengo come risultato un oggetto-funzione che ha al proprio interno i riferimenti a "f" e "x".

Questa funzione prepara un comportamento che servirà poi perché io chiamo ff e le passo f, x. Questa prende la funzione f e prende x (che si suppone sia un dato tipo intero o stringa) e predispone una nuova macchina in quanto sintetizza una nuova funzione anonima function(r) che quando sarà chiamata farà f(x)+r.
Se nessuno dovesse chiamare quella funzione, niente. **Qui c'è un problema di tempi di vita**: io "f" e "x" te li passo adesso, ma "r" potrebbe essere inserito mooooooooolto più avanti.
Se metto "f" e "x" nello stack, questi esistono finché esiste ff, peccato che ff restituisce una nuova funzione con "r", quindi si avrebbe null, null, r. **Perché esistano le chiusure va completamente cambiata la gestione degli argomenti!**

Origine delle chiusure: linguaggi funzionali (Lisp, Scheme ~1960). Le chiusure sono disponibili da sempre nei linguaggi che hanno funzioni come first-class entities (Javascript, Scala, Ruby,…).
Più recentemente, anche in linguaggi OO di largo uso (C#, Java 8, e seguenti), nella forma di "**lambda expression**", spesso però con limitazioni che le depotenziano.
Es. i delegati C# sono una "forma blanda" di "tipo funzione": se vengono riprogettate le Collections API, promuovono stile di codice molto più sintetico (meno «boilerplate code») e favoriscono l'adozione di nuovi idiomi.

17.6.1 - TEMPO DI VITA

In presenza di chiusure, il TEMPO DI VITA delle variabili di una funzione non coincide più necessariamente con quello della funzione che le contiene, quindi il **modello computazionale deve tenerne conto**: quelle variabili non si possono più allocare "semplicemente" sullo stack! Questo perché:

- Una variabile locale a una funzione "esterna" può essere indispensabile al funzionamento della funzione più interna (della cui «chiusura» fa parte)
- Il tempo di vita delle variabili locali che fanno parte di una chiusura è pari a quello della chiusura, anche se la funzione che le definisce termina prima

Per questo, tali variabili non sono allocate sullo stack, ma **sull'heap** che un **garbage collector** provvederà poi a ripulire.

Esempio Javascript

```
function ff(f, x) { // crea una chiusura
  return function(r){ return f(x)+r; }
}
```

Questa funzione ff restituisce, ogni volta che viene chiamata, una nuova entità-funzione il cui corpo sfrutta f, x (oltre che r).

function g (f,x) { return f(x); }

Questa è una funzione di secondo grado, ma tutto sommato questa f(x) non è che faccia sta gran cosa, fa solamente una chiamata. È comunque qualcosa che in Java si farebbe fatica a fare, però non c'è nessuna chiusura perché una volta finite, basta, x dopo muore e non serve più. Supponiamo che dopo la funzione g io metta una print:

function g (f,x) { return f(x); } print(g(Math.sin, pi/3));

In questo momento faccio già qualcosa in più, ma sto comunque già eseguendo quello che voglio adesso.
Se invece guardo la funzione a sinistra:

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

Qui viene fatto qualcosa di diverso: qui viene preparato un macchinino che dice "restituisci una NUOVA function con argomento r". Quindi ff prende "f", "x" e produce una f(r) senza sapere cos'è "r": ha sintetizzato un nuovo ingranaggio ma non l'ha messo in moto.
Se nessuno adesso usa f(r), questo è un lavoro inutile, non farà nulla. Ma se qualcuno scrive "f(13)" ad esempio, allora sì che si scatena la bestia.

La chiamata di ff, allora, avviene in un tempo T1, mentre la chiamata f(r) avviene in un tempo T2: "f" e "x" devono sopravvivere FINO a T2, se no non si avrebbe nessun risultato intelligente (perché x e f sarebbero già morte e sarebbero undefined: valore che non ha valore). Il risultato totale allora sarebbe undefined ☹

17.6.2 - SCOPO DELLE CHIUSURE

Le chiusure permettono di modellare molte cose carine:

- **Rappresentare e gestire uno stato privato e nascosto** perché le variabili della funzione "esterna", mantenute vive dalla chiusura, restano comunque invisibili fuori da essa
- **Creare una comunicazione nascosta:** definendo più funzioni nella stessa chiusura, esse condividono uno stato nascosto, usabile per comunicare privatamente
- **Definire nuove strutture di controllo**: la funzione esterna esprime il controllo, mentre quella ricevuta come argomento esprime il "corpo da eseguire"
- **Riprogettare/semplificare API e framework di largo uso**: metodi parametrici che «ricevono comportamento» come argomento, anziché metodi specializzati → interfacce più semplici, pulite e leggibili

Ad esempio, tu mi hai dato "for" e "while", ma supponiamo che noi vogliamo avere una nuova struttura che fa un ciclo in un altro modo: in Java non possiamo farlo, possiamo creare un metodo statico che fa quel comportamento. **Nei linguaggi funzionali invece ciò che costruisci è indistinguibile da ciò che è built-in**: ma esistono veramente le cose built-in? No, sono semplicemente librerie pre-caricate.
In Javascript ci sono delle limitazioni, mentre in Scala, come vedremo, questa cosa è fatta molto bene: potenzialmente noi potremmo dire "caro utente, nella nuova versione c'è questa nuova struttura di controllo che si chiama loopUntil": non è vero, l'ho scritta io, ma l'utente non se ne accorgerà mai! Di "predefinito" non c'è niente, ci sono solo cose settate di default per dare qualcosa a chi usa Scala.

Esempio chiusure C# e Java8 → Lambda expressions

Notazione C#: { params => body } vs Notazione Java 8: (params) -> {body}

Alcuni esempi in C#...

```
{ int x => x+1 }
{ int x, int y => x+y }
{ string s => int.parse(s) }
{ => Console.WriteLine("hi")}
```

….e in Java8

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

- **params** è una lista di parametri formali con tipo, separati da virgole; può mancare nel caso di funzioni senza argomenti
- **body** è una lista di istruzioni, di cui l'ultima può essere una espressione (che denota il valore dell'intera chiusura)

SEMANTICA INFORMALE: l'esecuzione della chiusura crea un oggetto contenente il codice del body + il contesto lessicale.

ATTENZIONE perché in JAVA8 le variabili-chiusura sono supportare SOLO SE SONO READ-ONLY, cioè se sono effectively final. Questo per evitare effetti collaterali e accoppiamenti difficilmente prevedibili in presenza di multi-threading.

Esempio chiusure in Scala e Javascript

JavaScript, Scala, kotlin offrono funzioni come first-class entities: è possibile definire un generico function literal e se ci sono variabili libere, al momento dell'uso nasce una chiusura.

Notazione Scala: (params) => body

Scala è Object-Oriented → le funzioni sono oggetti di tipo (A, B, C, …) => R

Notazione kotlin: (params) -> body

Anche Kotlin è OO → le funzioni sono oggetti di tipo (A, B, C, …) => R

Notazione JavaScript: function() { body }

Anche Javascript è OO → funzioni sono oggetti di tipo <function> costruiti dal costruttore Function

17.6.3 - CRITERI DI CHIUSURA

Occorre **stabilire un criterio sul come e dove cercare una definizione per le variabili libere da chiudere**.
In presenza di funzioni innestate, si apre una questione:

- Da un lato, il testo del programma contiene fisicamente una catena di ambienti di definizione innestati **(CATENA LESSICALE)**
- Dall'altro, l'attivazione delle funzioni crea a run-time una catena di ambienti attivi **(CATENA DINAMICA)** che riflette l'ordine delle chiamate

Le due catene sono in generale diverse, quindi bisogna scegliere quale seguire: i risultati sono in generale anch'essi diversi. (Un problema simile esiste anche in C e Java dove, pur non essendoci funzioni innestate, esistono però blocchi innestati)

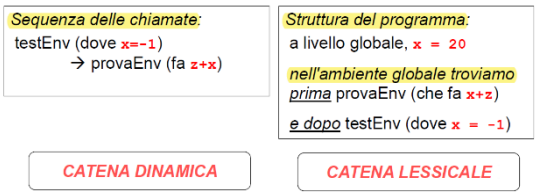
Le funzioni innestate sono di fatto funzioni dichiarate dentro altre funzioni. In C non si può fare, ed è una scelta voluta. La prima idea che viene in mente è quella di cercare nell'ambiente circostante più vicino: si segue lessicalmente il testo del programma: è una specie di "cipolla".
Tipicamente questo però non è l'ordine delle chiamate: è difficile che le chiamate si siano chiamate dalla più esterna alla più interna. Spesso e volentieri le funzioni si chiamano di qua e di là saltellando.
Guardando invece la variabile appartenente all'ambiente più vicino dinamicamente, cioè nello stack, allora si ha un altro tipo di approccio e si parla di catena dinamica.

```
var x = 20;
function provaEnv(z) {  return z + x; }
function testEnv() {
   var x = -1;
   return provaEnv(18);
}
```

Esempio Javascript

provaEnv ha una chiusura perché se viene invocata in un momento in cui non c'è x: errore a runtime. Altrimenti usa l'x più vicino a lei: ma vicino in senso lessicale o dinamico?
testEnv come variabile locale definisce una variabile x con un valore diverso da quella nella prima riga, che appartiene ad un ambiente più esterno. se faccio testEnv, chiamo provaEnv: allora quale "x" si usa?

Se si considera lessicalmente il testo del programma, la definizione di x "più vicina" (nel senso testuale, ossia "che precede l'uso") è quella in alto, cioè x=20. Sotto questa ipotesi, invocando testEnv sembrerebbe logico ottenere come risultato 18+20, ossia 38. (approccio C)



Se però si considera l'ordine delle chiamate le cose cambiano, perché la definizione di x più vicina (nel senso dinamico, ossia "più recente") è quella locale di testEnv, cioè x = -1. Sotto questa ipotesi, invocando testEnv sembrerebbe logico invece ottenere come risultato 18-1, ossia 17. (approccio Pascal)
Quale scegliere? Stabilire il significato di x in provaEnv seguendo la sequenza delle chiamate (dinamica) o la struttura del programma (lessicale) non è la stessa cosa!

	PRO	CONTRO
CHIUSURA LESSICALE	Si può leggere e capire un programma senza dover ricostruire la sequenza delle chiamate	Vincola a priori il simbolo ad un certo legame
CHIUSURA DINAMICA	Massima dinamicità: simboli legati in base alla specifica situazione del momento	Capire cosa fa il programma richiede di ricostruire la sequenza delle chiamate

Se ho un programma ENORME, nel modello computazione basato su chiusura lessicale basta isolare un pezzetto e sappiamo esattamente cosa fa e cosa succede. Non serve chiedersi cosa succederebbe se fossimo chiamati da qualcun'altro: quello che succede dipende solo da quello che c'è scritto intorno a me.
Nel modello computazionale basato su chiusura dinamica, invece, per sapere cosa succederà in un determinato momento, bisogna - mentalmente o simulando - far eseguire il programma fino a quel momento perché tutto dipende dall'ordine delle chiamate.

Poiché la comprensibilità è cruciale,
praticamente tutti i linguaggi di programmazione adottano il criterio di CHIUSURA LESSICALE.

17.7 – MODELLI COMPUTAZIONALI e VALUTAZIONE DI FUNZIONI

Quello che faremo ora sembra un'assoluta ovvietà, ma scopriremo che non è proprio una banalità, ovvero, cosa succede quando chiamiamo una funzione f con una certa lista di argomenti? Se dovessimo spiegarlo a qualcuno cosa diremmo?
Il controllo va da un'altra parte e i valori passati come argomenti vengono copiati nel contesto in cui siamo passati. Ciò ovviamente avviene prima che la funzioni cominci a girare. Poi il controllo passa alla funzione e il resto aspetta. Ad un certo punto arriva a "return 5": da lì si ritorna all'indietro ricopiando il 5 e finisce nel risultato del pezzo di codice precedente. A quel punto la funzione termina e il suo mondo muore e solo dopo il programma riprende da dove si era interrotto. Funziona sempre così? No.

Ogni linguaggio che introduca funzioni deve prevedere un modello computazionale per la loro valutazione.
Tale modello deve stabilire:

- QUANDO si valutano i parametri
- COSA si passa alla funzione
- COME si attiva la funzione (si cede il controllo?)

Tradizionalmente, **il più usato è il MODELLO APPLICATIVO:**

- QUANDO → i parametri si valutano subito, prima di ogni altra cosa
- COSA → di solito, si passa alla funzione il valore dei parametri (a volte il loro indirizzo)
- COME → si attiva la funzione chiamandola e cedendo il controllo → **modello sincrono**

PRO	CONTRO
• Pratico • Efficiente → valuta parametri 1 volta sola per ogni chiamata, passa dei valori/indirizzi • Leggibile/facile da capire (motivo della sua diffusione)	• Può causare inefficienze → a volte valutare i parametri non serve • Può determinare fallimenti non necessari se valutazione dà errori

Esempio Java e Javascript
Nel codice d'esempio, nella printOne o stampa il primo o stampa il secondo, non stampa tutti e due: però in questo modello tutti vengono valutati, poi al massimo non li uso.

Quando valido 3/0 viene sparata un'eccezione quindi il tutto muore, ma guardando la funzione sopra, siccome la condizione è verificata (x>y), avremmo stampato solo “a” (4) e non avremmo neanche guardando l'argomento “b” (che in questo caso non serve).

```
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
  }
}
```

Quindi, avremmo potuto evitare l'esplosione semplicemente ignorando la valutazione di tutti gli argomenti: se ho delle situazioni al limite come in questo caso, utilizzare l'approccio del "calcolo tutto così così poi stiamo dalla parte dei bottoni" porta ad un fallimento, che sarebbe stato evitabile.

Guardando il codice Javascript, l’abort è una schifezza, ma posso farla perché tanto non ho i tipi in Javascript: anche qui abbiamo un'esplosione per niente perché se ci fossimo limitati al primo argomento (verificato perché flag<10), avremmo stampato 5 e la funzione non avrebbe mai usato “x”. Invece, facendo la valutazione di tutti gli argomenti → abort.

```
var f = function(flag, x) {
 return (flag<10) ? 5 : x;
}
var b = f(3, abort() ); // Errore!!
document.write("result =" + b);
```

17.7.1 – MODELLO CALL-BY-NAME o NORMALE

Adotta un meccanismo diverso: meccanismo del burocrate pigro. Si fanno le cose solo quando sono proprio indispensabili. Come abbiamo già visto, questo approccio di "prendere tempo" ci permette di evitare disfatte: si aspetta per decidere cosa fare.

Il **Modello Normale** adotta un meccanismo di valutazione noto come Call-By-Name in cui:

- **I parametri non si valutano all'atto della chiamata**, ma solo al momento dell'uso e quindi solo se servono
- **Alla funzione si passano** non dei valori (o indirizzi) ma **degli oggetti computazionali** ("eseguibili")
- **Un parametro ricevuto può non essere mai valutato** se in quella invocazione non c'è effettivo bisogno di lui → si evita di fallire inutilmente

L'insieme di funzioni che terminano con successo con questo modello è più ampio rispetto al modello applicativo. Infatti, gli esempi di prima di Java e Javascript, con questo modello sarebbero andati a buon fine.

Nonostante il modello Call-by-Name sia così utile, **nei linguaggi di largo uso non è stato adottato perché è meno efficiente nei casi normali,** che sono la maggioranza, in quanto:

- **Valuta i parametri più volte per ogni chiamata** (si potrebbe ovviare valutandoli solo al primo uso)
- **Richiede più risorse a run-time**, dovendo gestire il passaggio di oggetti computazionali anziché semplici valori o indirizzi
- **Richiede una macchina virtuale capace di “lazy evaluation”** per catturare magari "pochi" (?) casi, che spesso sono errori di programmazione!
- **Molti casi di fallimento (quelli aritmetici) si evitano arricchendo l'aritmetica con NaN e Infinity**

Spesso il problema è che ti sei dimenticato di valutare una precondizione che è stata violata e quindi è colpa tua, programmatore, quindi è giusto che tu ti accorga dell'errore e che tu lo corregga. Altrimenti potresti scoprirlo più avanti.

Esempio Java VS Javascript

```
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
  }
}
```

```
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);
```

Ogni volta che c'è una situazione che non riesci a rappresentare puoi gestirla come errore oppure puoi farla rientrare nella normalità allargandone il concetto. Quindi, la divisione per zero puoi decidere se trattarla come errore oppure se allargare la tua "matematica" dicendo che darebbe come risultato "Infinity".
Infatti, a differenza di quanto ci si potrebbe aspettare, **questo esempio in Javascript non fallisce** perché per la macchina virtuale JavaScript la divisione 3/0 è un'operazione lecita con risultato "**Infinity**".

Il più noto ambito d'uso del modello Call-By-Name è quello delle **MACRO.**

```
#define f(FLAG, X) (((FLAG)<10) ? 5 : (X))
main(){
 float b = f( 3, abort()); // non dà errore!
}
```

Come negli esempi ipotetici rivisitati poco fa, guardando l'esempio di MACRO in C, il programma non dà errore perché abort non viene mai valutata.
La "chiamata" alla macro si espande infatti come: b = (((3)<10)? 5 : (abort()))
che, essendo la condizione sempre vera, dà come risultato 5 grazie alla valutazione in corto circuito dell'operatore ternario.

IL MODELLO CALL-BY-NAME PUÒ COMUNQUE ESSERE FACILMENTE SIMULATO nei linguaggi con funzioni come first-class entities. Per farlo basta fare a mano, artigianalmente, ciò che il modello call-by-name avrebbe fatto da solo, ossia, passare alla funzione non valori, ma opportuni oggetti computazionali che, quando eseguiti, producano i valori desiderati. **Per farlo è sufficiente:**

- **Sostituire ogni valore di parametro attuale con una funzione** che ritorni tale valore quando invocata
- **Sostituire ogni uso di un parametro formale dentro la funzione con una chiamata** alla funzione stessa

Se il linguaggio non ha funzioni come first-class entities, occorre simulare anche quel livello → alquanto contorto!

Esempio Javascript

L'esempio JavaScript originale (modello applicativo):

```
var f = function(flag, x) { // sono due valori
 return (flag<10) ? 5 : x;  // usa i due valori
}
var b = f(3, abort() );
document.write("result =" + b);
```

L'esempio ricostruito in stile call-by-name:

```
var f = function(flag, x) {    // diventano due funzioni
 return (flag()<10) ? 5 : x(); // invoca le due funzioni
}
var b = f( function(){return 3}, function(){ abort()} );
document.write("result =" + b);
```

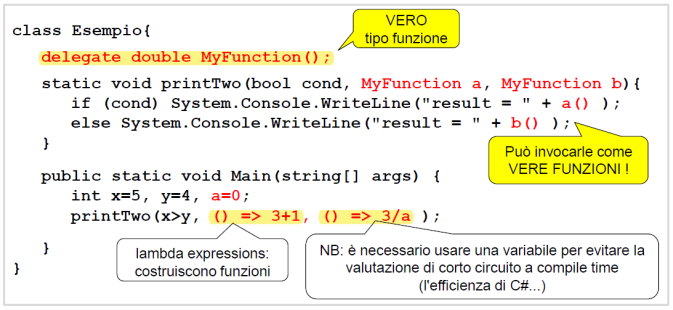
Ora funziona e ottengo che "result = 5".

In **Java 7**, le funzioni non sono entità di prima classe, quindi non è possibile creare una funzione e passarla. Si deve passare un oggetto di un'opportuna classe, appositamente definita e contenente un metodo che incapsuli la funzione. L'oggetto passato è un'istanza singleton di tale classe.
Conseguenza: pletora di classi (una per ogni funzione!) organizzate in gerarchia → complicatissimo e verboso

In **Java 8**, le lambda expression catturano funzioni, ma le funzioni non costituiscono un tipo autonomo. Il boilerplate code si riduce, ma solo da un lato (il chiamante).
Poiché non esiste un tipo-funzione autonomo, ogni lambda è vista in realtà come istanza di un'opportuna interfaccia funzionale e bisogna chiamare dei metodi con la loro esatta signature.

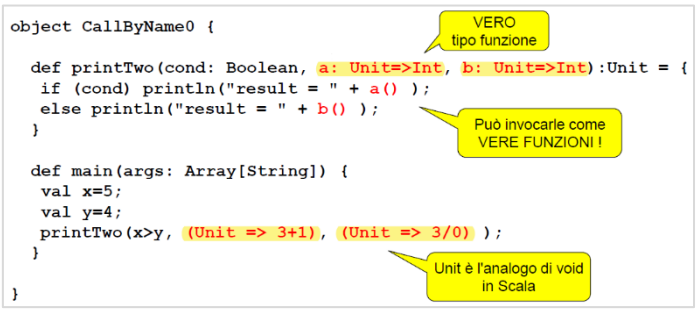
Perciò, in fase di chiamata si può scrivere una lambda, in modo che dal lato cliente si veda un mondo migliore, a basso costo, che è un vantaggio. Tuttavia, per lo sviluppatore di API non è così bello: lui sa cosa sta succedendo perché ci lavora, ma non è immediato. Se il programmatore è particolarmente skilled, ok, e l'utente medio va bene che usi la freccina e pace: è un compromesso.

In **C#**, le lambda expression catturano funzioni e i delegati esprimono veri "tipi funzione" completando l'opera → abbiamo piena espressività da ambo i lati (chiamante e chiamato).



Nell'esempio, il delegato ammette () e chiama da solo il metodo sottostante che fa il lavoro e io non devo sapere come si chiama.
Alla fine non c'è scritto 3/0 ma c'è scritto 3/a perché se no lo compila a compile-time (compilatore troppo ottimizzato e se ne accorge subito che non va bene, non a runtime)
Con questo programma, otteniamo il risultato 4.

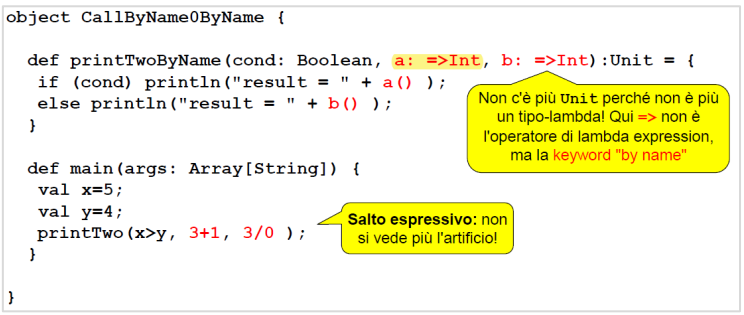
Anche in **Scala** si può ricostruire la Call-By-Name come in C#. Nell'esempio a e b sono funzioni da void a int, ma void è Unit: quindi sono funzioni da Unit a Int. Siccome sono funzioni, le chiami con () e il cliente passa la funzione.



In realtà, in Scala non serve ricorrere a tutto ciò perché il linguaggio supporta direttamente la call by name!

- Gli argomenti da passare by name vanno etichettati con =>
- Il runtime di Scala provvederà a gestire le lambda sotto banco, mascherando all'utente il livello

Si ha quindi un **netto salto di espressività**: l'utilizzatore (chiamante) della funzione non deve catturare i valori in funzioni, può passarli apparentemente come semplici valori.



18 – MULTI-PARADIGM PROGRAMMING with JAVASCRIPT

Javascript unisce potenza espressiva a un modello computazionale potente e flessibile perché:

- **Adotta un approccio funzionale** con funzioni e chiusure semplice e pratico, con una **sintassi leggera**
- Adotta un **modello object-based senza classi**, basato sul concetto di prototipo
- **È un linguaggio interpretato**, con aspetti dinamici di grande interesse → ottima palestra per sperimentare
- È **adatto a creare applicazioni multi-paradigma**

Tuttavia, presenta alcune **limitazioni**:

- Incorpora alcuni "errori di giovinezza"
  - Passò dalla "non esistenza" all'ampia adozione in (troppo) poco tempo: non ci fu il tempo per "ripulirlo"
  - **Un ambiente globale** (mal definito) **con variabili globali**
  - Alcuni operatori contorti
  - **Richiede disciplina** per evitare stili che non rispettano le buone pratiche
- Offre **alcune caratteristiche controverse**
  - Il **LOOSE TYPING** riduce le possibilità di intercettare errori di tipo, ma è potente, molto flessibile, ed evita i cast. D'altronde, anche lo strong typing non elimina la necessità di testing…
  - L'**ereditarietà prototype-based** è concettualmente potente, ma non facile da applicare per chi viene da linguaggi "class based"

18.1 – LE BASI DEL LINGUAGGIO

Javascript è un **linguaggio di scripting** inventato da Netscape, inizialmente denominato LiveScript, rinominato in JavaScript nel 1995 dopo un accordo con Sun. Esiste la variante Microsoft: JScript (minime differenze).
È diventato standard in rapida evoluzione: da ECMAScript 262 a ECMAScript 9 (2018).
È Object-based (ma non object-oriented), quindi simile a C, Pascal, Java (ma di Java ha piuttosto poco); tuttavia **per molti versi assomiglia più a linguaggi funzionali.**

I **MOTORI JAVASCRIPT** si trovano nei browser, come virtual machine stand alone o come API embedded in altre piattaforme/linguaggi. I più diffusi sono:

- **V8** (Google): stand alone
- **JScript** (Microsoft): non stand alone
- **GraalVM** (Java 11+): stand alone, integrato con Java
- **Nashorn** (Java 8-11): non stand alone, integrato con Java (incluso nel JDK8 ma deprecato da Java11)
- **Rhino** (Java<8): stand alone
- **SpiderMonkey** (Mozilla): stand alone

Per inserirlo nelle pagine web, basta usare il tag <**SCRIPT**> dentro il body:

```
<SCRIPT Language="JavaScript">
        <!-- Per non confondere i vecchi browser
        … IL PROGRAMMA JAVASCRIPT …
        // Commento Javascript per non confondere browser -->
</SCRIPT>
```

18.1.1 – ELEMENTI LINGUISTICI

- **STRING**: il tipo string denota stringhe di caratteri Unicode
  - Come in Java, ogni stringa è un oggetto immutabile dotato di proprietà e metodi
  - Come in Java, l'operatore + concatena (occhio ai numeri…)
  - **CHAR NON ESISTE:** un carattere è una stringa lunga 1
- **COSTANTI:** sono delimitate da virgolette o apici (per annidare virgolette e apici, occorre alternarli)
- **NUMBER:** denota un reale a 64bit.
  - **Non ci sono interi** (niente overflow, niente conversioni)
  - La divisione è sempre fra reali, anche con operandi interi
  - Gli operatori bitwise operano su interi ottenuti convertendo sul momento il valore reale: sono lentissimi e inefficienti → evitare
- **COSTANTI NUMERICHE**: sequenze di caratteri numerici non racchiuse da virgolette o apici
  - La costante **NaN** rappresenta il "risultato" di operazioni matematiche impossibili: non è uguale a nulla, incluso lei stessa
  - Un'operazione che coinvolga un NaN dà come risultato NaN
  - La costante **Infinity** rappresenta un valore maggiore del massimo reale positivo rappresentabile ($1.79 * 10^{+308}$)
- **COSTANTI BOOLEAN**: true e false. Attenzione agli operatori che valutano condizioni…
- **COSTANTE NULL**
- **COSTANTE UNDEFINED**: indica un valore indefinito ed è restituito da funzioni che non restituiscono nulla. Inizialmente tutte le variabili non inizializzate sono undefined
- **COMMENTI**:
  - Singola riga //
  - Su più righe /* */

18.1.2 – ESPRESSIONI

Le espressioni sono gestite sostanzialmente come in Java:

- **Espressioni numeriche**: somma, sottrazione, prodotto, divisione (sempre fra reali), modulo, shift, …
- **Espressioni condizionali** con ? … :
- **Espressioni stringa**: concatenazione con +
- **Espressioni di assegnamento**: con = (e sue varianti)

Esempi document.write(18/4) //fra reali document.write("paolino" + 'paperino')

18.1.3 – VARIABILI

Le variabili in Javascript sono loosely typed. È possibile assegnare alla stessa variabile prima un valore di un tipo, poi un altro tipo. Sono consentiti incrementi, decrementi e operatori di assegnamento estesi (++, +=, …)
Esempio alfa = 19 beta = "paperino" alfa = "zio paperone" //tipo diverso!!

La **DICHIARAZIONE DI UNA VARIABILE** può essere:

- **Implicita** (la si usa e basta) pluto = 18
- **Esplicita** (con la parola chiave var) var pippo = 19

Lo **SCOPE** delle variabili è:

- **Locale**, in caso di dichiarazione esplicita dentro a funzioni
- **Globale**, in tutti gli altri casi

A differenza di Java, un blocco non delimita uno scope: variabili definite dentro a blocchi innestati sono comunque riferite all'ambiente che le contiene!

Il TIPO di una variabile **non è fissato a priori**: dipende dal contenuto attuale della variabile. L'operatore **typeof** restituisce il tipo di un'espressione:

- a=18; typeof(a) dà number
- a="ciao"; typeof(a) dà string
- typeof(18/4) dà number
- typeof("aaa") dà string
- typeof(false) dà boolean
- typeof(document) dà object
- typeof(document.write) dà function
- typeof([1,2,3]) dà object anziché array (!!!)

18.1.4 – ISTRUZIONI

Le istruzioni sono separate o da un punto e virgola (come in C o Java) **o da un fine riga** (come in Pascal).
Se si va a capo dove non si deve, c'è il rischio di **SEMICOLON INSERTION**, cioè comunque Javascript pensa che sia un'istruzione. Quindi ad esempio la stringa "multi riga" non funziona.

18.1.5 – STRUTTURE DI CONTROLLO

Ci sono le solite strutture: if, switch, for, while, do/while…
In più ci sono dei costrutti per operare su oggetti:

- **for … in …** → non equivale al foreach di Java poiché itera sui nomi degli elementi, non sugli elementi. Per gli array, i nomi degli elementi sono gli indici (0,1,2…) → caratteristica utile per gli array associativi, che però può risultare fuorviante se sconosciuta o inattesa
- **with**

il for classico

```
v = [13,24,37];
for (i=0; i < v.length; i++)
  document.write( v[i] );
// stampa 13,24,37
```

il for…in

```
v = [13,24,37];
for ( item in v )
  document.write( item );
// stampa 0,1,2
```

Esempio
Nel for…in viene stampato il numero degli indici e non il contenuto dell'array.

18.1.6 – OPERATORI

Esistono i soliti operatori relazionali (==, !=, >, <, >=, <=) e logici AND (&&), OR (||), NOT (!).
Ci sono però un paio di problemi:

- **PROBLEMA 1**: nella valutazione **si considera falso** non solo il valore false, ma **ogni valore falsy** ovvero anche null, undefined, la stringa vuota (''), il valore 0 ed NaN. Ogni altro oggetto, inclusa la stringa 'false', è vero.
- **PROBLEMA 2**: "==" e "!=" applicano **type coercion** secondo regole innaturali, con risultati talora incomprensibili → sono chiamati **"The evil brothers"**

The good brothers sono invece i nuovi operatori "===", "!==": offrono una alternativa più sensata al comportamento discutibile dei due operatori classici "==", "!=".

Esempio – The evil brothers

- 0 == '' → true, perché sono entrambi falsy values ☹
- 0 == '0' → true, perché 0 è coercibile a '0' ☹
- false == 'false' → false, come è giusto che sia 😊

- false == '0' → true, perché sono due falsy ☹
- false == undefined → false (ok, accettabile)
- false == null → false (ok, accettabile)
- null == undefined → true … Fulgido esempio di coerenza ☹
- ' \t\r\n' == 0 → true ☹

18.2 – IL LATO FUNZIONALE

Le funzioni sono introdotte dalla keyword **function:**

- **Il nome è opzionale**: JavaScript ammette anche funzioni anonime, analoghe alle lambda-expression
- **Function expression vs. function declaration**
- Possono non restituire nulla (procedure)
- Non si usa la keyword void
- I **parametri formali sono privi di dichiarazione di tipo** Es. function sum(a,b) { return a+b }
- Sono invocate tramite il classico operatore di chiamata **()**, fornendo i parametri attuali
    - Se i tipi dei parametri attuali non hanno senso, **errore a runtime**
    - A differenza di Java, **i parametri attuali possono non essere tanti quanti i parametri formali:**
        - **Se sono di più** → quelli extra sono ignorati
        - **Se sono di meno** → quelli mancanti sono undefined

18.2.1 – FUNZIONI COME FIRST-CLASS ENTITIES

In Javascript, una funzione è un'entità-oggetto, pienamente manipolabile come ogni altro tipo di dato:

- Può essere assegnata a variabili — `var f = function(x){ return x/10; }`
- Può essere definita e usata "al volo" con un function literal — `var z = function(y){return y+1;}(8); // z=9`
- Può essere passata come argomento a un'altra funzione — `var p = ff(f);`
- Può essere restituita da una funzione "generatrice" — `var f = fgen(…); … var z = f(3);`

18.2.2 – FUNCTION EXPRESSION VS DECLARATION

La **FUNCTION EXPRESSION** **assegna una funzione a una variabile:**

- Il **nome** della funzione è **inessenziale** (c'è la variabile)
- Lo **scope** del nome, se presente, è il **corpo della funzione** (utile solo per eventuali chiamate ricorsive)

Esempio `var f = function g(x){ return x/10; }` `g(32) // ERRORE: il nome "g" è qui indefinito`

La **FUNCTION DECLARATION** introduce una **funzione senza assegnarla a una variabile:**

- Il **nome** è **essenziale** (è il solo modo per riferirsi ad essa)
- Lo **scope** è l'**ambiente di definizione della funzione**

Esempio `function g(x){ return x/10; }` `g(32) // il nome g è noto nell'ambiente`

Esempi

```
var f1 = function (z) { return z*z; }       // f. expr.
var f2 = function g(x){ return x/10; }      // f. expr.
var f3 = f2;       // aliasing
var r1 = f1(7);    // 49
var r2 = f2(36);   // 3.6
```

```
var r3 = function(y){return y+1;}(8);       // 9
function calc(f, x) { return f(x); }        // f.decl.
var r4 = calc(Math.sin, .8)         // 0.7173560908995228
var r5 = calc(Math.log, .8)         // -0.2231435513142097
var r6 = calc(f1, .8)               // 0.6400000000000001
var r7 = calc(f2, .8)               // 0.08
var r8 = calc("x*x", .8)            // NO! non è una funzione, serve una funzione come nell'esempio sotto
var r8 = calc(function(r){return r*r*Math.PI}, .8)          // 2.0106192982974678
```

18.2.3 – FUNZIONI INNESTATE E CHIUSURE

A differenza di C e Java, **si può definire una funzione dentro un'altra → chiusure**

- Molteplicità di ambienti lessicali definiti uno dentro l'altro
- Molteplicità di ambienti in essere a run-time (modello a run-time)
- Variabili di chiusura allocate in heap (tempo di vita ≠ funzione)
- **Criterio di chiusura delle variabili libere: chiusura lessicale**

Nasce una chiusura quando una funzione innestata fa riferimento a variabili della funzione esterna: l'entità-chiusura nasce a runtime all'atto della chiamata della funzione "generatrice" esterna, ma sopravvive al suo termine, perché occorre **mantenere vive le variabili ancora referenziate** (necessarie all'esecuzione successiva).

Esempio

Definizione funzione esterna "generatrice": restituisce una nuova funzione a un argomento (r) che contiene riferimenti a variabili della funzione esterna (f, x).

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

A ogni invocazione di ff nasce una nuova chiusura:

```
var r9 = ff( Math.sin, .8)(3)                    // 3.71735609089952
var r10 = ff( function(q){return q*q}, .8 )(0.36)  // 0.64 + 0.36 = 1
```

Le due variabili della funzione esterna (f, x) sopravvivono al termine di ff, altrimenti la chiamata alla funzione restituita esploderebbe.

CHIUSURE IN JAVASCRIPT

1) RAPPRESENTARE UNO STATO PRIVATO E NASCOSTO:

In Javascript, le proprietà degli oggetti sono pubbliche. Si può ottenere una proprietà privata tramite una chiusura, mappando lo stato su un argomento della funzione "generatrice".

Esempio

```
function incBy2From(x) {
        return function(){ return x+=2;}
}
```

Ora, ad ogni invocazione di incBy2From nasce una nuova funzione:

- var getNextEven = incBy2From(0) → incorpora uno stato che evolve da x = 2,4,6..
- var getNextOdd = incBy2From(1) → incorpora un diverso stato che evolve da x = 3,5,7..

Invocando più volte getNextEven o getNextOdd si ottengono via via i successivi valori del SUO stato interno nascosto.

Esempio

```
function genContatore(){
        var contati=0;
        function tick() { return contati++; }
        function num() { return contati;}
        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
```

2) 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 restituirli entrambi, si può restituire un array di due funzioni.

Esempio

```
function myChannel() {
        var msg = "bla bla";
        return [ function(x) { msg = x; }, // set
        function() { return msg; } ]; // get
}
```

```
var channel = myChannel()
var msg1 = channel[1]();   // recupera "bla bla"
channel[0]("buuh");        // setta il nuovo msg
var msg2 = channel[1]();   // recupera "buuh"
```

Ad ogni invocazione di myChannel nasce un nuovo canale privato accessibile solo ai due metodi restituiti da myChannel stessa.

Esempio

Si può anche restituire un object literal con due accessor.

```
 function canale(){
        var msg = "";
        function set(m) { msg=m;}
        function get() { return msg; }
        return { set, get };
 }
```

```
var ch = canale();
document.writeln(ch.set("hello")); // undefined
document.writeln(ch.set("world")); // undefined
document.writeln(ch.get()); // world
```

3) REALIZZARE NUOVE STRUTTURE DI CONTROLLO

Una funzione di secondo ordine incapsula il controllo, argomenti-funzione rappresentano le azione da svolgere.

```
function loop(statement, n) {
  var k = 0;
  return function iter() {
     if (k<n) { k++; statement(); iter(); } }
}
```

nome necessario per la chiamata tail-ricorsiva

Esempio

Ad ogni invocazione di loop nasce un nuovo ciclo, che ripete quello specifico statement (una funzione!) per esattamente n volte, ma **l'operatore () rivela che loop non è un vero costrutto built-in** ☹ (in Scala rimedieremo)

```
var i=1; loop( function(){i++}, 10)(); //ritorna 11
```

Esempio

```
function loop(n){
 return function statement(action){
   if(n>0) { action(); n--; statement(action); }
 }
}
```

Variante (con currying): la funzione esterna cattura solo il numero di iterazioni, mentre l'azione da svolgere è specificata nella funzione interna. L'uso esplicito di function() rivela anche qui che loop5 non è un vero costrutto built-in.

```
loop(5)( function(){ document.writeln("ciao"); })
```

CHIUSURE E BINDING DELLE VARIABILI

Una chiusura = una istanza delle sue variabili → da tenere presente se si sfrutta una variabile di chiusura per creare e restituire ciclicamente più funzioni. **Rischio di mal-interpretare il significato delle variabili!**

Esempio: creare un array di funzioni che restituiscano numeri diversi (ad es. 4,5,6).
Idea sbagliata: ogni funzione restituisce l'indice del ciclo

```
function fillFunctionsArray(myarray){
        for (var i=4; i<7; i++)
        myarray[i-4] = function(){ return i;};
} //stampa 7, 7, 7
```

Peccato che ci sia una sola variabile i nella chiusura e tutte le funzioni restituiscono il valore finale di i (7) ☹
L'errore viene dalla mia interpretazione di i come left-value: è quello il problema. Non ho 4 "i" diverse, ne ho solo una e quindi metto sempre quella dentro all'array, alla fine del ciclo.

Per catturare il valore di una variabile che cambia occorre "fotografarlo" in una variabile ausiliaria: **serve una funzione intermedia**, il cui argomento faccia da variabile tampone per la variabile di chiusura che cambia.
SOLUZIONE: ogni funzione è una istanza della funzione ausiliaria che cattura un dato valore della variabile di ciclo:

```
function fillFunctionsArray(myarray){
        function aux(j){ return function(){ return j;}}
        for (var i=4; i<7; i++)
        myarray[i-4] = aux(i);
} //stampa 4,5,6
```

Stavolta, pur essendoci una sola variabile i nella chiusura, ci sono tante variabili j quante le funzioni ausiliarie: ogni j fotografa uno dei valori che i assume nel tempo. 😊

18.3 – IL LATO A OGGETTI

Javascript adotta un modello ad oggetti ma in un senso un po' diverso dal solito: è basato sull'idea di oggetto ma non la porta al punto di avere le classi, quindi le relazioni tra classi non hanno granché senso.
Si ha una sintassi abbastanza familiare a Java e quindi fa pensare che ci sia qualcosa di simile, invece dietro c'è qualcosa di diverso.

Javascript adotta un modello object-based (non object-oriented), senza classi, basato sulla nozione di prototipo. In questo approccio, **un oggetto:**

- **Non è istanza di una classe** (non esistono classi!)
- **È solo una collezione di proprietà** (dati o funzioni) **pubbliche**, accessibili tramite "dot notation"
- **È costruito, tramite new**, da un costruttore che ne specifica la **struttura iniziale** → il nome del costruttore è "tutto ciò che resta" delle classi
- È **associato dal costruttore a** un "oggetto padre", detto **prototipo**, di cui eredita le proprietà → **PROTOTYPE-BASED INHERITANCE**

Gli oggetti non appartengono alle classi perché non esistono, però possono assomigliare ad altri oggetti che vengono presi come prototipi. Quindi, si usa una sorta di ereditarietà, ma non fra classi → fra oggetti.
Un oggetto nasce con la new, ma non possiamo interpretarlo nello stesso modo di java perché non faccio new "classe". Dopo la new, qui, si può mettere un costruttore: è un ponte tra i due mondi. Tutto è basato sull'idea di funzione perché stiamo parlando di un linguaggio funzionale.

18.3.1 – OBJECT LITERALS

Un primo modo di specificare oggetti è la forma object literal, elencandone le proprietà in termini di coppie **nome:valore.** In questa modalità:

- Si possono anche definire metodi, tramite function
- I nomi delle proprietà possono essere stringhe qualunque (anche illecite come identificatori), purché siano messe fra virgolette

Esempi

{ x:10, y:7 }; p3 = { x:10, y:7, getX:function(){return this.x;} }; p4 = { "for-me": 10, "for you": "bleah" };

Da questo this capisco che sono in un linguaggo object-based perché si suppone che faccia riferimento ad una istanza.

18.3.2 – COSTRUZIONE DI OGGETTI

Un **costruttore** è una normale funzione, che però **specifica le proprietà e i metodi dell'oggetto da creare** mediante la parola chiave **this**, con ciò distinguendole dalle proprietà e dai metodi della funzione stessa.
Si distingue perché **invocata indirettamente**, tramite l'operatore new.

Esempio

```
Point = function(i,j){
  this.x = i; this.y = j;
  this.getX = function(){ return this.x; }
  this.getY = function(){ return this.y; }
}
```

```
p1 = new Point(3,4);
p2 = new Point(0,1);
```

Se scrivo:

function test() {**Point** = function (i,j) { this.x = i; this.y = j;}
p1 = new **Point**(3,4);
println(p1); //stampa [object Object]

Point è globale, può essere usato fuori dalla funzione test. Se invece al posto di Point, uso Pippo, sia dentro test, sia facendo new Pippo, eccezione perché Pippo è locale, non esiste fuori dalla funzione test.

18.3.3 – PROPRIETÀ

Poiché tutte le proprietà sono **pubbliche**, sono anche **liberamente modificabili**. In realtà, esistono anche alcune proprietà di sistema, non visibili né enumerabili con gli appositi costrutti.

Esempio WITH

Per accedere a più proprietà *di uno stesso oggetto* in modo rapido si può usare il costrutto `with`:

`with (p1) x=22, y=2` *equivale a* `p1.x=22`*, etc*
`with (p1) {x=3; y=4}` *equivale a* `p1.x=3`*, etc*

Un oggetto non è più un'istanza di una classe, qui sei autonomo, libero, indipendente: se sei nato con due coordinate, perché non puoi averne una terza? Aggiungiamola!
Infatti, le proprietà specificate nel costruttore non sono le uniche che un oggetto può avere: sono quelle iniziali. È possibile **AGGIUNGERE dinamicamente nuove proprietà** semplicemente nominandole e usandole. Ciò è possibile proprio perché non esiste il concetto di classe come "specifica di struttura" (fissa) di una collezione di oggetti. Per lo stesso motivo, è anche possibile **RIMUOVERE dinamicamente proprietà**, mediante l'operatore **delete**.

Esempio

```
p1.z = -3;        // da {x:10, y:4} diventa {x:10, y:4, z: -3}
delete p1.x       // da {x:10, y:4, z: -3} diventa {y:4, z: -3}
```

CONSTRUCTOR

Ogni oggetto tiene traccia del costruttore che lo ha creato mediante la **proprietà constructor.** Attenzione perché **l'oggetto può nel frattempo essere cambiato:** potrebbe avere più o meno proprietà di quelle specificate!

PROPRIETÀ E METODI DI CLASSE

I costruttori Javascript riflettono le "categorie" che in altri linguaggi sarebbero state rappresentate con classi. **Le proprietà "di classe" possono essere quindi modellate come proprietà della funzione-costruttore**:

Infatti, poiché una funzione è (anche) un oggetto, può avere le sue proprietà (come ogni oggetto). È quindi naturale pensare che le proprietà comuni a tutti i Point possano essere ospitate nell'unico oggetto che essi hanno in comune, ossia proprio la funzione-costruttore omonima.

PROPRIETÀ PRIVATE

Le proprietà in generale sono pubbliche, ma le **proprietà private possono essere modellate tramite chiusure, con apposite funzioni-accessor**.

```
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
```

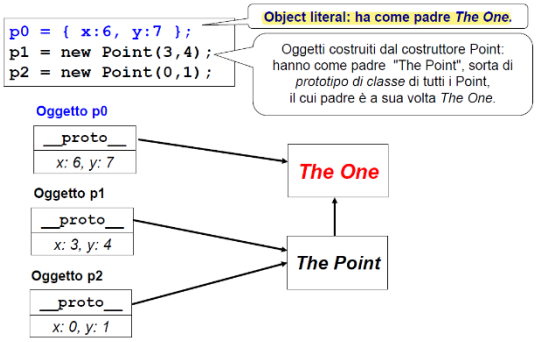
18.3.4 – PROTOTIPI DI OGGETTI

Ogni oggetto è associato a un "oggetto padre", detto prototipo, di cui eredita le proprietà. Ciò dà luogo a una forma particolare di ereditarietà, senza classi, detta **prototype-based inheritance** con le relative "gerarchie di ereditarietà prototipale".
Ogni oggetto referenzia il suo prototipo tramite una proprietà nascosta, chiamata **__proto__.** Chi sia esattamente l'oggetto padre di un dato oggetto dipende da chi ha costruito l'oggetto.

Esiste però un **antenato comune a tutti**, le cui proprietà sono ereditate da tutti: lo chiameremo (per ora) **"THE ONE"**. Esso è antenato diretto di ogni "object literal" e antenato indiretto di ogni altro oggetto.
Gli oggetti costruiti da un dato costruttore hanno infatti come padre una sorta di "prototipo di classe" che a sua volta ha come padre "The One".

Esempio

Che p0 sia un point lo diciamo noi, lui è un oggetto con due proprietà e basta. Chi è suo padre? "The One".
p1 e p2 nascono uguali, poi possono diventare qualsiasi cosa, ma nascono uguali.
"Point" deve dare un padre a p1 e p2: non può dire che derivano direttamente da "The one", ma ha predisposto un livello intermedio che faccia da punto comune a tutti i Point, che è molto simile al concetto di classe. Allora abbiamo The Point, che è il padre di tutti i punti, ed ha The One come padre.

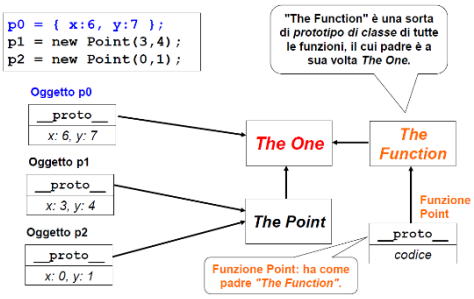


TASSONOMIA DEI PROTOTIPI

I prototipi sono il mezzo offerto da Javascript per esprimere relazioni e "parentele" fra oggetti.

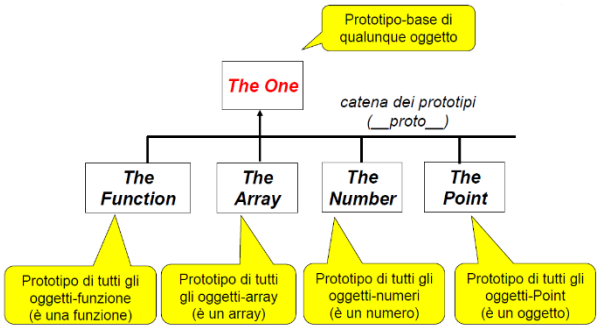
Esiste un prototipo predefinito per ogni "categoria" di oggetti: funzioni, array, numeri… Point, Persona, …
Noi chiameremo "The Function" il padre di tutte le funzioni, "The Number" il padre di tutti i numeri, etc. etc.
Ma **questi prototipi sono anch'essi degli oggetti**, quindi hanno a loro volta un prototipo: The One!
Da qui nasce la **TASSONOMIA DI PROTOTIPI** che esprime l'ereditarietà nella forma prototype-based.

Esempio



Il costruttore Point ha ipotizzato come padre il sig. The Point che supponiamo abbia come padre The One.
Ma nella figura di prima non c'era il costruttore, che è un oggetto come gli altri: quindi il signor Point è una funzione, che è un cittadino di prima classe, quindi anche lui è un oggetto con le sue proprietà. E chi è suo padre? The Function. E il padre di The Function? The One.
Si crea un intreccio di roba che nella nostra testa stava a livelli diversi, ma qui tutto è object-based, poi sei tu che ti devi ricordare chi ha quale ruolo.

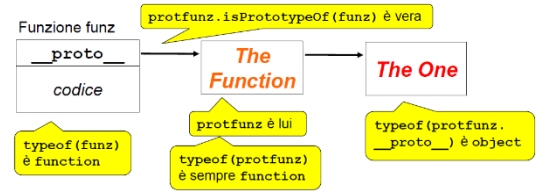
Ecco perché Javascript non è usato tipicamente per fare programmi grossi: ci vuole una disciplina ferrea per non confondersi perché altrimenti viene fuori una roba orrenda da debuggare.



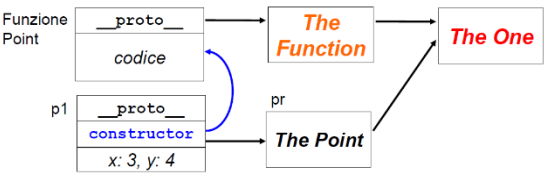
Esempi - Chrome

Il motore Javascript di Google Chrome mostra anche la proprietà __proto__, che altri nascondono: perfetto per sperimentare! Usiamo l'ambiente JScript IDE disponibile sul sito del corso e sfruttiamo l'operatore **typeof** per scoprire il tipo di un oggetto e l'operatore **isPrototypeOf** per risalire la catena dei prototipi.
Supponiamo che function siano funzioni, anche se non è detto. In Point c'è quanto visto negli esempi precedenti.
In questo sistema, il padre di tutte le funzioni, da noi chiamato affettuosamente "**The Function**", è la simpatica funzione function **Empty(){}**.

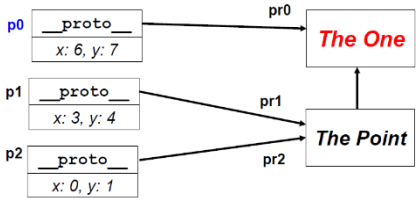
```
funz  = function(x,y){ return x+y; }
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
typeof(funz)                          function
protfunz = funz.__proto__
typeof(protfunz)                      function
typeof(protfunz.__proto__)            object
protfunz.isPrototypeOf(funz)          true
```




```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p1 = new Point(3,4)
pr = p1.__proto__
typeof(p1)                            object
typeof(pr)                            object
pr.isPrototypeOf(p1)                  true
p1.constructor                        Point
```



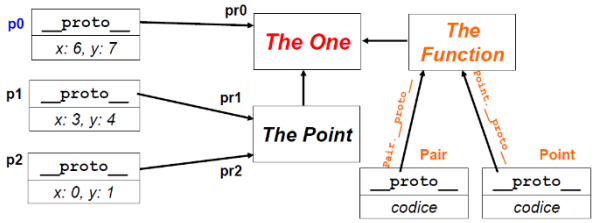

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p0 = { x:6, y:7 };   pr0 = p0.__proto__;
p1 = new Point(3,4); pr1 = p1.__proto__;
p2 = new Point(0,1); pr2 = p2.__proto__;
pr1 == pr2                            true
pr0 == pr2                            false
pr1.__proto__ === pr0                 true
```



Il nonno di p1 (quindi anche di p2) è quello di p0 quindi the one? Sì.

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
Pair = function(a,b){ this.a=a; this.b=b; }
p0 = { x:6, y:7 };   pr0 = p0.__proto__;
Pair.__proto__ === Point.__proto__
Point.__proto__.__proto__ === pr0
Pair.__proto__.__proto__  === pr0
```

```
true
true
true
```



PROTOTIPO DI COSTRUZIONE - PROTOTYPE

Il povero costruttore quando costruisce p1, come può affibbiargli il padre? Che tecnica usa? Il costruttore è configurabile, ha i suoi default, ma si può modificare.

Il prototipo di un oggetto dipende da chi lo costruisce: gli oggetti costruiti da uno stesso costruttore condividono lo stesso prototipo (sorta di "prototipo di classe"…).
Il prototipo che un costruttore attribuisce agli oggetti da lui creati è espresso dalla **proprietà prototype:**

- **prototype è una proprietà (pubblica) dei soli costruttori**
- Non va confusa con la proprietà __proto__, che invece è caratteristica di qualunque oggetto (e non è pubblica)

Per ricordarne lo scopo, la proprietà prototype è spesso chiamata prototipo di costruzione.

Il problema è che fino ad un certo punto un costruttore può avere un padre e poi ad un certo punto può averne un altro. Questo significa che un oggetto fino ad un certo punto nel programma fa qualcosa, poi fa qualcos'altro. Qui tutto quello che è implicito va reso esplicito e può cambiare runtime.
Non ho come in java la gerarchia delle classi che quella è e quella rimane per tutto il tempo di vita del programma.

È tramite la proprietà prototype che si riescono a referenziare (indirettamente) gli oggetti predefiniti "The Function", "The Point", "The One", etc. Infatti, se p è un oggetto creato dal costruttore Point, per definizione stessa di prototype: **p.__proto__ == Point.prototype**

Esempio

Detto Object il costruttore degli oggetti, Object.prototype è "The One", l'antenato comune (innominato) di tutta la tassonomia JavaScript! **Object è il costruttore dei literal, Object.prototype è invece The One.**
Ma attenzione: Object in sé è un costruttore, ossia una funzione il cui __proto__ è quindi "The Function", non Object! Attenzione a non perdersi coi nomi…

COSTRUTTORI PREDEFINITI

In particolare:

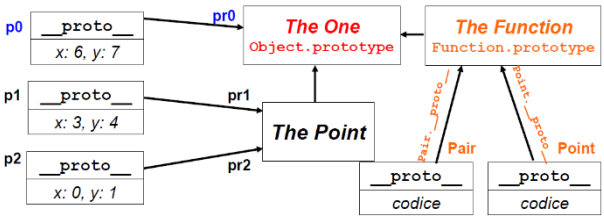
- Le funzioni sono costruite dal costruttore Function, che usa come prototipo "The Function"
  → **"The Function" è Function.prototype**
- Gli array sono costruiti dal costruttore Array, che usa come prototipo The Array"
  → **"The Array" è Array.prototype**
- Gli object literal sono costruiti dal costruttore Object, che usa come prototipo l'oggetto "The One"
  → **The One" è Object.prototype**

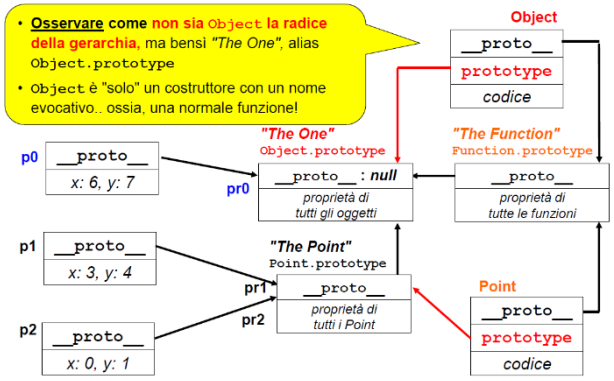
... e così via!

Esempi

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p0 = { x:6, y:7 };   pr0 = p0.__proto__;
Object.prototype === pr0
pr0.__proto__ == null
Function.prototype === Point.__proto__
Object.__proto__ === Function.prototype
Point.prototype == pr1
```

```
true
true
true
true
true
```





PROTOTIPI ED EFFETTI A RUN-TIME

visto che i prototipi, compresi i costruttori, sono accessibili e le loro proprietà pure, non dovrebbero essere modificabili. E se invece li modifico?

Le relazioni fra oggetti espresse dalle catene di prototipi sono mantenute e reificate a run-time, quindi prototype è una proprietà del costruttore che affibbia ad esempio ai point il quadratino The Point.

Come tali, sono dinamiche: eventuali modifiche sono immediatamente reificate nel sistema.

In particolare **è possibile, con diverse conseguenze:**

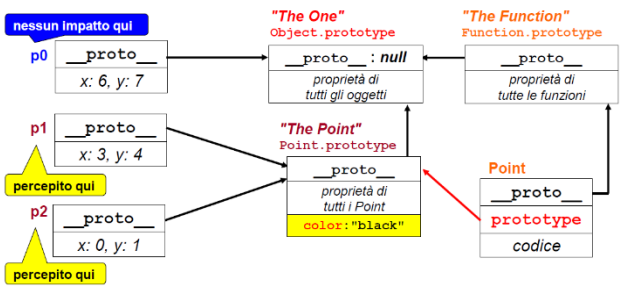
- **Aggiungere proprietà ad un prototipo in uso** → **TYPE AUGMENTING**
  - **Effetto: impatto immediato sugli oggetti già esistenti**. La struttura del sistema cambia a runtime. Può essere utile in alcuni casi, ad esempio se voglio mantenere le stesse proprietà per oggetti "vecchi"
  - **Si fa senza accedere a __proto__ direttamente** (perché non è pubblica e richiede l'esistenza di un tale oggetto), ma **sfruttando la proprietà prototype del corrispondente costruttore**
- **Togliere proprietà ad un prototipo in uso**
- **Sostituire un oggetto-prototipo con un altro (diverso)**:
  - **CREATING & INHERITING**: sostituire il prototipo di costruzione con un altro. **Non ha impatto sugli oggetti già esistenti, cambiano le regole da qui in poi.** Possiamo così creare una nostra ereditarietà prototipale reimpostando prototype del costruttore e agganciandolo ad un opportuno oggetto-prototipo.

Esempio - Type Augmenting

Consideriamo il solito esempio con i soliti tre punti (un " object literal" e due costruiti dal costruttore Point). Se si aggiungono proprietà a Point.prototype la modifica si riverbera immediatamente e dinamicamente sui due Point p1 e p2 già esistenti. Non impatta invece su p0, che fa riferimento a un prototipo diverso!

```
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 proprietà color!
showProps(p0);  // è rimasto com'era
```

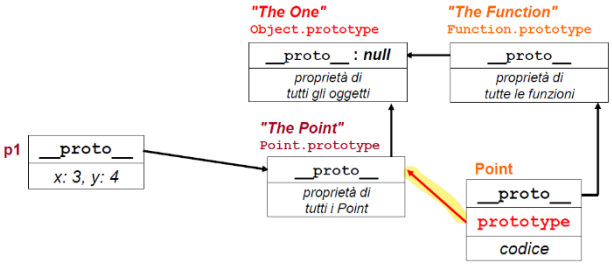



```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p0 = { x:6, y:7 };
p1 = new Point(3,4);
p2 = new Point(0,1);
Point.prototype.color = "black"
```

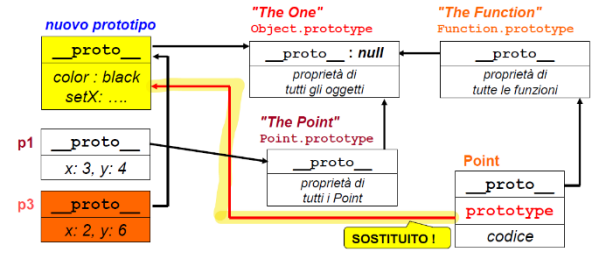
Esempio – Creating & Inheriting

Riprendendo il solito esempio, se ora sostituiamo il prototipo di costruzione di Point reimpostando la proprietà Point.prototype, tutti i futuri Point risulteranno diversi: i "Point futuri" non risulteranno "imparentati" coi "Point vecchi", facendo parte di diverse catene prototipali.

```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p1 = new Point(3,4); showProps(p1);
p2 = new Point(0,1); showProps(p2);
```




```
Point = function(i,j){ this.x=i; this.y=j; this.getX= … }
p1 = new Point(3,4);
Point.prototype = { color:"black",
                    setX:function(x){ this.x = x; } };
p3 = new Point(2,6); p3.setX(9)
```



18.3.5 – EREDITARIETÀ PROTOTYPE-BASED

L'approccio precedente è usato tipicamente per definire le relazioni di ereditarietà fra i propri tipi.

Per default, i costruttori usano come prototipo di costruzione un oggetto diverso per ogni costruttore ("The Point" per i Point, "The Pair" per i Pair, etc), ma ciò porta ad avere **oggetti senza parentele fra loro**, a parte l'antenato comune "The One".

Per avere oggetti in relazione padre/figlio ("is a") occorre impostare i prototipi di costruzione in modo da riflettere la tassonomia desiderata: l'ereditarietà fra classi dei linguaggi class-based diviene qui un'ereditarietà prototipale espressa da prototype.

Esempio

Obiettivo: esprimere l'idea che la **categoria Studente eredita dalla categoria Persona**. Occorre definire:

- Il costruttore Persona in accordo alle proprietà desiderate per le persone
- Il costruttore Studente in accordo alle proprietà desiderate per gli studenti

Se ci si limita a questo, però, Studente è una categoria stand-alone, non imparentata con Persona (se non nella nostra mente) → Per esprimere il fatto che Studente "is-a" Persona, gli oggetti-studente che saranno costruiti dovranno avere come prototipo una persona, non "The Student" (o un altro oggetto che non ha nulla a che fare con le persone).

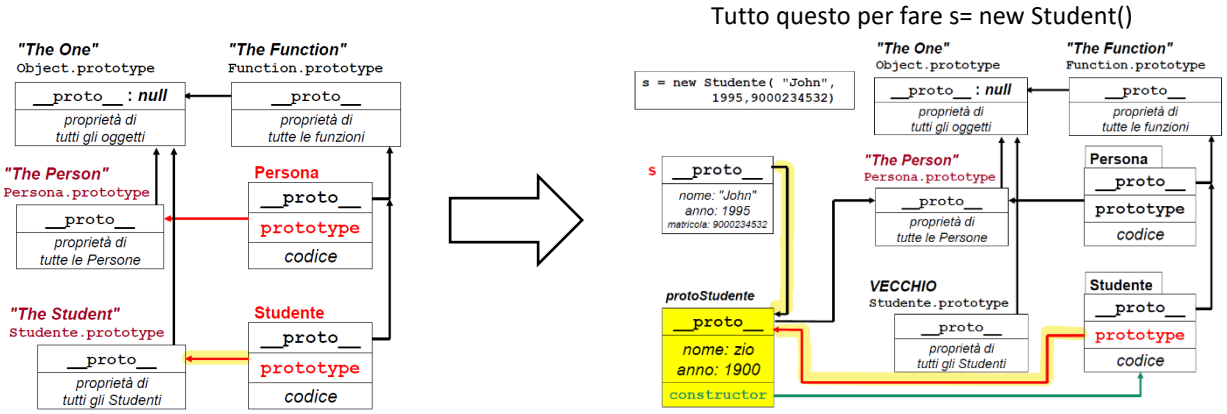
A tal fine, occorre:

1. **Scegliere un oggetto-persona che faccia da prototipo per tutti gli studenti** (che sia bello…)
2. **Impostare su esso Studente.prototype**, in modo che tutti gli studenti che verranno creati facciano riferimento ad esso
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 indispensabile, ma aiuta a mantenere la consistenza globale. Così facendo, infatti, chiedendo a un oggetto Studente chi sia il suo constructor, la risposta sarà Studente (come ci si aspetta) e non genericamente Persona (che sarebbe stato fuorviante)

```
Persona = function(nome, annoNascita){
  this.nome = nome; this.anno = annoNascita;
  this.toString = function(){
   return this.nome + " è nata nel " + this.anno }
}
```

```
Studente = function(nome, annoNascita, matricola){
  this.nome = nome; this.anno = annoNascita;
  this.matricola = matricola;
  this.toString = function(){
   return this.nome + " è nata nel " + this.anno +
    " e la sua matricola è " + this.matricola }
}
```

1) protoStudente = new Persona("zio", 1900);
2) Studente.prototype = protoStudente;
3) Studente.prototype.constructor = Studente



S = new Studente("John", 1995, 9002345) // stampa "John è nato nel 1995 e la sua matricola è 9002345"
protoStudente.hasOwnProperty("constructor") //true
s.hasOwnProperty("constructor") //false
s.constructor //function (nome, annonascita, matricola) { … }

RIFERIMENTI ALLA CLASSE BASE

Non avendo l'idea di classe, JavaScript non ha un modo diretto per potersi riferire alla "classe" base: niente keyword "super". Tuttavia, il **METODO CALL** permette di chiamare un **oggetto-funzione, svolgendo "circa" lo stesso compito → PATTERN:** funzione.call(target,…argomenti…)
Funziona a rovescio: è un inverso del controllo. Voglio che questo codice venga ad eseguire qui da me quando lo chiamo!

Esempio

Se ora infatti ripeto il codice di prima, sostituendo il contenuto della funzione associata a Studente, mi stampa comunque John del 1995.

```
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 è " + this.matricola }
}
```

18.3.6 – OGGETTO GLOBALE

JavaScript prevede un ambiente globale, che ospita funzioni e variabili definite fuori da ogni altro scope. È un opportuno oggetto, avente:

- **Come metodi, tutte le funzioni predefinite**
  - Inclusi i costruttori predefiniti (Object, Array, Boolean, Function, Number, String)
  - Nonché Date, RegExp, Math …
- **Più tutte le funzioni e le variabili globali definite dall'utente**

Non è prestabilito CHI o COSA sia: **dipende dall'ambiente ospite!**

- In un **browser**, tipicamente è l'oggetto **window**
- In un **server**, probabilmente è qualcun altro (l'oggetto **response**)

È importante saperlo solo se si usa **EVAL** o un'altra funzione che possa risentire dello specifico ambiente di valutazione. eval è un valutatore, quindi javascript può chiedere a se stesso di valutare un codice scritto in Javascript: dentro a quella eval, il suo ambiente globale è il mio? La tua risposta è la stessa che darei io?

FUNZIONE EVAL

La funzione eval valuta la stringa ricevuta interpretandola come programma Javascript:

- **Una stringa (testo) diventa codice e dà luogo a comportamento.** Attenzione: il codice ha accesso read/write access agli ambienti esterni ad esso! Al contrario, una chiamata a eval indiretta, via call, opera a livello globale
- **REFLECTION + INTERCESSION**

In questi casi, eval("var f") può essere diverso da var f:

- Nel primo caso, la variabile è definita in uno scope non globale
- Nel secondo caso, essa è invece in uno scope globale

Qui bisogna stare attenti al senso delle frasi. Se passo var f (che da sola non ha un gran senso… servirà per farci qualcosa dopo) vuol dire che viene introdotta una variabile globale o locale. Ma se nel mio programma c'è scritto eval("var f"), quell'eval viene eseguita nel mio ambiente o viene creato qualcosa di nuovo ed esterno? Ogni eval ha il suo mondo oppure usa quello in cui è stata chiamata?
A seconda di come viene interpretata e di dove viene messa, viene interpretata come globale o locale. Sapendo che nel client l'ambiente è "window", tutto quello che pensi sia globale, in realtà è dentro window. Allora bisogna chiedersi: la f che vedo qua è la f di window? Se la risposta è no, tu credi di essere in un mondo tutto tuo, invece non è così. → Se è importante saperlo, testare f === window.f

18.4 – DOVE I DUE LATI S'INCONTRANO

18.4.1 – COSTRUZIONE DINAMICA DI UNA FUNCTION

Come sappiamo, **ogni funzione JavaScript è un oggetto**, che però è **invocabile** e quindi **incapsula comportamento**.
Finora abbiamo utilizzato sempre dei ***function literal***, introdotti dalla parola chiave **function**: tale costrutto prevede una lista di argomenti fra parentesi tonde (…) e un corpo racchiuso in un blocco fra parentesi graffe { … }.
In realtà, è anche possibile **costruire dinamicamente oggetti-funzione**, tramite il costruttore **Function**:

- È un **costrutto linguistico di meta-livello**
- È un oggetto-funzione **destinato a creare altri oggetti-funzione**
- I suoi argomenti sono tutte stringhe
    - I primi N-1 sono i nomi dei parametri della funzione da creare
    - L'ultimo è il testo del corpo della funzione da creare

In pratica, se quello che vuoi fare è costruire una funzione che poi non modificherai, allora ti offro la parolina chiave function (con la f minuscola) che nasconde la new e che ha una sintassi semplificata. È solo uno shortcut per rendere facile quello che è l'uso prevalente, ma il meccanismo vero e generale è un altro e cioè è il signor Function (con la f maiuscola).

Esempio di Function literal

```
square = function(x){ return x*x }
```

Qui gli argomenti sono i parametri formali della funzione, il corpo della funzione è prestabilito e cablato nel blocco {…}. Se scopro che ad un certo punto voglio fare x*x+1, devo fermare il programma, cambiare la riga e farlo ripartire. Non si può agire mentre sta andando perché quello è codice cablato.
Tuttavia, in un mondo dinamico in cui gli oggetti li creo con delle new, posso creare oggetti quando ne ho bisogno e con i parametri che voglio io.

Esempio costruzione esplicita di una nuova funzione

square = new Function("x", "return x*x")

Qui sia gli argomenti sia il corpo della funzione sono stringhe e quindi non necessariamente sono costanti. Il corpo della funzione non è più prestabilito e cablato: è esso stesso un argomento di ingresso!
In questo modo ho un approccio più dinamico e cioè posso aggiungere funzioni a runtime e queste funzioni possono essere eseguite, non dobbiamo fermare tutto e farlo ripartire. È ovvio che i casi in cui questa funzionalità è usata non sono tantissimi…

18.4.2 – FUNZIONI COME DATI

Il costruttore Function permette di sintetizzare dinamicamente la funzione desiderata, creando "al volo" un pezzo di comportamento eseguibile → **Si rompe la tradizionale barriera codice/dati**: un "dato" (oggetto) può essere creato e immediatamente incorporato come parte del "codice" (comportamento eseguibile).
Ciò rende possibili **nuovi scenari ad alta dinamicità:**

- L'utente scrive da tastiera il testo della funzione desiderata
- Il costruttore Function sintetizza al volo una funzione che fa esattamente quanto richiesto dall'utente
- Un istante dopo, tale funzione può essere eseguita come se fosse sempre stata parte dell'ambiente

Certo che la sicurezza in questo modo viene meno perché se ho una totale apertura vuol dire che ho un mondo di casini possibili davanti a me perché se chiedo all'utente di creare la funzione leggendo da tastiera, potrebbe scrivermi delle boiate. Ad esempio potrebbe scrivere "delete * from..." e a quel punto è finita.

Esempio

```
var funz = prompt("Scrivere f(x): ");
var f = new Function("x", "return "+funz);
var x = prompt("Calcolare per x = ? ");
confirm("Risultato: " + f(x) );
```

Se l'utente scrive "x*x-1 4" → si esegue "return x*x-1" → si ottiene "15" … tutto ok.
Ma attenzione: "x+1 4" → return x+1 → 41 (argomenti = stringhe, quindi + significa "concatenazione")
Per risolvere, bisogna mettere un parseInt forzando il tipo della variabile x.

PROPRIETÀ COSTRUTTORI FUNCTION

Proprietà **statiche** (esistono anche mentre non esegue):

- **length** - numero di parametri formali (attesi)
  (in realtà sappiamo di poter chiamare la funzione anche con un numero diverso di parametri)

Proprietà **dinamiche** (mentre la funzione è in esecuzione):

- **arguments** - array contenente i parametri attuali
- **arguments.length** - numero dei parametri attuali
- **arguments.callee** - la funzione in esecuzione stessa
- **caller** - il chiamante (null se invocata da top level)
- **constructor** - riferimento all'oggetto costruttore
- **prototype** - riferimento all'oggetto prototipo

Metodi invocabili su un oggetto funzione ereditati dal prototype di object:

- **toString** - una stringa fissa (ma si può cambiare…)
- **valueOf** - la funzione stessa
- **call e apply** – supportano i pattern di chiamata indiretta

CALL E APPLY

Una funzione si invoca solitamente per nome, in modo diretto, tramite l'operatore di chiamata di funzione (). Nel caso di un metodo, ciò assume la nota forma **object.methodname(args)**.

Esistono però anche forme di chiamata indiretta, in cui:

- Il **target apparente** è il l'**oggetto-funzione** methodname
- L'**oggetto-target** della chiamata, object, è il **primo argomento**

methodname.call(object, argsList)

methodname.apply(object, argsArray)

Le due primitive call e apply si differenziano soltanto per il modo con cui passano gli argomenti args (una lista a,b,c… o un array [a,b,c,…]).

Esempio

Qui la funzione invocata è indipendente dall'ambiente di esecuzione, perché il corpo non contiene riferimenti a "this": **test = function(x, y, z){ return x + y + z }**

Questa non è una chiusura, quindi non abbiamo problemi di variabili globali o locali.

Di conseguenza, l'oggetto-target obj è totalmente irrilevante:

test(3,4,5) test.apply(obj, [3,4,5]) → array test.call(obj, 3, 4, 5) → list

Come sempre in Javascript, i parametri sono opzionali: se non ne esistono, apply e call assumono ovviamente la medesima forma.

Esempio

Questa funzione invece fa riferimento a "this": **prova = function(v){ return v + this.x }**

Quindi, l'oggetto destinatario del messaggio diventa rilevante perché determina l'environment di valutazione di x:

```
x = 88
prova.call(this,3)
```

Restituisce 3 + 88 = 91

```
x = 88
function Obj(u){
 this.x = u
}
obj = new Obj(-4)
prova.call(obj,3)
```

Restituisce 3 + -4 = -1

18.4.3 – ARRAY JAVASCRIPT

Un array Javascript è di fatto una lista "numerata":

- Come in Java, **gli elementi si numerano da 0**, length dà la lunghezza dell'array, la notazione prevede le parentesi quadre
- A differenza di Java, **length dà la lunghezza dinamica dell'array**
- **Cade, inoltre, il vincolo di omogeneità in tipo**: le celle contengono oggetti, cioè qualunque cosa
- È costruito sulla base del **costruttore Array o con […]**

Esempi

colori = new Array("rosso", "verde", "blu") varie = ["ciao", 13, Math.sin]

Si possono poi aggiungere nuovi elementi dinamicamente:

colori[3] = "giallo"

Mentre la typeof di una funzione dice function, una typeof di un array non dà array, ma dice "object", quindi per sapere se l'oggetto è un array, serve una circonvoluzione strana.

"Object.prototype" è "The One" e gli chiedo di eseguire la toString(), ma voglio che si esegua sull'entità colori, allora uso apply → Object.prototype.toString.apply(colori)

18.4.4 – OGGETTI COME ARRAY

Gli array costituiscono il supporto per gli oggetti: poiché ogni oggetto è caratterizzato da un insieme (variabile) di proprietà, un array è un modo pratico ed efficace per supportarlo. Di più, **gli oggetti sono array**:

- La **NOTAZIONE ARRAY-LIKE […] è usabile** per accedere per **nome** alle proprietà degli oggetti. obj[propname] permette di accedere a **proprietà di oggetti non prefissate**, **senza cablarne il nome nel codice**: propname può essere una variabile!
- La **NOTAZIONE PUNTATA OBJ.X** consente di accedere alla proprietà x dell'oggetto obj, ma **x è fisso e cablato nel codice**

Così come ( ) è la scorciatoia di invoke, scrivere il . è un modo per dire "accedi a quell'array nel punto x".
E se io volessi accedere ad una proprietà che ho aggiunto in corso d'opera e non mi ricordo come si chiama? Se sapessi il nome, potrei fare oggetto.proprietà, ma se non mi ricordo la proprietà e quindi non so se c'è oppure no? Allora, quando scrivo "object.x" vuol dire che sotto c'è un array a cui accedi con la stringa x.

NOTAZIONE ARRAY-LIKE: INTERCESSION & INTROSPEZIONE

La notazione array-like […] permette dunque di accedere per nome a proprietà di oggetti anche non prefissate e di selezionare per nome funzioni da un elenco.

- Se nome fisso: obj["x"] = obj.x obj["getX"]() = obj.getX()
- Se nome variabile: obj[nome] = ??

La possibilità di aggiungere/togliere proprietà a un oggetto pone il problema di scoprire **quali proprietà esso abbia in un dato istante** → **INTROSPEZIONE**
Il costrutto "for (variabile in oggetto) …" itera sui **nomi** (non sui valori) di tutte le proprietà, cioè gli indici perché per un array, i nomi sono gli indici 0, 1, 2, …

Esempio

```
function show(ogg){
        for (var pName in ogg) document.write("proprietà: " + pName +"<BR>")
}
```

Per accedere alle proprietà per nome (ed eventualmente modificarle) serve la notazione array-like → **INTERCESSIONE**.
Avere la notazione obj["x"] apre la porta ad una stringa non costante, quindi ad esempio ad una variabile. Qui siamo oltre la reflection: non vedo solo quello che c'è sotto, posso anche leggerlo e modificarlo, ecco perché parliamo di **intercession**.

Esempio

```
function show(ogg){
        for (var pName in ogg)
                document.write("proprietà: " + pName + ", tipo " + typeof(ogg[pName]) + "<BR>")
}
```

L'invocazione **show(p1);** sull'oggetto Point produce:
proprietà: x, tipo number proprietà: y, tipo number proprietà: z, tipo **number**
proprietà: getX, tipo function proprietà: getY, tipo function

L'invocazione **show(p2);** sull'oggetto Point produce:
proprietà: x, tipo number proprietà: y, tipo number
proprietà: getX, tipo function proprietà: getY, tipo function

18.5 – APPLICAZIONI MULTI-PARADIGMA

In passato abbiamo visto che da Java abbiamo usato tuProlog per risolvere le derivate passaggio per passaggio così come lo faresti a mano. Esistono molti interpreti fatti in modo da operare con Java.
Da alcuni anni un **interprete molto conosciuto** è **RHINO (Javascript for Java)**.
In Java 8 hanno aggiunto NASHORN che è più performante e integrato, ma Java ha deciso di dropparlo perché è troppo costoso. A quanto pare stanno lavorando su un altro motore che vuole integrare anche Javascript.

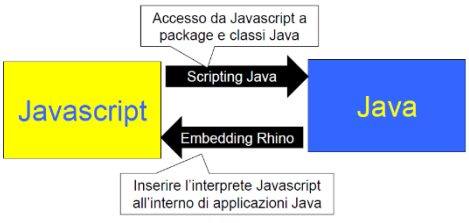
18.5.1 – JAVASCRIPT FOR JAVA

OBIETTIVO: **avere un interprete Javascript interoperabile con Java**, ma **usabile anche come** componente **stand-alone** e come **supporto a programmazione multi-linguaggio e multi-paradigma.**

- **1997-2014: RHINO**
  - Originariamente, costola di un browser Java-based di Netscape
  - Un tempo compilava il codice JavaScript in bytecode, poi modo interpretato
  - **Punto debole: prestazioni**
- **2014-2018: NASHORN (JAVA 8-11)**
  - Totalmente reingegnerizzato: il codice javascript gira nativamente sulla JVM
  - Prestazioni nettamente migliori ("competes with other standalone engines like Google V8, the engine that powers Google Chrome and Node.js")
- **2018: NASHORN deprecato in JAVA 11 (JEP 335)**
  - Marcato per possibile futura rimozione (non ancora avvenuta)
  - Motivo: "When it was released, it was a complete implementation of the ECMAScript-262 5.1 standard. With the rapid pace at which ECMAScript language constructs, along with APIs, are adapted and modified, we have found Nashorn challenging to maintain."
  - In sostanza: piccolo team di sviluppo, difficile star dietro ai cambiamenti… anche perché Oracle stessa è coinvolta in una alternativa: GraalVM
- **2018+: GRAALVM (www.graalvm.org)**
  - "**Universal VM for a polyglot world**": GraalVM support efficient execution of additional languages beyond JavaScript such as Ruby, R, and Python
  - **Standalone or not**: GraalVM can run either in the context of its own installation based on JDK 8 or via the standard JDK installation starting from JDK 11
  - **Supporto JavaScript più completo** (inclusi Node.js server framework) e veloce
  - Fin da luglio di due anni fa, è stato annunciato supporto per la migrazione da Nashorn

Fare reverse engineering della roba degli altri è illegale a meno che non sia per studiare o per fare un prodotto interoperabile. **Interoperabilità**: nel momento in cui vuoi provare ad unire due linguaggi, ci sono vantaggi nel farli toccare? Oppure non voglio che i due mondi vengano inquinati?
Se volessi passare ad esempio un punto con le variabili x e y da Javascript a Java, cosa gli passo? Una copia del mio punto oppure gli mando un riferimento a casa mia? Se in Javascript comincio ad aggiungere proprietà e poi passo il punto a Java, per lui cos'è quel robo che gli ho passato? È ancora un punto?

18.5.2 - RHINO



Multi-Paradigm Programming: Java ↔ Javascript

- È possibile usare classi e package Java da codice Javascript
- È possibile usare un motore Javascript da codice Java

java -jar js.jar [source.js] → avvia la shell di Rhino (ovvero, l'interprete Javascript).
L'interprete esegue istruzioni Javascript memorizzando oggetti, funzioni e variabili in un contesto che rimane disponibile fino alla chiusura dell'interprete (CTRL+C).
La shell di Rhino definisce:

- **Una variabile globale Packages** per accedere ai package java e com
- **Una variabile globale java** = Packages.java
- **Una funzione importPackages** analoga alla direttiva import di Java

È inoltre possibile usare classi Java scritte da noi.

Alcuni ***esempi*** a slide 118-121 pacchetto 19.

EMBEDDING IN JAVA

Per eseguire uno **script Rhino in Java,** occorrono:

- **Un contesto** per mantenere le informazioni thread-specific → **oggetto Context**
- **Uno scope** per mantenere le info script-specific (variabili globali, oggetti std, etc) → **oggetto Scope**
- **Lo script**, sotto forma di stringa o di Reader da cui leggerlo
- **Un dominio di sicurezza**, in cui eseguirlo (anche null)

L'esecuzione effettiva è poi lanciata dai metodi:

- Object **evaluateString**(scope, source, … , securityDomain)
- Object **evaluateReader**(scope, source, … , securityDomain)

Un Context mantiene le informazioni thread-specific sull'ambiente di esecuzione di uno script:

- Context ctx = Context.enter() → associa il thread corrente ad un Context e lo restituisce
- Context.exit() → rilascia il contesto

Uno Scope è un set di oggetti JavaScript che mantiene le informazioni script-specific sull'ambiente di esecuzione:

- Variabili globali dello script
- Oggetti standard (Function, Object...)

Uno Scope viene creato da un Context, ma è indipendente da esso:

- ScriptableObject scope = ctx.initStandardObjects();

18.5.3 – NASHORN

OBIETTIVI:

- **Riprogettare il motore JavaScript** sfruttando le **nuove istruzioni dinamiche aggiunte alla JVM** nel recente passato
- Migliore **aderenza allo standard ECMAScript 262** (v5, ora v6)
- **Nuova architettura**
- Aderenza alle **Scripting API** (JSR-223)

RISULTATI:

- **Nuovo codice, nuova architettura, prestazioni migliori** (a JVM "hot")
- **Ambiente interattivo per sperimentare** (interprete jjs)
- **Interoperabilità con Java più semplice** grazie a:
    - **Oggetti JavaScript predefiniti** che incapsulano package di uso comune
    - **La funzione Java.type** per accedere a qualunque tipo Java → scatena la reflection
    - **Sintassi shortcut** per manipolare liste e mappe Java
    - **Funzioni JavaScript usabili come lambda expression** (convertite in functional interfaces Java8)

ACCEDERE A JAVA

1. Accedere ai package e tipi Java:
   - Utilizzare gli oggetti Javascript predefiniti per accedere a package
   - Sfruttare la funzione Java.type per ottenere un riferimento al tipo
2. Creare oggetti e lavorarci:
   - Per creare oggetti, usare l'operatore new di Javascript
   - Per accedere a metodi statici, invocarli direttamente
   - Per accedere a proprietà di oggetti Java, usare la notazione Javascript

Esempio

HP: il bytecode della classe Java Thermo.class è nella directory corrente. Ovviamente devo impostare il classpath se no non vede le classi

Jss -classpath . var Thermo = Java.type('Thermo') var th1 = new Thermo()
th1 → Thermo operating at [20.0-21.0]

- L'operatore Java.type fornisce un riferimento JavaScript alla classe Java
- L'operatore new di JavaScript crea un oggetto JavaScript mappato sul corrispondente oggetto Java "retrostante"
- La richiesta di valutare in JavaScript th1 viene tradotta nella chiamata al metodo toString della classe Java corrispondente, che genera la stringa mostrata.

INTEGRAZIONE

- **Le stringhe Nashorn sono stringhe JavaScript (non Java)**, ma 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), quindi, reali (non interi)**. La parte frazionaria è rimossa automaticamente se il valore è passato a un metodo Java che si aspetta un intero
- Sotto, **Nashorn cerca comunque di mantenere la computazione fra interi quando possibile**, per motivi di efficienza (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 da/verso array Java tramite Java.from e Java.to sono molto facili
- Nashorn fornisce una **sintassi agevolata per liste e mappe Java:**
  - L'operatore JavaScript […] può operare anche su liste e mappe Java, sia in lettura (mappata sul metodo get) che in scrittura (mappata sul metodo set)
  - Si può anche iterare su esse: nel caso delle mappem si può iterare sia sulle chiavi (tramite il ciclo for) che sui valori (tramite il nuovo ciclo for each)
- Nashorn fornisce una **sintassi agevolata per manipolare facilmente le proprietà di oggetti Java:**
  - Si può usare la notazione JavaScript per le proprietà (obj.property) anche su oggetti Java (vengono effettuate le opportune chiamate ai metodi setter/getter)
  - È ammessa anche 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 possibile lanciare direttamente uno script che contenga istruzioni JavaFX con -fx
  - La variabile d'ambiente $STAGE fornisce un riferimento allo Stage

- Nashorn **permette di estendere in JavaScript una classe Java e/o implementare in JavaScript un'interfaccia Java (gestione dell'ereditarietà)**
    - Funzione speciale Java.extend + qualunque numero di classi e interfacce
    - Il tipo restituito è un'istanza del tipo passato come argomento, quindi, i metodi e le proprietà invocabili sono solo quelle da lui definite
    - Eventuali proprietà aggiuntive sono usabili solo internamente

Se l'interfaccia o classe astratta da estendere ha un solo metodo (S.A.M.: Single Abstract Method), si può specificarne l'implementazione inline, senza neanche dover precisare il nome del metodo: di fatto, è una lambda expression!
Se un metodo Java si aspetta una interfaccia funzionale (ossia un oggetto lambda), si può passargli una implementazione Javascript inline.

DA JAVA A NASHORN

Nashorn può essere invocato da Java tramite il meccanismo generale per gli scripting engine, JSR-223

- **Meccanismo language-independent, introdotto in Java 6**
    - Creare un opportuno ScriptEngine (un'interfaccia di javax.script)
    - Invocare su di esso il metodo eval con opportuni argomenti → risultato, se c'è, è un Object
- **Lo script da eseguire può essere fornito a eval in vari modi:**
    - Direttamente come stringa
    - Sotto forma di Reader da cui leggere
- **Oggetti Java possono essere passati a Javascript con il metodo put**
    - Mapping fra nomi di metodi e proprietà JavaScript/Java
    - Importante che le relative classi siano pubbliche, se no chiamate a metodi e proprietà risulteranno undefined

Alcuni ***esempi*** a slide 139-145 pacchetto 19.

18.5.4 - GraalVM

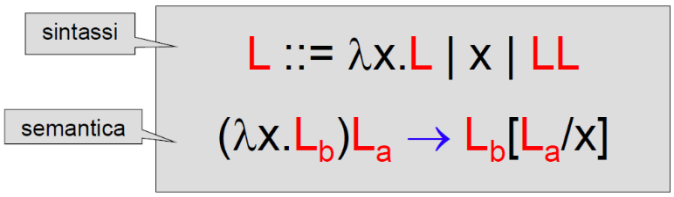
Progetto Oracle orientato a Polyglot programming: fornisce un'apposita **Polyglot API** ma aderisce comunque anch'esso alle Scripting API (JSR-223).
Interoperabilità con Java:

- Supporta l'interoperabilità con Java in modo molto simile a Nashorn (spesso con identica sintassi) → migration guide su GitHub
- Non permette accesso diretto alle classi Java (per sicurezza), obbligando a passare da toJava.type()
- Previene conversioni di tipo con perdita di informazione (i relativi metodi lanciano eccezione)

Esempi slide 148-149 pacchetto 19.

19 – INTRODUZIONE AL LAMBDA CALCOLO

Contemporaneamente alla MdT (A. Church, ~1930), nacque l'idea di sviluppare il **minimo formalismo capace di descrivere qualunque algoritmo (Turing-equivalente)** partendo da **un solo concetto, la funzione**, e usando **un solo paradigma, l'applicazione di funzioni.**
Era una sfida: mentre Turing voleva una macchina che risolvesse qualsiasi cosa computabile, un matematico voleva una funzione che si trasformasse in qualcos'altro. Si è scoperto che il set di algoritmi risolvibili è sempre quello, solo espresso in modo diverso.
Quando si hanno linguaggi piccoli, detti **LINGUAGGI FONDAZIONALI**, su quelli si possono dimostrare teoremi e proprietà, poi vengono verniciati in versioni disponibili al grande pubblico.
Nacque quindi la **base fondazionale per tutti i linguaggi funzionali e object-oriented del XX secolo.**



Per esprimere il lambda calcolo, ci bastano una sola regola sintattica e una sola regola semantica.
L ha tre forme: x sono i simboli terminali (o costanti), λ ha un significato particolare, così come il punto. Cosa si può scrivere? λnome.qualcosa oppure un identificatore oppure puoi scriverlo tante volte.
Se scriviamo λx.x+1, poi diventa: function(x){return x+1}
Quindi λ = function → x = (x) → . = {...} → x+1 = return x+1

Quello che fa la regola semantica è semplicemente search & replace. Al posto del parametro formale ci va il parametro attuale (e questa sostituzione si fa ogni volta che lo leggi). Si ottiene solo Lb dove, ad ogni occorrenza di x, è stato sostituito La.

L può esprimere qualsiasi struttura dati e qualsiasi algoritmo. Rappresenta una **trasformazione atomica:**

- Idea semplice: **sostituzione di testo**
- Il "risultato" è il termine Lb con, al suo interno, tutte le occorrenze di x sostituite da La

L'idea è: ti consento di scrivere funzioni che possono essere applicate ad argomenti, il tutto facendo semplicemente search & replace.

19.1 – SINTASSI

Un **LAMBDA-TERMINE** può essere:

L,M,N ::= λx.L | x | LL

- x, una **variabile** (λ e . sono simboli terminali)
  - Simboli di variabili: x, y, z, w
- λx.L, una **funzione**
  - Senza nome (chiamata chiusura)
  - Con un parametro x (una variabile)
  - E un **corpo L**, che è a sua volta un lambda-termine
- LM, **l'applicazione di una funzione**
  - **L è una funzione**
  - **M è il parametro** attuale che viene **applicato alla funzione** (prende il posto della variabile sostituita)

19.1.1 – JAVASCRIPT

In JavaScript le funzioni sono first-class objects, ci sono le chiusure ed esistono le funzioni anonime (come nella notazione lambda).

JAVASCRIPT ↔ LAMBDA

Definizione della funzione:	f = function(x){return expr;}	λx.<expr>
In JavaScript > 1.7, anche:	f= function(x) expr;	
Applicazione della funzione a un argomento:	f(arg)	(λx.expr)(arg)

Nelle lambda non è previsto che io possa assegnarle ad un nome, ma in Javascript sì ed è già qualcosa di più carino. Quando devo applicare la funzione ad un argomento, il mapping è ancora più diretto ed evidente.

19.1.2 – C# E JAVA8

In C# & Java 8 la forma λ prefissa è sostituita da operatori infissi

- => in C#
- -> in Java 8

I tipi degli argomenti possono essere espliciti o inferiti

- Delegati in C#
- Interfacce funzionali in Java 8

C# & JAVA 8 ↔ LAMBDA

Definizione funzione anonima:	(x,y) => expr;	λ x.<expr>
	(int x) => expr;	λ x.<expr>
Chiamata:	f(arg)	(λ x.expr)(arg)

dove f è una variabile del "giusto" tipo:

delegate void f(int n); f = lambdaexpression; ProperFunctionalInterface f = lambdaexpression

Questo si può fare perché in C# si usano i delegati. Non aggiungiamo nulla però, è solo per comodità, per non scrivere tutto ogni volta.

Esempio

Lambda Expressions: (x,y)=> x+y; z => z*z;

Il termine più a destra è una espressione (denota un **valore**).

Lambda Statements:	(x,y)=> x==y;	(x,y)=> return x+y;	() => Console.WriteLine("Lambdizzatevi!");
	() => return "Ciao Lambda!";		(int x, int y) => return x>y;
	z => return z*z;		k => k=1; Console.WriteLine(k);

Tipi **espliciti**: devono corrispondere al delegato usato per chiamarla, come si vede nei cast ad int.
Tipi **impliciti**: il tipo è inferito dal delegato usato per chiamarla, come si vede nelle ultime due lambda. Siccome si diventerebbe scemi a mettere i tipi ovunque, i compilatori fanno **Type Inference**, ma il concetto è lo stesso. Nell'ultima lambda, il termine più a destra è una istruzione (**NON** denota un **valore**).

19.1.3 – LAMBDA CALCOLO vs OOP (Object-Oriented Programming)

DEFINIZIONE DI METODI:

- **λx.x si può leggere come Object m1(Object x){ return x;}**

- Le funzioni in lambda sono senza nome
- I nomi delle variabili sono irrilevanti
- λx.x (o anche λz.z, λq.q, etc) è la ***funzione identità***

- **λx.L si può leggere come Object m2(Object x){return expr;}**
  - Dove L è il lambda-termine corrispondente a expr

INVOCAZIONE DI METODI → **(λx.x)y si può leggere come m1(y)**
Dove m1 è il nome (obbligatorio in C# e Java) del metodo che contiene il codice corrispondente a λx.x
In pratica ho chiamato la funzione identità, le ho dato y e mi ha restituito y sostituendola ad x.

19.1.4 – AMBIGUITÀ GRAMMATICALE

Abbiamo un problema: **LA GRAMMATICA LAMBDA È AMBIGUA!** Come si deve interpretare λx.xy ?

- Come (λx.x)y → applicare y alla funzione (λx.x), funzione identità
- Come λx.(xy) → una funzione di parametro x e corpo xy, a sua volta interpretabile come applicazione di y alla funzione x

Per disambiguare, parentesi e/o associatività:

- **L'applicazione di funzioni è associativa a sinistra** → yLxM si legge come ((yL)x)M
- **I corpi delle funzioni si estendono il più possibile** → λx. λy.xyx si legge come λx.(λy.xyx) ovvero come λx.(λy.(xy)x)

Se usassi il search & replace, potrei fare in un giro solo la sostituzione di due variabili x e y? No, quindi si fanno due passaggi utilizzando una chiusura.

19.2 - SEMANTICA

La semantica è definita tramite le regole di trasformazione nel riquadro. $(\lambda x.L)M \rightarrow L[M/x]$

Passi computazionali:

- **Identificazione di un pattern del tipo (λx.L)M** e applicazione della funzione λx.L al parametro M
- **Sua trasformazione nel lambda-termine L[M/x]** sostituendo, nel corpo della funzione L, ogni occorrenza di x con M. Questa trasformazione si chiama **RIDUZIONE** (anche se le allunga…).

Nella fase di identificazione, si cerca una funzione con qualcosa di fianco: vuol dire che puoi sostituire uno dentro l'altro. Poi, nella trasformazione, prendi il corpo della funzione L e ogni volta che trovi x ci metti M.

Esempi

- x è chiaramente non riducibile
- λx.x è pure non riducibile
- (λx.x)y si riduce a y in quanto si applica il parametro attuale y alla funzione identità (λx.x).Questo passaggio si scrive in breve come (λx.x)y → y.
  Infatti, con riferimento allo schema generale (λx.L)M → L[M/x], qui L è x, M è y, e dunque L[M/x] si legge come x[y/x], ovvero y
- (λx.xx)(λy.(λz.yz)) si riduce a (λy.(λz.yz))(λy.(λz.yz)) in quanto qui L è xx, ergo (λx.xx) duplica il suo parametro formale x, che qui è il lambda termine (λy.(λz.yz)).
  Quando ho una lambda ho una funzione, non ho una sequenza di funzioni, non ho i cicli, quindi ho bisogno di usare questi "trucchetti" per fare in modo che il mio calcolo si ripeta. In questo caso 2 volte, ma se ne mettessi di più, avrei tanti copia e incolla in base a quanti passi voglio io

19.2.1 – SIGNIFICATO INTESO

Cosa significa "applicare" una funzione a un argomento? In $(\lambda x.L)M \rightarrow L[M/x]$ lambda, significa "valutarne" il corpo dopo aver sostituito al parametro formale il parametro attuale.

Ad esempio, $(\lambda x.x)y$ → y si può leggere come "dato un metodo Object m(Object x){ return x;}, invocandolo con parametro attuale y: m(y) si ottiene come risultato y.
Dualmente, una chiamata C o Java come f(4), dove f sia: int f(int x){ return x*2*x;} si può rappresentare in lambda calcolo come $(\lambda x.x*2*x)[4/x]$.

19.2.2 – FUNZIONE A CON PIÙ ARGOMENTI

Una funzione a più argomenti può essere rappresentata in Lambda come funzione di 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, $(\lambda x.\ \lambda y.xy)LM$ → $(\lambda y.xy)[L/x]M = (\lambda y.Ly)M$ perché prendo il corpo xy e scrivo L al posto di x
- A sua volta, $(\lambda y.Ly)M$ è una funzione in y, in cui y vale M pertanto $(\lambda y.Ly)M$ → $(Ly)[y/M] = LM$.
  Siccome ho un altro argomento, devo prendere il corpo Ly e schiaffarci M
- In definitiva: $(\lambda x.\ \lambda y.xy)LM$ → $xy[L/x][M/y]$
  È un modo contorto di fare le cose: ti dà un ordine perché non puoi fare contemporaneamente più sostituzioni.

Analogamente, con tre argomenti:

- $\lambda x.\ \lambda y.\ \lambda z.L$ è una funzione nei tre argomenti x,y,z
- $(\lambda x.\ \lambda y.\ \lambda z.L)MNO$ applica la funzione ai tre argomenti attuali M,N,O ottenendo L[M/x,N/y,O/z]

19.3 – FUNZIONI NOTEVOLI

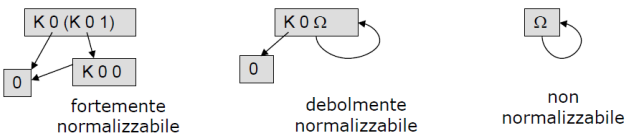
Alcune funzioni notevoli hanno un nome standard:

- I ::= $\lambda x.x$ → funzione **identità**, dato x restituisce x
- K ::= $\lambda x.\ \lambda y.x$ → prende il primo e butta via il secondo: **selezione e proiezione**
- S ::= $\lambda n.\ \lambda z.\ \lambda s.s(nzs)$ → utile per somma i numeri naturali: se c'è 1, rappresenta i numeri successivi
- ω ::= $\lambda x.xx$ → questo è il **copia e incolla**
- Ω ::= $\omega\ \omega = (\lambda x.xx)(\ \lambda x.xx)$ → se devi essere turing equivalente devi fare in modo di inlooparti.
  Questo è il copiatore che fa copia e incolla di sé stesso: **si riproduce da solo** all'infinito e l'editor non avrà mai alla fine perché non esiste una fine

A queste funzioni notevoli **si aggiungono alcuni combinatori (Y, Z)** che vedremo in seguito.

19.4 – FORME NORMALIZZATE

Un lambda termine è in forma normale se non si possono applicare altre riduzioni e ciò può dipendere dall'ordine con cui si riduce.
Si dice che **un Lambda termine è:**



- FORTEMENTE NORMALIZZABILE se **qualunque sequenza** di riduzioni porta a una forma normale

- **DEBOLMENTE NORMALIZZABILE** se **esiste almeno una sequenza** di riduzioni che porta a una forma normale
- **NON NORMALIZZABILE** se **non si arriva mai a una forma normale**

19.4.1 – TEOREMA DI CHURCH-ROSSER

Il Teorema di Church-Rosser dice che "**OGNI LAMBDA HA AL PIÙ UNA FORMA NORMALE**".
Di conseguenza, essa **può** quindi **essere interpretata come risultato** e il **lambda-calcolo è deterministico**: i grafi aciclici hanno una e una sola foglia.

- **Se** il termine è **fortemente normalizzabile**, siamo sicuri di poter raggiungere la forma normale
- **Se** il termine è **solo debolmente normalizzabile**, scegliere il "giusto" ordine di valutazione per non incappare nel "ramo infinito" del grafo è cruciale → **importanza della strategia di valutazione**

Quindi:

- **Lambda termini fortemente normalizzabili** rappresentano computazioni che **terminano sempre**
  Esempio: la computazione f(2,4) con int f(int x,int y){ return x+y+1; }
- **Lambda termini debolmente normalizzabili** rappresentano computazioni che **terminano solo se si sceglie il "giusto" ordine di valutazione**
  Esempio: la computazione f(0) con int f(int x){ return x==0 ? 0 : f(x); }
- **Lambda termini non normalizzabili** rappresentano computazioni che **non terminano mai**
  Esempio: la computazione f(2) con int f(int x){ return f(x); }

19.5 – COMPUTARE CON LE LAMBDA

Gli algoritmi descrivono funzioni, quindi prendono in input una struttura dati e 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**
- **Si esprime la computazione applicando L a D** (ossia, scrivendo LD) **e riducendo LD "il più possibile"**
- Il **risultato** è il nuovo lambda-termine **R**

Da notare che **i dati sono essi stessi dei termini-funzione** "opportunamente scelti" (ossia, non riducibili).

Esempio - Booleani
Boolean come tipo di dato astratto → Due valori (true, false) + Operazioni (algoritmi): OR, AND, NOT, XOR.
Per caratterizzarli serve:

- Due valori "opportuni" per rappresentare true, false
  - True → **λx.λy.x (T)** funzione che restituisce il 1° di due argomenti
  - False → **λx.λy.y (F)** funzione che restituisce il 2° di due argomenti

- Garantire che NOT(true)=false e che NOT(false)=true
  - Not → **λx.xFT** funzione che
    - Se x è T, restituisce il 1° di FT, ossia F
    - Se x è F, restituisce il 2° di FT, ossia T
- Garantire che AND(true,true)=true e che AND(true,false)=false
  - And → **λx. λy.xyF** funzione che
    - Se x è T, restituisce il 1° di yF, ossia y
    - Se x è F, restituisce il 2° di yF, ossia F
- Garantire che OR(x,y) = NOT(AND(NOT(x),NOT(y)))

T: λx.λy.x
F: λx.λy.y
Not: λx.xFT
And: λx.λy.xyF

Da qui, gli assiomi sono:

- (λy.Ly)M
- Not **T** = (λx.xFT)**T** → **TFT** → (λx. λy.x)**FT** → (λy.**F**)**T** → **F**
- Not **F** = (λx.xFT)**F** → **FFT** → (λx. λy.y)**FT** → (λy.y)**T** → **T**
- And **T** X = (λx. λy.xy**F**)**T**X → (λy.**T**y**F**)X → **T**X**F** → X
- And **F** X = (λx. λy.xy**F**)**F**X → (λy.**F**y**F**)X → **F**X**F** → **F**

Mappando le lambda in funzioni otteniamo che:

TRUE	**λ**x.**λ**y.x	Object T(Object x,Object y){ return x;}
FALSE	**λ**x.**λ**y.y	Object F(Object x,Object y){ return y;}
NOT	**λ**x.xFT	boolean Not(boolean x){ return x ? false : true; }
AND	**λ**x. **λ**y.xyF	boolean And(boolean x, boolean y){ return x ? y : false; }

Anche cose semplici come i boolean sono rappresentate da sintassi "strane" come "λx. λy.x" e "λx. λy.y". Può lasciare perplessi, ma è solo questione di sintassi. In realtà, è quello che facciamo sempre, in qualunque linguaggio, quando scegliamo un modello dei dati.
La "stranezza" del lambda calcolo è che anche le costanti sono "termini funzione" (non riducibili): è il caso di true e false nell'esempio precedente.

19.5.1 – STRATEGIE DI RIDUZIONE

Abbiamo detto che **nel lambda-calcolo:**

- Un primo lambda-termine **L descrive l'algoritmo**
- Un altro lambda-termine **D descrive i dati di input**
- **Si esprime la computazione applicando L a D** (ossia, scrivendo LD) **e riducendo LD "il più possibile"**
- Il **risultato** è il nuovo lambda-termine **R**

L'idea di ridurre "il più possibile" introduce il concetto di strategia di riduzione: d'accordo ridurre "il più possibile", ma come? **La strategia può fare la differenza**: con certe strategie le sostituzioni potrebbero talora non terminare!

Adottare una strategia o l'altra può fare la differenza:

- Sostituzioni infinite → non avere un risultato
- Sostituzioni finite → arrivare a un risultato

I due PRINCIPI ISPIRATORI ALTERNATIVI sono:

- **EAGER EVALUATION**: privilegiare la valutazione dell'argomento, rispetto all'applicazione dell'argomento alla funzione → **valuta l'argomento prima possibile** (strict evaluation)
- **LAZY EVALUATION**: privilegiare l'applicazione dell'argomento alla funzione, rispetto alla valutazione dell'argomento → **valuta l'argomento più tardi possibile**, "solo quando serve" (non-strict evaluation)

Da questi princìpi sono state sviluppate varie strategie, usate poi nei diversi linguaggi di programmazione.

STRATEGIE EAGER (strict)

Più usate ma si potrebbero inloopare.

- **APPLICATIVE ORDER (rightmost innermost)**
  - Prima valuta/riduce gli argomenti, poi li applica alla funzione

- Tenta anche di ridurre il più possibile i termini dentro la funzione prima di applicare gli argomenti, cioè prima di chiamarla
- Può non trovare la forma normale se termini debolmente normalizzabili
  → **non è una strategia normalizzante**
- Si parte dalla cosa più a destra e più interna

- **CALL BY VALUE** (famiglia di strategie)
  - Simile alla precedente
  - Non riduce i termini dentro la funzione prima di chiamarla
  - Considera come corpo solo lambda termini della forma E ::= x | xE | λx.L escludendo ad esempio l'identità
  - **La più usata nei linguaggi classici come C e Java**
  - Se il valore è un puntatore, stesso effetto di call by reference
- **CALL BY REFERENCE**
  - Variante della prima: riceve un riferimento implicito all'argomento del chiamante
  - Non è identica al passaggio di un puntatore in call by value, lì il riferimento è esplicito
  - Disponibile in vari linguaggi (C++, Pascal) ma quasi mai come default
- **CALL BY COPY-RESTORE**
  - Nota anche come "call by value return/result"
  - Caso speciale della precedente adatto al multi-threading

STRATEGIE LAZY (non-strict)

Usate di rado, ma se c'è una forma normale, la trovano di sicuro

- **NORMAL ORDER (leftmost outermost)**
  - Riduce prima la lambda più a sx e considera corpo = blocco che compare alla sua dx
  - È "normale" perché se esiste una forma normale, la trova
  - Può causare la creazione di copie multiple dello stesso termine (inefficiente)
  - Eliminata dalla call by need
- **CALL BY NAME**
  - Come sopra, ma considera irriducibili le lambda abstraction (λx.x)
  - Non porta le sostituzioni fin dentro le funzioni interne
  - In .NET è simulabile con delegati o parametri Expression<T> passando un AST
- **CALL BY NEED** → la più usata, di solito sinonimo di Lazy Evaluation
  - Come sopra, ma evita rivalutazioni multiple dello stesso termine, memorizza risultato
- **CALL BY MACRO**
  - Simile alla call by name, si basa su sostituzioni testuali
  - Richiede attenzione ai nomi, per evitare effetti spuri non voluti

Esempio - Forma fortemente normalizzabile

Normal Order (Lazy) **VS** **Applicative Order (Eager)**

(λx.λy.x) p ((λu.v)u) → (λy.x)[p/x] ((λu.v)u) → (λy.p) ((λu.v)u) → p [y/((λu.v)u)] → **p**

(λx.λy.x) p ((λu.v)u) → (λx.λy.x) p v[u/u] → (λx.λy.x) p v → (λy.x)[p/x] v → (λy.p) v → p[v/y] → **p**

Esempio – forma debolmente normalizzabile

Normal Order (Lazy) **VS** **Applicative Order (Eager)**

(λx.a) Ω → a [Ω/x] → a

(λx.a) ((λx.xx) (λy.yy)) → (λx.a) (xx)[x/(λy.yy)]
→ (λx.a) ((λy.yy)(λy.yy)) = **(λx.a) Ω** ***si riproduce!***

19.6 – TURING EQUIVALENZA

Il Lambda calcolo è Turing-equivalente. Per esserlo occorre infatti saper rappresentare:

- I numeri naturali: 0,1,2,3…
- Il "successore" succ(n)=n+1
- La "proiezione" f(x,y)=y
- La composizione f(g(x))
- La ricorsione

Il **problema** è che nel lambda calcolo l**e funzioni sono anonime!**

19.6.1 – IL PROBLEMA DELLA RICORSIONE

PUNTO FISSO:
in matematica, x è un *punto fisso* per una funzione f se x=f(x), ossia la funzione mappa il valore x in se stesso.

Per esprimere la ricorsione occorrono due ingredienti:

- La capacità di definire funzioni 😊
- La capacità di richiamare funzioni per nome ☹

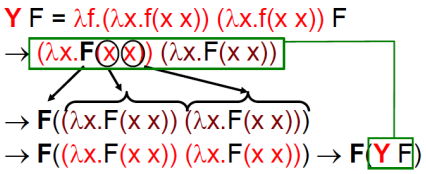
Occorre quindi un modo alternativo per poter eseguire il corpo della funzione, ma al contempo "rigenerare in qualche modo" la funzione stessa → A questo provvede il **combinatore di punto fisso Y**.

IL COMBINATORE DI PUNTO FISSO Y è definito come: $Y = \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$

Applicandolo a una funzione F l'effetto è YF → F(YF), ossia si ottengono esattamente i due effetti richiesti:

- Eseguire il corpo della funzione, ma al contempo F(..)
- "rigenerare in qualche modo" la funzione stessa YF

Dove **l'entità YF è un punto fisso per la funzione F.**



Attenzione perché il combinatore Y presuppone la strategia call by name: se usato con strategia call by value DIVERGE, perché il lambda termine (x x) presuppone di applicare la funzione x a se stessa (per capirlo meglio, conviene tradurne il codice in Javascript).
Si può risolvere la cosa simulando la call by name nel solito modo.

Esempio – Traduzione in Javascript

Per usare questo approccio in un modello computazionale basato su call-by-value occorre simulare la call by name nel solito modo, introducendo una funzione intermedia ausiliaria.
Il combinatore Y modificato per funzionare con la call-by-value viene anche chiamato **COMBINATORE Z**.

```
function Y(f) {
 return (
          (function(x) {return f( x(x) ); })
          (function(x) {return f( x(x) ); })
    );
}
```

DIVERGE *con call by value perché per chiamare la funzione x bisognerebbe prima valutare x*

Combinatore Y originale: (richiede call by name)
$Y = \lambda f.(\lambda x.f(\mathbf{x\ x}))\ (\lambda x.f(\mathbf{x\ x}))$

⟹

Combinatore Z modificato: *(funziona con call by value)*
$\mathbf{Z} = \lambda f.(\lambda x.f(\lambda v.((x\ x)\ v)))\ (\lambda x.f\ (\lambda v.((x\ x)\ v)))$

In questa forma, può essere implementato in qualunque linguaggio che abbia funzioni come oggetti di prima classe (Javascript, C#, Scala, Java8…).

Esempio – Traduzione in Javascript

```
function Z(f) {
 return (
   (function (x) {
     return f( function(v){ return x(x)(v); } ); })
   (function (x) {
     return f( function(v){ return x(x)(v); } ); })
    );
}
```

Per avere algoritmi ricorsivi con il combinatore di punto fisso, bisogna allora applicare questo approccio:

- **Occorre eliminare la ricorsione esplicita**, sostituendola con una implicita
  - **Trovare una F di ordine superiore** il cui punto fisso sia la funzione ricorsiva f desiderata

- **Estrarre dalla funzione ricorsiva f il passo-chiave**, incapsulandolo in una nuova funzione step **non ricorsiva**
- Delegare al combinatore Y la "composizione" dei passi di step, tramite la scrittura **Y step**
- La "**chiamata**" di funzione sarà quindi **(Y step)(args)**

Esempio - Fattoriale

Partendo dalla classica definizione ricorsive, introduciamo perciò una funzione di ordine superiore F, che riceva f come argomento. Sarà poi il combinatore Y a far funzionare il tutto.

La classica definizione ricorsiva:
fact(*x*) = (x==0) ? 1 : x * **fact**(x-1)

F := λ**f**.λx.((x==0) ? 1 : x * **f**(x-1))

Con questa funzione di ordine superiore, l**a computazione effettiva del fattoriale si esprime scrivendo (YF)(args) ossia in questo caso (YF)(n).**

3! = (YF)(3) = F(YF)(3) =
λ**f**.λx.((x==0) ? 1 : x * **f**(x-1))(**YF**)(3) =
λx.((x==0) ? 1 : x * **(YF)(x–1)**)(3) =
(3==0) ? 1 : 3 * (YF)(3-1) =
3 * (YF)(2) =
3 * 2! =
e così si prosegue, fino a..
6 * (YF)(1) = 6 * (YF)(0) = 6 * 1 = 6

Dove

Z = λf.(λx.f(λv.((x x) v))) (λx.f (λv.((x x) v)))

Javascript:

```
FactGen  =  function(f){
              return function(x){
                return (x==0) ? 1: x*f(x-1); }
            }

var fact = Z(FactGen)
```

Usiamo il combinatore **Z** anziché **Y** perché quest'ultimo divergerebbe con la strategia call-by-value adottata da Javascript

ESEMPIO

3! = `fact(3) = Z(FactGen)(3) = …`

```
function Z(f) {
 return (
   (function (x) {
     return f( function(v){ return x(x)(v); } ); })
   (function (x) {
     return f( function(v){ return x(x)(v); } ); })
    );
}
```

Z viene usato perché se usiamo il combinatore originale Y con la strategia call by value, il risultato diverge.

Esempio Java slide 51-54, C# slide 55, Scala slide 56 - pacchetto 20

Il **Lambda calcolo è stato usato** per **definire e formalizzare proprietà dei linguaggi di programmazione**, come **sorgente d'ispirazione per i linguaggi funzionali** (ML, Scheme, Lisp, Haskell, Javascript, Scala, C#, Java8..) e come **base per studiare i concetti dei sistemi di typing**.
Naturalmente però non è destinato agli utenti finali, ma ne traggono vantaggio inconsapevolmente.

20 – SCALA

Scala è un linguaggio:

- **Scalabile**: da piccoli script (interpretati) a grandi sistemi
- **Java-based**: compila in bytecode e gira su JVM
- **Pienamente integrato/integrabile con Java** a livello di librerie
- È un **linguaggio blended**
  - **Object-oriented** con pieno supporto allo stile imperativo
  - **Funzionale** con **funzioni come first-class entities**, chiusure, netta distinzione fra valori e variabili, tail-recursion optimization
  - **Stile di codifica conciso** (in media: -50% Java) che renda semplice l'uso di strutture e idiomi complessi

"A blend of OO and functional programming in a statically typed language"

Le principali caratteristiche di Scala sono:

- **Scalabilità**: programmi di diversa dimensione richiedono costrutti diversi e Scala fornisce gli strumenti per definirli anziché imporli
  - **Tipi user-defined indistinguibili dai tipi predefiniti** (ridefinizione operatori)
  - Possibilità di **definire nuove astrazioni di controllo**
- **Fusione di stili**
  - **Tutto è un oggetto** – inclusi i tipi (ex) 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**
  - **Mixin composition** fra oggetti e classi (risolve ereditarietà multipla)
  - **Type inference sofisticata**
  - **Shortcut linguistici** e **sintassi infisse** gestite automaticamente

Questo linguaggio è stato ispirato da diversi principi e linguaggi:

- **Forte influenza di Java (e C#)**
  - Espressioni, istruzioni, blocchi, classi, package, tipi, librerie
  - **Stesso modello di esecuzione di Java**
- Ma anche:
  - **Uniform object model** ispirato a Smalltalk & Ruby
  - **Uniform access principle** per metodi e proprietà ispirato a Eiffel
  - **Approccio funzionale** ispirato a ML & co. (incluso F#) e Haskell
  - **Parametri impliciti** ispirati a Haskell
  - **Concorrenza su modello ad attori** ispirati a Erlang
- Più vari **contributi originali:**
  - **Tipi astratti** come alternativa più object-oriented ai tipi generici
  - **Tratti (traits)** come evoluzione delle interfacce → componibilità
  - **Extractor** per estrarre proprietà & pattern matching

20.1 – IDEE DI BASE

TIPI

- **Non esistono tipi primitivi,** sostituiti da **tipi-valore Int, Float…** rimappati sui tipi primitivi Java a livello bytecode per motivi di efficienza
- **Alla radice della gerarchia, Any non è Object**
- Si distingue fin dall'inizio fra **tipi-valori (AnyVal)** e **tipi-riferimenti (AnyRef)**

VALORI E VARIABILI

- **Valori (immodificabili) e variabili (modificabili) sono enti distinti**
  - Valori → val **Un val è intrinsecamente final**: non è più modificabile
  - Variabili → var
- **La specifica di tipo è postfissa** (anziché prefissa come in C e Java), in questo modo è semplice ometterla se non necessaria
- **La specifica di tipo non è obbligatoria** se può essere inferita (e lo è quasi sempre, essendo la type inference molto avanzata)
- **I tipi Java sono comunque tutti disponibili in Scala**

Esempio

Se da terminale scriviamo "scala", appare "Welcome to Scala…"

Se scriviamo 3+4*5 → res0: Int = 23 → crea valore di nome resN (non modificabile)
Se scriviamo println(res0/4) → 5 che può essere riusato
Se scriviamo **var** msg = "ciao" → msg: String = ciao
print(msg) → ciao
msg = "ciao mondo" → msg: String = ciao mondo → var è modificabile
val msg2 = "Mondo" → msg2: String = Mondo
msg2 = "Ciao Mondo" → <console>:8: error: reassignment to val … → val no

FUNZIONI

- **Nuova sintassi** introdotta dalla **parola chiave def**
- **Specifica di tipo postfissa** (anziché prefissa come in C e Java): di solito omessa, se è possibile inferirla
- **Molti shortcut linguistici evoluti**
  - Metodi senza argomenti, volutamente indistinguibili da proprietà
  - Metodi a singolo argomento invocabili in modo infisso, come operatori
  - Caratteri speciali ammessi nei nomi degli operatori
- **OTTIMIZZAZIONE DELLA TAIL RECURSION (DIRETTA)**
- **FUNZIONI COME "FIRST-CLASS ENTITIES"**
  - **Veri oggetti**, istanze di specifici tipi-funzione
  - Manipolabili e innestabili
  - Eventualmente anonime, **in grado di generare vere chiusure**

Esempio def abs(x:Float) : Float = { if (x<0) –x else x }

Stile funzionale: come nell'operatore ternario di C e Java, l'espressione if denota valori: non occorre return
Per usarla con interprete interattivo Scala, devo scrivere: abs(-38.5F) altrimenti errore di tipo
Posso anche omettere il tipo di ritorno perché facilmente deducibile:

def abs(x:Float) = { if (x<0) –x else x }

Si può omettere anche l'uguale: def abs(x:Float) { if (x<0) –x else x }

Una funzione che non ritorna valori (procedura) ha per convenzione tipo di ritorno **Unit**, che è anche il tipo di ritorno degli assegnamenti: ammette come **solo valore** la **costante ().**

def hello(s:String) = { println("Hello" + s) } → hello: (s: String)Unit

In Java: (s=readLine()) != null // s != null
In Scala: (s=readLine()) != "" // sempre true
Scala assegna a s la stringa letta, ma l'assegnamento in sé denota il valore () → "()" diverso da ""

Esempio Array

- Sintassi uniforme con le altre collection → keyword **Array** (() come operatore di accesso).
- Parametrizzazione in tipo con sintassi [tipo] anziché <tipo>

val v = new Array[String](3); (val, quindi array immodificabile, ma non le sue celle)
v(0) = "Paperino";
val v2 = Array("Paperino", "Pippo", "Pluto"); → type inference automatic che riconosce String

CLASSI

- **Costruttore primario come single point of entry**, definito a partire dai **parametri di classe** (nuova sintassi). Si evitano i classici assegnamenti "campo per campo" stile this.x=x
- **Costruttori ausiliari** che **si rimappano sul costruttore primario**
- **Membri public per default** (non package visibility come in Java)
- **Niente più campi e metodi statici**

SINGLETON OBJECTS: nuova sintassi con **parola chiave object** permette di definire oggetti singleton.
COMPANION OBJECTS: **"parte statica" di una classe spostata nel suo oggetto compagno:** singleton omonimo, definito nello stesso file.

Un main Scala non sta in una classe (come in Java), ma in un oggetto si ngleton – un object. A differenza di Java, Scala non ammette membri statici nelle classi. Di conseguenza, **cambia leggermente la dichiarazione:**
"public static void main(String[] args)" diventa "def main(args: Array[String])"
Niente keyword **static**, perché tutti i membri dell'object lo sono.
Nuova keyword **Array** per definire gli array (uniformità collection).

```
object Test1 {
  def main(args: Array[String]) {
      var k=0;
      while (k<5) {
        println("Iterazione " + k)
        k += 1; // l'operatore ++ non esiste
      }
  }
}
```

Dev'essere modificabile → var

```
object Test2 {
  def main(args: Array[String]) {
      var k=0;
      while (k < args.length ) {
        println("Argomento " + k + ": " + args(k))
        k += 1;
      }
  }
}
```

```
<terminated> Test25 [Scala Application] C:\Progra
Argomento 0: La
Argomento 1: nebbia
Argomento 2: agli
Argomento 3: irti
Argomento 4: colli
Argomento 5: piovigginando
Argomento 6: sale
```

TRATTI

- **Evoluzione delle interfacce, possono contenere codice:** favorisce lo sviluppo di rich interfaces senza l'onere di dover sempre implementare tutto: implementazioni di default evitano adapter (~copiato in Java 8)
- **Una classe che eredita da un tratto, ne implementa l'interfaccia, ma contemporaneamente ne eredita il codice**

OLTRE L'EREDITARIETÀ MULTIPLA

- Pur estendendo una sola super-classe (come in Java), **una classe Scala può ereditare da più tratti** con una **nuova semantica** di composizione, chiamata **mix-in**

- **Classi e tratti vengono mixati per linearizzazione**. L'idea di base è: nuova semantica, dinamica, della **parola chiave super**
- Si ottiene come **risultato: stackable behaviour**

PATTERN MATCHING

- Una **nuova sintassi** basata sulle **parole chiave match e case** permette di fare **pattern matching su oggetti e classi**
  - Semplifica notevolmente la scrittura di funzioni che lavorano per casi
  - **Sostituisce ed estende l'antico costrutto switch**

STRUTTURE DI CONTROLLO

Scala ha solo cinque strutture di controllo predefinite: **if, while (e do-while), for, try, match**

- Principio: anziché "più sintassi", meglio "più librerie"
- Altre strutture di controllo possono essere definite dall'utente

Le strutture di controllo denotano valori:

- Espressioni, non istruzioni
- Approccio orientato al funzionale: codice più semplice e leggibile
- Si evitano variabili di appoggio e i bug legati alla loro gestione
- L'if di Scala è sostanzialmente l'operatore ternario di C e Java
- **Preferenza al for (più funzionale) rispetto al while (più imperativo)**

L'assegnamento è un'espressione di tipo Unit: niente assegnamento multiplo (a differenza di C/Java)

Esempio - Foreach

Il metodo **foreach** sfrutta le funzioni come first-class entities. L'operazione da svolgere è passata come argomento-funzione.

DA EXTERNAL ITERATION A INTERNAL ITERATION:

- Cambia il punto di vista: separazione WHAT/HOW
- Il cliente non ha più il controllo dell'iterazione: si limita a specificare COSA fare, non COME farlo
- Il controllo sul come è in mano alla collection, che può adot tare logiche sue (riordinare, lazy eval, etc)

```
object Test4 {
  def main(args: Array[String]) {
     args.foreach( s => println("Argomento: " +s) )
  }
}
```

Lambda expression come argomento di `foreach`

operatore => per definire lambda

```
object Test4 {
  def main(args: Array[String]) {
     args.foreach(println)
  }
}
```

Argomento s omesso (e con lui l'intera lambda!)

È un esempio di *funzione parzialmente specificata*

Riflessione sul **FOR:** nonostante la somiglianza sintattica, il for Scala è ben diverso da quello di Java.

L'argomento s in effetti è un valore, non una variabile: "sembra" cambiare solo perché a ogni iterazione viene creato un nuovo valore, ma non è modificabile dentro il ciclo → **è un val, non un var!**

```
for (s <- args) {
  println("Argomento: " + s)
}
```

Ma soprattutto, **l'espressione interna è un generatore**, cioè un'espressione che genera una serie di valori su cui iterare, e può essere anche molto complessa.

Esempio – For innestati e filtrati

L'espressione generatrice può anche contenere **condizioni di filtro**: quella in figura stampa solo Pippo e Pluto, escludendo Paperino. Sono **ammesse condizioni**

```
val v = Array("Paperino", "Pippo", "Pluto");
for (s <- v if s.length()<=6) println(s)
```

```
val v = Array("Paperino", "Pippo", "Pluto");
for (s <- v
      if s.length()<=6 ;
      if s.contains("i")) println(s)
```

multiple, separate da ';': il secondo for stampa solo Pippo, escludendo Paperino e Pluto.

È possibile perfino esprimere cicli **for innestati e filtrati**, semplicemente concatenando più generatori con ';'

In questo esempio si fa una "grep": si estraggono da più file tutte le righe che soddisfano un certo pattern.

```
def grep(path: String, pattern: String){
  for ( file <- new java.io.File("./" + path).listFiles
        if file.getName().endsWith(".scala");
        line <- scala.io.Source.fromFile(file).getLines.toList;
        if line.trim.matches(pattern)
      ) println(line)
}
```

```
def main(args: Array[String]) {
def main(args: Array[String]) {
def main(args: Array[String]) {
def main(args: Array[String]) {
def main(args: Array[String]) {
def main(args: Array[String]) {
def main(args: Array[String]) {
```

Ad esempio
grep("src",".*main.*")
produce le 7 righe dei miei 7 esempi...

```
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).getLi...oList;
        temp = line.trim;
        if temp.matches(pattern)
      ) yield temp
}
```

Accumula qui i valori *yielded*

Trattiene tutti i risultati parziali

Il for è qui sfruttato come *espressione che ritorna un valore* (in questo caso, di tipo array di stringhe)

```
def main(args: Array[String]) {
  for ( s<- grep("src", ".*main.*") )
      println(s)
}
```

Itera sull'array restituito

Spesso lo scopo di un'iterazione è di accumulare in qualche struttura dati i risultati parziali via via calcolati → Si può farlo facilmente con **keyword yield**

CASE CLASSES

- Una **nuova sintassi** (costrutto case class) **permette di definire classi speciali** (case classes), **che godono di una gestione particolare, largamente automatizzata e semplificata**
  - Si possono creare istanze senza bisogno dell'operatore new
  - Metodi getter generati automaticamente come metodi senza argomenti, così da permetterne l'uso come proprietà
  - Equals, hashcode, toString generati in automatico (con uguaglianza strutturale)

PACKAGE INNESTATI

- **Possibile una doppia sintassi** – sia Java classica, sia C#-like: package ed { package utils { … } }
- A differenza di Java, i package sono **semanticamente innestati**
- **L'identificatore _root_ per il package top level** permette di scrivere anche riferimenti "assoluti"
- **import importa anche singoli oggetti e altri package** (non solo classi e interfacce)
- **import è ricorsiva sui sotto-package**

Esempio - Scope

Scala consente di definire in un inner scope una variabile omonima a una già esistente in un outer scope (shadowing), ma è molto poco leggibile… non abusare.

```
val a = 1
val a = 2          → vietato
{val a = 3}        → ok, è in un inner scope
println(a)         → stampa 1
```

Esempio - Leggibilità

Stile imperativo (foreach *à la Java*)

```
for (s <- args) {
  println("Argomento: " + s)
}
```

Esprime tutto il controllo: per capire il significato del programma, *occorre simulare mentalmente l'esecuzione.*

Stile funzionale (lambda expression)

```
args.foreach( s => println("Argomento: " +s) )
```

Fattorizza il controllo: per capire cosa fa, *basta capire il significato l'espressione passata come argomento*

Stile pienamente funzionale (lambda + funzioni parziali)

```
args.foreach(println)
```

Astrae il controllo: stile dichiarativo, *basta conoscere la semantica di* `println`

20.2 - SINTASSI

Il generatore "0 to 2" usato nell'esempio del for è un caso particolare di uno strumento più generale di Scala: la **sintassi infissa**. La notazione 0 to 2 equivale a **0.to(2)**:

```
val v = Array("Paperino", "Pippo", "Pluto");
for (i <- 0 to 2) println(v(i))
```

- È un altro modo per chiamare un metodo
- In questo caso, il metodo to della classe Int, che infatti si può anche chiamare direttamente:

In generale, **in Scala un metodo con un solo argomento può essere sempre invocato in modo infisso** omettendo il punto della dot notation e le parentesi.
Ancora più in generale, **ogni operatore infisso è una chiamata di metodo**.

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.**
Rientrano in questo sia il caso, già visto, degli array inizializzati sia l'operatore di selezione in lettura delle celle di un array → String s = v(2); equivale a String s = v.apply(2);

Analogamente, **una lista di valori fra parentesi usata come L-value sottintende una chiamata implicita al metodo update dell'oggetto ricevente.**
In particolare, rientra in questa casistica l'operatore di selezione in scrittura delle celle di un array:
v(2) = "Quack!" equivale a v.update(2,"Quack!");

FUNZIONI

- **Procedure** → una funzione che non restituisce nulla è di tipo **Unit**, non void
- **Gli argomenti sono sempre implicitamente val**, non var, perciò non sono modificabili dentro la funzione, Es. def change(i:Int) : Unit = { ~~i = 2~~ }
- **Valore di ritorno implicito** → una funzione (non-void) priva di un'istruzione **return** esplicita restituisce comunque l'**ultimo valore calcolato,** Es. def increment(i:Int) : Int = { tot += i } ritorna tot
- **Se una funzione calcola un solo valore**, le parentesi **graffe** attorno al corpo della funzione sono **opzionali**, Es. def increment(i:Int) : Int = tot += i

Il simbolo ';' si può omettere a fine riga, perché viene dedotto dal compilatore (**semicolon inference**).
Attenzione a possibili effetti imprevisti, es. non lasciare su una riga a sé stante un'istruzione che, presa singolarmente, possa avere significato autonomo diverso.

Esempio

If (a==b)	x+	(x	x
Print("ok")	y	+y)	+y
OK	OK	OK	NO: x ha significato autonomo

Il tipo di ritorno Unit in generale si può omettere, purché si elimini contemporaneamente il simbolo di = e si mantengano le parentesi graffe.
Senza simbolo "=", la funzione sarà sempre interpretata come procedura, anche in presenza di un valore di ritorno, cioè il valore di ritorno andrà perso.

Esempio
def increment (i:Int) : Unit = { i+2 } → def increment (i:Int) { i+2 }
Mancando l'=, si sottintende comunque ":Unit", nonostante il tipo implicito del risultato sia Int, quindi increment(4) non restituirà nulla.

NUMERO VARIABILE DI ARGOMENTI

Scala non consente di definire funzioni con un numero variabile di argomenti (vararg) in modo classico, ma **permette di ripetere l'ultimo argomento, con una sintassi analoga alle regular expression**: *** (asterisco)**
I tipi degli argomenti ripetuti:

- **INTERNAMENTE alla funzione, l'argomento ripetuto è mappato su un array**, quindi il tipo dell'argomento ripetuto è **Array** di quel tipo
- **ESTERNAMENTE**, però, **il tipo argomento ripetuto è considerato diverso** da un array e quindi **incompatibile** con esso

Esempio def myprint(args: String*) { for (a <- args) print(a) }
myprint("3","4","5") //stampa 3 4 5
myprint(Array("3","4","5")) //errore type mismatch → ho chiamato la f con un solo arg
myprint(Array("3","4","5") : _*) //stampa 3 4 5 → ho spacchettato l'array nei suoi elementi

OTTIMIZZAZIONE DELLA TAIL RECURSION

Scala ottimizza di default la tail recursion diretta: si può disabilitare l'ottimizzazione con -g:notailcalls.

È utile confrontare con javap -c il bytecode prodotto:

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

Prestazioni (con n=30)
- 6159 ns vs 7186 ns (guadagno 15%)
- stack length 21 vs 51 (guadagno 60%)

La chiamata alla funzione `f` è scomparsa !
Al suo posto, un `goto` ripete il ciclo

Con TRO
```
 0: iload_2
 1: iload_3
 2: if_icmple 7
 5: iload_1
 6: ireturn
 7: iload_2
 8: iload_1
 9: imul
10: iload_2
11: iconst_1
12: iadd
13: iload_3
14: istore_3
15: istore_2
16: istore_1
17: goto 0
```

Senza TRO
```
 0: iload_2
 1: iload_3
 2: if_icmple 9
 5: iload_1
 6: goto 20 // ireturn
 9: aload_0
10: iload_2
11: iload_1
12: imul
13: iload_2
14: iconst_1
15: iadd
16: iload_3
17: invokevirtual #16
    // Method f
20: ireturn
```

OGGETTI SINGLETON

Un oggetto singleton:

- È introdotto dalla parola chiave **object** anziché class
- È **strutturalmente identico a una classe**
- **Non definisce un tipo**

Questo non significa che un singleton non abbia un tipo: **un singleton estende una classe-base**, del cui tipo, quindi, si considera istanza, perciò, può essere passato a metodi che accettino tale tipo, referenziato da riferimenti di quel tipo, etc.

```
object Boss {
  private val name = "The Boss";
  def getName() = name
}
```
Tutti i campi *devono* essere obbligatoriamente inizializzati
Proprietà e metodi di un singleton sono *pubblici* per default

```
object Main {
  def main(args: Array[String]) {
    println(Boss.getName())
  }
}
```
L'oggetto singleton è usabile senza doverlo creare esplicitamente (come nelle parti statiche Java)

```
val b: Boss = null
not found: type Boss
```
Boss è un oggetto, NON un tipo

Non ammette parametri né costruttori: per questo, tutti i suoi membri devono essere già inizializzati (stessa semantica delle parti statiche di Java). **Può essere usato:**

- Da solo, come **standalone object**
- Insieme a una classe, come suo **companion object**
- **Nella composizione di tratti**

OGGETTI COMPANION

Un oggetto companion **è un particolare singleton**: **omonimo di una classe e definito nel suo stesso file**.

Counter.scala
```
class Counter(v:Int) {
  Counter.howMany += 1;
  ...
}
object Counter {
  private var howMany : Int = 0
  ...
}
```
Accesso privilegiato al campo privato dell'oggetto companion
Variabile privata dell'oggetto singleton

Fra i due si instaura una **speciale relazione**: ognuno dei due può **accedere ai membri privati dell'altro** (come fra la parte statica e non-statica di una classe Java).

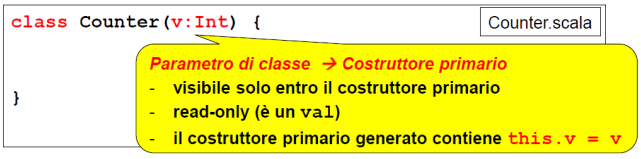
COSTRUTTORI

In Scala **una classe non ha costruttori espliciti:** al loro posto, ha **parametri di classe** usati dal compilatore per generare automaticamente il costruttore primario:

- Evitano la serie di assegnamenti this.nome = nome

- Tutto il codice scritto dentro una classe che non sia dentro a una qualche funzione è considerato codice del costruttore primario

Costruttori ausiliari solo definibili **solo tramite this,** così il **costruttore primario funge da entry point comune.**



Il **COSTRUTTORE PRIMARIO:**

- È unico, costruito a partire dai parametri di classe
- Contiene la serie di assegnamenti this.nome = nome più tutto il codice che non sia dentro a una funzione
- Viene chiamato automaticamente da tutti i costruttori ausiliari → costituisce il single point of entry della classe

Esempio class Counter(v:Int) { **Counter.howMany +=1;** }
Il costruttore primario generato contiene le due righe this.v=v; Counter.howMany += 1;

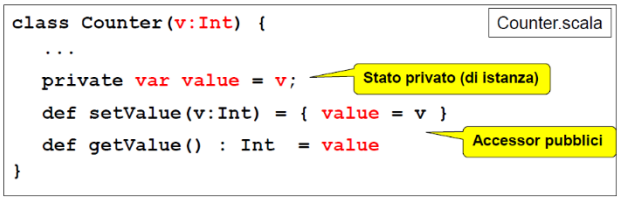
Gli **EVENTUALI COSTRUTTORI AUSILIARI:**

- Sono definibili solo tramite la parola chiave **this** e per questo **si appoggiano tutti**, intenzionalmente, **al costruttore primario**
- Altrettanto intenzionalmente **non possono richiamare il costruttore della classe base,** per non bypassare il single point of entry
- Non a caso, la parola chiave **super non esiste in Scala**

Esempio class Counter(v:Int) { Counter.howMany +=1; **def this() = this(1);** }
In questo caso è un costruttore a zero args e richiama il costruttore primario passandogli "1".

PARAMETRI DI CLASSE vs INSTANCE FIELDS

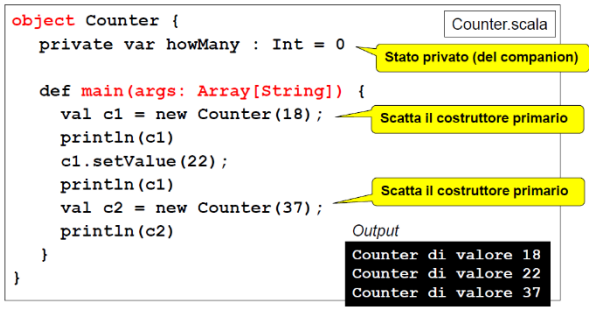
I parametri di classe hanno tempo di vita pari al costruttore primario. Se servono anche dopo la terminazione del costruttore, occorre copiarli in variabili di istanza (come in Java).



Per ridefinire metodi ereditati è richiesto l'uso esplicito del qualificatore **override** (come in C#).

Esempio override def toString() = "Counter di valore " + value

Essendo statico, il main dev'essere posto in un singleton. Ad esempio, nel companion object di una classe.



Il companion object può contenere anche suoi metodi, ad esempio una sua toString() per rivelare quanti Counter siano stati istanziati finora.
Dentro il main basta aggiungere dopo println(c1), l'istruzione **println(Counter.toString())**, evocando quindi il metodo statico del companion.

Esempio - Frazione
Costruttori:

- Costruttore primario a due argomenti (n, d)
- Un costruttore ausiliario a un solo argomento (per gli interi)

- **Precondizioni** tramite l'espressione **require:** se la precondizione espressa da require è falsa, viene generata automaticamente una IllegalArgumentException → altro tassello per ridurre «boilterplate code» e «rumore di fondo»

Dati di istanza: numeratore e denominatore (nun, den)
Metodi:

- Opportuna toString ridefinita
- Operazioni add, sub,.. ecc. o magari (meglio) operatori +, -, …

Frazione.scala

```
class Frazione(n:Int, d:Int) {
  private val num = n;
  private val den = d;
  require(d!=0) // lancia IllegalArgumentException se violata
  def this(n:Int) = this(n,1)
  override def toString = if(d==1) ""+n else n+"/"+d;
  def calcMCD(a:Int, b:Int) : Int =
     if (a%b==0) b else calcMCD(b,a%b)
  def minTerm : Frazione = {
      val mcd = calcMCD(num.abs, den.abs);
      val nuovoNum = num/mcd;
      val nuovoDen = den/mcd;
      new Frazione(nuovoNum, nuovoDen);
  }
...
```

Costruttore ausiliario

if è un'espressione (come l'operatore ternario in Java)

Metodi a zero argomenti → *parentesi tonde opzionali*

return implicita

Frazione.scala

```
...
  def sum(f: Frazione): Frazione = {
      val n = this.num * f.den + this.den * f.num;
      val d = this.den * f.den;
      return new Frazione(n,d).minTerm;
   }
  def +(f: Frazione): Frazione = sum(f)

  def mul(f: Frazione): Frazione = {
      val n = this.num * f.num;
      val d = this.den * f.den;
      return new Frazione(n,d).minTerm;
  }
  def *(f: Frazione): Frazione = mul(f)
...
```

Metodo a 0 argomenti → *tonde opzionali*

Operatori sono normali metodi

Frazione.scala

```
...
  def equals(f:Frazione): Boolean = {
      f.num * this.den == f.den * this.num;
  }
}
```

Come sempre, return inutile..

Frazione.scala

```
object Frazione { // Companion object
  def main(args: Array[String]) {
    val f1 = new Frazione(3,4); println(f1)
    val f2 = new Frazione(2);   println(f2)
    println(f1 + f2)
    println(f1 * f2)
    println(f1 + new Frazione(5,4))
  }
}
```

Sembrano operatori built-in, ma sono normali metodi !

Output

```
3/4
2
11/4
3/2
2
```

- Se la precondizione espressa da **`require`** è falsa, viene lanciata automaticamente ***`IllegalArgumentException`***
  - con messaggio d'errore "`requirement failed`"

Frazione.scala

```
object Frazione {
  def main(args: Array[String]) {
      ...
      val fZ = new Frazione(2,0);  println(fZ)
  }
}
```

```
Exception in thread "main"
java.lang.IllegalArgumentException: requirement failed
        at scala.Predef$.require(Predef.scala:221)
        at Frazione.<init>(Frazione.scala:4)
        at Frazione$.main(Frazione.scala:47)
        at Frazione.main(Frazione.scala)
```

VISIBILITÀ

A differenza di Java, **in Scala i membri delle classi sono pubblici per default.** Sono **privati, invece, i parametri di classe.** Nell'esempio, num e den sono quindi accessibili a tutti (seppur solo in lettura, essendo dei val).
Per accedere posso usare infatti f2.num, ad esempio.

20.3 – SINTASSI SPECIALE

Il backtick rende legale qualunque sequenza di caratteri, ad esempio `while` è un identificatore lecito e indica che "qualunque sequenza" è accettata dal runtime.

Una sequenza di caratteri-operatore è un unico operatore, ad esempio **++?** è un singolo operatore.
Sono caratteri-operatore 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 (es. a<-b).
Java, infatti, separerebbe i token come "a" "<" "-b", Scala invece tokenizza come "a" "<-" "b".
L'**underscore** ha un significato particolare → **introduce gli identificatori misti**

IDENTIFICATORI MISTI

Il carattere underscore introduce i mixed identifiers e ha la sintassi: **parola + underscore + operatore**

- Operatori unari → unary_-
- Metodi setter di proprietà → myprop_= (in coppia con il getter)

Esempio
def unary_-(): Frazione = new Frazione(-this.num, this.den) //da aggiungere alla classe frazione
println(-f1) //da aggiungere al main nel companion Frazione → Stampa -3/4

VAL vs COSTANTI

Un val è immodificabile, ma è pur sempre una variabile, quindi può cambiare valore se re-istanziato durante l'esecuzione, ad esempio come fanno le variabili locali a blocchi, parametri di classe, etc.
Una vera e propria costante invece mantiene sempre lo stesso valore nel tempo:

- In Java, nomi TUTTI_MAIUSCOLI
- In Scala, **standard CamelCase con iniziale maiuscola**

CONVERSIONI IMPLICITE

Come in C++, **Scala consente di definire conversioni implicite di tipo** che saranno applicate automaticamente, soprattutto **per permettere l'uso commutativo degli operatori**.

Esempio def +(i:Int): Frazione = sum(new Frazione(i)) //da aggiungere alla classe Frazione
Vogliamo implementare frazioni che si sommano anche a interi, il metodo + di Frazione somma una Frazione a un'altra, ma è facile definirne una versione overloaded che accetti un intero.
Ciò consente di scrivere espressioni come println(f1+3), **MA NON** println(3+f1) perché + è un metodo di Frazione, non di Int. Per rendere simmetrico l'uso del + occorre definire una conversione implicita → **parola chiave implicit.**
Il metodo di conversione implicita dev'essere definito fuori dalla classe Frazione, nello scope di utilizzo, quindi ad esempio nell'oggetto companion. Il nome del metodo non ha alcuna importanza (qui si chiama conv).

Frazione.scala

```
// Companion object
object Frazione {
   implicit def conv(i:Int): Frazione = new Frazione(i)
   def main(args: Array[String]) {
        ...
     println(f1 + 3) // SI', con operator+ overloaded
     println(3 + f1) // SI', con conversione implicita
   }
}
```

Output

```
15/4
```

ECCEZIONI

Le eccezioni in Scala sono simili a Java, ma:

- Come in C#, **non esiste la clausola throws**, esiste però una **annotation @throws facoltativa** con analoga semantica
- Anche l'espressione **throw ha un tipo di ritorno: Nothing**. Essa, tuttavia, non sarà mai valutata e quindi non produrrà mai un "valore"
- La **clausola catch opera per pattern matching**
  - **Nuova sintassi senza parentesi tonde**, con case interni al blocco
  - **Semantica più evoluta** rispetto al best-type matching di Java
- Anche il **costrutto try/catch/finally è un'espressione**
  - Restituisce il valore del ramo try, se esso ha avuto successo
  - Restituisce il valore di uno dei rami catch, se l'eccezione è stata catturata
  - Non restituisce alcun valore, se l'eccezione è stata lanciata ma non catturata
  - A differenza di Java, **l'eventuale valore computato nel finally non è considerato e va quindi perduto.**

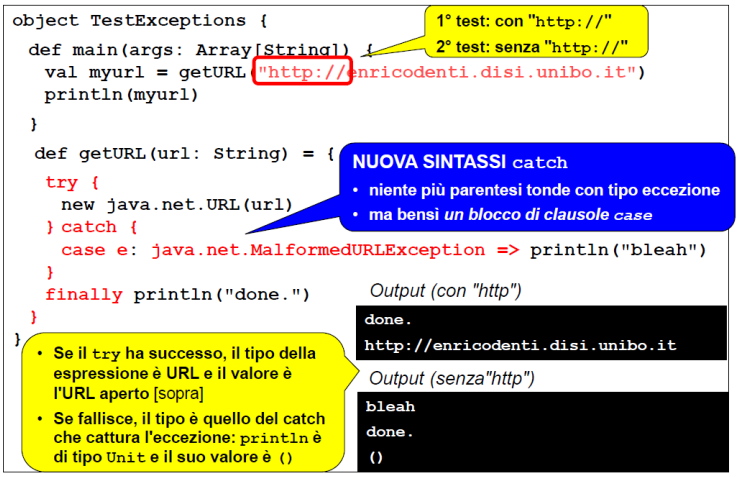
Esempio

Ogni case cattura un caso via pattern matching. Sintassi:

case e: tipoeccezione => espressione

L'espressione è opzionale, ma la freccia => è obbligatoria.

Attenzione alla definizione della funzione: il simbolo '=' è cruciale perché getURL abbia tipo URL. Se omesso, il tipo di ritorno sarà Unit e quindi non ci sarà alcun valore di ritorno.



ESPRESSIONI MATCH

L'obsoleto costrutto switch è sostituito in Scala dalla ben più potente espressione match che, in quanto espressione, **restituisce un valore** e aiuta a separare il "calcolo del caso" dal "cosa fare dopo", quindi anche computazione da interazione.

- **Semantica a casi distinti**, senza "fall through" (**niente break**)
- **Match non solo su valori scalari o stringhe, ma su pattern**
  - Variabili (fanno match con qualunque valore)
  - Costruttori di istanze (fanno match con le istanze con quelle proprietà)
- **Caso di default espresso dal simbolo underscore (_)**
- **Classi speciali** (case classes) per sfruttarne appieno la potenza

Esempio

```
Object TestMatch {      def main(args: Array[String]) {
                              val arg = if (args.length>0) args(0) else “”
                              println( arg match {
                                    case “Pippi” => “Goofy”
                                    case “Paperino” => “Donald Duck”
                                    case _ => arg //untouched } ) } }
```

LETTERALI-FUNZIONE

Un letterale-funzione (lambda) ha la forma generale: **(argomenti) => blocco**

Il **blocco può essere costituito da un'unica espressione** (nel qual caso non richiede parentesi graffe). Se il blocco è costituito da **più espressioni/istruzioni, il tipo e il risultato del letterale sono quelli dell'espressione più a destra.**

Tecnicamente: un letterale-funzione è compilato su classi Function0, Function1, le cui istanze rappresentano a run-time **valori-funzione** di quell'arità.

La funzione può essere eseguita:

- **Direttamente**, tramite l'operatore di chiamata ()
- **Indirettamente**, tramite il metodo apply della classe FunctionN

Esempio

```
val f = (x: Int) => x+1      //ritorna f: Int => <function1> (letterale-funzione semplice, una sola espressione)
f(2)                         //ritorna res0: Int = 3 (chiamata diretta)
f.apply(2)                   //ritorna res1: Int = 3 (chiamata indiretta)
val g = (x:Int) => { print(x); x+1; print(x+1) } //ritorna Int => Unit = <function1> (l-f con blocco, tipo di ritorno Unit)
g(2)                         //ritorna 23 (tipo di ritorno Unit, quindi non stampa risultato)
val h = (x: Int) => {print(x); x+1; print(x+1); x*x} //ritorna h: Int => Int = <function1> (l-f blocco, tipo ritorno Int)
h(2)                         //ritorna 23res4: Int = 4 (stampa il risultato)
```

I letterali funzione (lambda expression) sono utili per passare un comportamento ad altri metodi: particolarmente utile nelle collection per specificare iteratori, condizioni di filtro, etc.

Esempio

```
val v = Array(18, 25, 33, 44, -51)      //ritorna Array[Int] = Array(18, 25, 33, 44, -51)
v.foreach( x => println(x/2) )          //ritorna 9 12 16 22 -25
v.filter( x => x%2==0 =                 //ritorna res12: Array[Int] = Array(18, 44)
```

SINTASSI SHORTCUT

Sono ammesse sintassi shortcut per casi frequenti:

- Se il tipo degli argomenti può essere inferito, si può ometterlo: (x) => x+1
- In tal caso, si possono omettere anche le parentesi tonde: x => x+1
- Se appare una sola volta, si può perfino omettere l'argomento
  - In tal caso, si sopprime anche l'operatore =>
  - Al posto dell'argomento si mette un underscore: _+1

Potenzialmente, l'underscore può rimpiazzare l'intera lista di argomenti, ma attenzione alla **semantica**: ogni underscore sta per UN DIVERSO argomento, non per più occorrenze dello stesso argomento.

Esempio val f = (_ : Int) * (_ : Int) → il primo _ sta per x, il secondo sta per y
Ritorna: f: (Int, Int) => Int = <function2>

```
f(3, 4)   //ritorna res16: Int = 12 perché calcola x*y
f(3)      //dà errore perché il tentativo di interpretarlo come doppia occorrenza di x fallisce
```

FUNZIONI PARZIALMENTE SPECIFICATE

Questo uso dell'underscore è un caso particolare di funzione parzialmente specificata: si definisce una nuova funzione, con meno argomenti, "sopra" un'altra che ne avrebbe di più.

Esempi

```
Funzione base   def sum(a:Int, b:Int, c:Int) = a+b+c                    sum(3,4,5)    → 12
Funzioni parzialmente specificate:
val sum3 = sum _                    (sost. intera lista di args)        sum3(3,4,5)   → 12
val sum2 = sum(10, _:Int, _:Int)                                        sum2(3,4)     → 17
val sum1 = sum2(20, _:Int)          (sost. singolo arg)                 sum1(3)       → 33
val sum0 = sum1(33)                                                     sum0          → 63
```

Principio generale: **l'underscore è sempre richiesto per sostituire l'argomento mancante** (o la lista di argomenti), così si rispetta la signature della funzione. **Eccezione: se l'argomento è esso stesso una funzione, in tal caso non c'è ambiguità**, quindi, la sintassi può essere alleggerita senza rischi.

Esempi

```
val f1 = sum                        → qui underscore non è necessario
val f2 = sum _                      → qui underscore è necessario
Array(3,4,5).foreach(println _)     → qui underscore accettato
Array(3,4,5).foreach(println)       → qui underscore è opzionale perché foreach accetta solo una funzione
```

CHIUSURE

Il pieno supporto alle funzioni come first-class objects comporta un altrettanto **pieno supporto alle chiusure**, adottate con **strategia di chiusura lessicale**.

Esempi

Questa lambda expression contiene una variabile libera val f = (a:Int) => a+**x**

Se eseguita in un ambiente in cui x è sconosciuta, dà errore "not found", mentre in un ambiente in cui x è definita definisce un nuovo valore-funzione: la CHIUSURA, che può essere usato normalmente.

```
val x = 500              //ritorna x: Int = 500
val f = (a:Int) => a+x   //ritorna Int => Int = <function1>
f(21)                    //ritorna res0: Int = 521
```

Le variabili libere sono chiuse su riferimenti alle variabili esterne, cioè percepiscono i loro cambiamenti nel tempo. Quindi, se il valore della variabile libera viene modificato, la chiusura vede il cambiamento:

```
val x = 900              //ritorna x: Int = 900
val f = (a:Int) => a+x   //ritorna Int => Int = <function1>
f(21)                    //ritorna res0: Int = 921
x=44                     //ritorna x: Int = 44
f(21)                    //ritorna res1: Int = 65
```

Le inner classes di Java non consentono proprio di accedere a variabili modificabili dello scope esterno, ma **solo a variabili final** (in Java 8 effectively final, ossia final "de facto", pur senza senza keyword).

Se un ambiente più interno definisce una nuova variabile omonima di quella legata (shadowing), **la variabile libera rimane legata alla variabile originale**. Java non consente lo shadowing.

```
val x = 900              //ritorna x: Int = 900
val f = (a:Int) => a+x   //ritorna Int => Int = <function1>
f(21)                    //ritorna res1: Int = 65
val x=600                //ritorna x: Int = 600   → shadowing var x pre-esistente in uno scope più
f(21)                    //ritorna res2: Int = 65    esterno. Il valore di chiusura non cambia
```

Le chiusure sono quindi un modo efficace per creare effetti permanenti, ad esempio facendo la somma degli elementi di un array.

```
val elenco = Array(12, 23, 45, 65, 2);   var somma = 0;   elenco.foreach(x => somma += x);   somma;
```

Ritorna res4: Int = 147

La lambda expression ha una var libera, quindi si crea una chiusura. Allora, il valore di somma, modificato dalla funzione, sopravvive alla chiusura ed è utilizzabile come accumulatore del risultato.

Queste operazioni si possono fare, più o meno nello stesso modo, anche in <u>Javascript</u>.

In caso vi siano istanze multiple di tali variabili, chiusure diverse ne possono fotografare istanze diverse:

```
def test(list: ListBuffer[Function0[Int]]) : Unit = {   for (i <- 4 to 6) list.append( () => i );  }
var l = ListBuffer[Function0[Int]]();   test(l);   l.foreach( f => println(f()) )
```

Le 3 chiusure catturano ciascuna la i attiva al momento della rispettiva creazione. Stampa 4,5,6.

20.4 - NUOVE STRUTTURE DI CONTROLLO

Scala fornisce gli strumenti per definire nuovi costrutti anziché definirne a priori un insieme fisso e prestabilito:

- Funzioni come first class objects
- Letterali funzione (lambda) e chiusure
- Funzioni parzialmente specificate
- **Block-like syntax** per passare un argomento usando parentesi graffe anziché tonde, in modo che "non sembri una funzione"

- **Call by name** nel passaggio degli argomenti (a richiesta)
- **Currying**, ovvero la possibilità di **applicare a una funzione a più argomenti una serie di argomenti in sequenza, uno dopo l'altro, anziché tutti assieme** nella classica "lista degli argomenti".

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; così facendo, in realtà si definiscono due funzioni:

- Una funzione "esterna" che, dato a, produce una funzione con argomento intero b
- Una funzione "interna" che, dato b, produce il risultato voluto

Es. Definizione: def difC(a:Int) (b:Int) = a-b → Chiamata: val result = difC(8) (3)

difC equivale in effetti a

Definizione: def fExt(a:Int) = (b:Int)=>a-b → Chiamata 1: val fInt = fExt(8)

→ Chiamata 2: val result = fInt(3)

In realtà, non occorre definire tali funzioni esplicitamente: è possibile recuperare il riferimento alla funzione "esterna" usando una sintassi compatta basata sull'underscore. Questo è un **particolare caso di "notazione shortcut"**, che ri-sfrutta ad hoc la nozione di **funzione parzialmente specificata.**

BLOCK-LIKE SYNTAX: **Affinché funzioni definite dall'utente sembrino anche visivamente built-in, Scala permette di usare parentesi graffe anziché tonde in presenza di un solo argomento.**

Forma standard:

```
println("Paperon De' Paperoni")
3.1 + 4     equivalente a      3.1 .+(4)
Array(3)    crea un array di un solo elemento che contiene 3
```

Alternativa con block-like syntax:

```
println {"Paperon De' Paperoni"}
3.1 + {4}   equivalente a      3.1 .+{4}
Array{3}    crea un array di un solo elemento che contiene 3
```

Esempio

```
def raddoppia(v:Double):Double = v*2  //ritorna (v: Double)Double
raddoppia(5)                          //ritorna res8: Double = 10.0. Invocazione classica con le tonde
raddoppia {5}                         //ritorna res9: Double = 10.0. Questo usa block-like syntax
```

Nel primo caso, raddoppia si riconosce essere una funzione definita dall'utente anziché un "vero" costrutto built-in , a causa delle parentesi tonde usate per delimitare l'argomento 5.
Nel secondo caso, invece, la presenza delle parentesi graffe fa sembrare raddoppia una parola chiave che introduce un costrutto "built-in" del linguaggio: **costrutti user-defined diventano indistinguibili da "veri" built-in.**

ARGOMENTI BY NAME

La strategia call by name:

- Valuta un argomento al momento dell'uso, non a priori
- Di norma non disponibile nativamente nei linguaggi imperativi, che adottano il modello applicativo
- Comunque emulabile passando funzioni anziché valori (facile nei linguaggi che supportano funzioni come first-class entities)

In Scala non è necessario emularla: un argomento passa automaticamente by name se preceduto da =>

- Di fatto, è un modo compatto per definire la lambda () => arg
- Sintassi alleggerita sia lato cliente che lato servitore:
- Il cliente passa semplicemente exp anziché () => exp
- La funzione può usare direttamente l'arg ricevuto v anziché doverselo procurare invocando v()

Esempio
La Call by value classica (modello applicativo) sarebbe: **def se(c:Boolean, a:Int, b:Int) : Int = if(c) a else b**
Questa funzione esplode se chiamata con un argomento la cui valutazione causi errore, anche se poi non viene usato. Ad esempio se uso 13/0.

def seBN(c:Boolean, a: =>Int, b: =>Int) : Int = if(c) a else b

Questa funzione invece non esplode se chiamata con un argomento la cui valutazione causi errore, se esso non viene realmente usato: infatti se metto 13/0 al terzo argomento non esplode più, ma dà 12.
Senza questa notazione shortcut, avremmo dovuto gestire in esplicito il passaggio di letterali-funzioni e il loro uso, come in Javascript.

NUOVI BUILT-IN

Ora che abbiamo tutti gli ingredienti, possiamo definire nuovi costrutti di controllo che sembrino "built-in".
Obiettivo: un costrutto repeat con **SINTASSI**: **repeat(n) { istruzioni }** e **SEMANTICA**: ripete n volte le istruzioni.
Impianto:

- **Una funzione che riceva un intero n e una procedura action**, ossia una funzione con tipo di ritorno Unit
- **Al suo interno, se n>1 chiami action e poi richiami se stessa** (tail recursion, che comunque è ottimizzata) **con n-1 e action**
- **Call by name per action**
- **Rifiniture**: currying per separare gli argomenti + block-like syntax per il "tocco finale"

PRIMA VERSIONE (non curried)

Call by name

```
def repeat(n:Int, action: =>Unit) : Unit = {
    action; if (n>1) repeat(n-1, action)
}
```

USO: `repeat(3,print("hi"))`
Sa ancora troppo di funzione", non sembra davvero un costrutto built-in

SECONDA VERSIONE (curried)

Currying

```
def repeat(n:Int)(action: =>Unit) : Unit = {
    action; if (n>1) repeat(n-1)(action)
}
```

Currying

USO: `repeat(3)(print("hi"))`
Va meglio (argomenti disgiunti) ma "sa ancora troppo di funzione".

Per completare l'opera bisogna agire lato chiamante, adottando la block-like syntax per il secondo argomento
USO: repeat(3){ print("hi") } → Ora sì che ci piace!
L'approccio funziona anche con più di un'istruzione nel blocco, perché quello è formalmente il corpo di un letterale-funzione:

Naturalmente, se `repeat(3)` dovesse servire spesso, si può fattorizzare in una funzione parzialmente specificata:
`val rep3 = repeat(3)_`

```
repeat(3){
   val a=12+3; println(a);
}
```

```
rep3 { println("hi"); }
```

```
15
15
15
```

20.5 - 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):

- **Funzione con lista di argomenti vuota** def pluto() : Int = 18
- **Funzione senza lista di argomenti** def pippo : Int = 22
- **Una funzione SENZA lista di argomenti deve essere invocata senza l'operatore ()**
  Es. pippo //ritorna res4: Int = 18 pippo() //ritorna errore
- **Una funzione CON lista di argomenti VUOTA può invece essere invocata con o senza l'operatore ()**
  Es. pluto //ritorna res6: Int = 22 pluto() //ritorna res7: Int = 22

Le funzioni senza lista di argomenti sono importanti perché supportano il **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.

Quando usare funzioni senza argomenti e quando invece con lista di argomenti vuota?

- **Funzioni senza argomenti per i puri accessor**, ossia per funzioni che leggono dati senza modificarli
  - Per retrocompatibilità, eventuali accessor realizzati come funzioni con lista di argomenti vuota possono comunque essere invocati nel nuovo stile
- **Funzioni con lista argomenti (eventualmente vuota) quando non si tratta di puri accessor**, in cui "è bene che si veda" che è una funzione e non un dato

Tecnicamente, le due tipologie si possono "mischiare" in libertà nell' override di metodi: un metodo senza lista di argomenti può sovrascriverne uno ereditata che aveva lista di argomenti vuota – e viceversa.

20.6 – CLASSI ASTRATTE & EREDITARIETÀ

CLASSI ASTRATTE & SOTTOCLASSI

Una classe astratta in Scala è molto simile a Java → parola chiave **abstract** all'inizio della classe, ma non nei singoli metodi.
Una classe derivata si definisce come in Java → parola chiave **extends** nella dichiarazione

- Se la classe base è omessa si sottintende AnyRef (~Object)
- A differenza di Java, **non si ereditano i membri privati** (e neanche quelli omonimi, perché vengono sovrascritti)
- **Per il principio di accesso uniforme, l'indistinguibilità fra metodi e dati implica che un campo-dati possa ridefinire un metodo senza argomenti, e viceversa**
- Metodi e dati appartengono allo stesso namespace: a differenza di Java, **non possono esistere un metodo e un dato omonimi**

Esempio

```
abstract class Forma {
  def area : Double;          // non dimenticare il tipo,
  def perimetro : Double;     // altrimenti inferisce Unit
  override def toString = "Una forma qualsiasi"
}
```
Necessaria

```
class Cerchio(r:Double) extends Forma {
  def area = r*r*Math.PI
  def perimetro = r*2*Math.PI
  override def toString = "Un cerchio di raggio " + r +
                          ", area " + area + " e perimetro " + perimetro
}
```
Keyword override non necessaria qui, perché non c'era niente prima..
Necessaria

Ulteriore variante: campo dati pubblico *automatico*

```
class Cerchietto(val raggio:Double) extends Forma {
  def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override def toString = "Un cerchietto di raggio " +raggio
                          + ", area " + area + " e perimetro " + perimetro
}
```
val introduce un campo dati pubblico *omonimo del parametro di classe*

```
def main(args: Array[String]) {
  val c3 = new Cerchietto(1) ; println(c3)
  println(c3.raggio)        // ACCESSIBILE, è pubblico
}
```

```
Un cerchietto di raggio 1.0, area 3.141592 e perimetro 6.283185
1.0
```

Variante: campo dati pubblico nel costruttore

```
class Cerchione(r:Double) extends Forma {
  val raggio = r
  def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override def toString = "Un cerchione di raggio " + raggio
          + ", area " + area + " e perimetro " + perimetro
}
```
Campo dati pubblico che mantiene lo stato

```
def main(args: Array[String]) {
  val c2 = new Cerchione(1) ; println(c2)
```

Altra variante: campo dati automatico e *modificabile*

```
class Cerchiozzo(var raggio:Double) extends Forma {
def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override def toString = "Un cerchiozzo di raggio " +raggio
          + ", area " + area + " e perimetro " + perimetro
}
```
var introduce un campo dati pubblico *omonimo del parametro di classe e modificabile*

```
def main(args: Array[String]) {
  val c4 = new Cerchiozzo(1) ; println(c4)
  c4.raggio = 2.0          // MODIFICABILE, è var !
  println(c4)
}
```

```
Un cerchiozzo di raggio 1.0, area 3.14159 e perimetro 6.283185
Un cerchiozzo di raggio 2.0, area 12.56637 e perimetro 12.56637
```

Per il principio di accesso uniforme, volendo, è possibile ridefinire toString come val anziché def.
Apparentemente va tutto bene, ma quando andiamo a eseguire, la stampa dopo la modifica del raggio non si adegua più: è diventata una costante → **un val è immodificabile, una funzione è ricalcolata ogni volta!**

```
class Cerchiozzo(var raggio:Double) extends Forma {
def area = raggio*raggio*Math.PI
  def perimetro = raggio*2*Math.PI
  override val toString = "Un cerchiozzo di raggio " +raggio
          + ", area " + area + " e perimetro " + perimetro
}
```

Un cerchiozzo di raggio 1.0, area 3.14159 e perimetro 6.283185
Un cerchiozzo di raggio 1.0, area 3.14159 e perimetro 6.283185

SOTTOCLASSI & COSTRUTTORI

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 parola chiave super.
In Scala, ogni classe ha un solo costruttore primario, gli altri sono tutti ausiliari, rimappati sul primario tramite **this**.
Coerentemente, **una classe derivata può invocare solo il costruttore primario della sua classe-base** e lo fa SENZA usare la parola chiave super (che ha altri usi), ma **indicandolo** direttamente nella dichiarazione **extends** → class Sottoclasse(...) extends ClasseBase(argomenti)

Esempio

```
abstract class Forma(nome:String) {
  def area : Double;
  def perimetro : Double;
  override def toString = nome
}
```

```
class Cerchio(val r:Double) extends Forma(nome) {
  def area = r*r*Math.PI
  def perimetro = r*2*Math.PI
  override def toString = nome + " di raggio " + ...
  def this(v:Double) = this("cerchio", v)
}
```

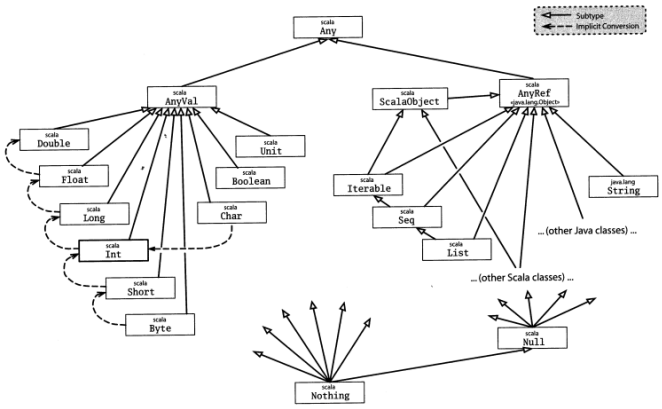
```
def main(args: Array[String]) {
  println(new Cerchio(1));
  println(new Cerchio("Monetina", 0.6))
}
```

```
cerchio di raggio 1.0, area 3.1415926 e perimetro 6.2831853
Monetina di raggio 0.6, area 1.1309734 e perimetro 3.7699112
```

TASSONOMIA SCALA

La classe base Any definisce i metodi generali:

- final def **==** (that:Any) : Boolean
- final def **!=** (that:Any) : Boolean
- def **equals** (that:Any) : Boolean
- final def **eq** (that:Any) : Boolean
- def **hashcode** : Int
- def **hashcode** : Int



Le due **sottoclassi AnyVal e AnyRef fanno da base** rispettivamente ai valori e ai riferimenti, mentre i metodi generali agiscono in modo trasparente su valori e riferimenti. In particolare, **== e != hanno la semantica di equals/not equals:**

- Confrontano valori per AnyVal & figli
- Sono alias per equals e !equals per AnyRef & figli

Le sottoclassi di AnyVal sono tutte astratte e finali:

- **Non si possono estendere né istanziare**, perché i loro valori si scrivono solo come letterali (l'intero 14 è 14, non new Int(14))
- 8 su 9 sono mappate sui tipi primitivi Java corrispondenti; La 9°, Unit, ammette l'unica istanza () già discussa
- Conversioni implicite garantiscono l'interoperabilità fra i tipi

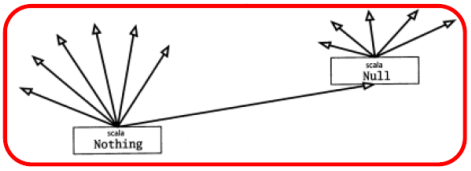
AnyRef è la radice dei tipi-riferimento:

- Nome volutamente distinto da Object per non vincolarsi a Java
- In Java, AnyRef corrisponde in effetti a java.lang.Object, ma ad esempio in .NET corrisponde a System.Object
- L'operatore == non equivale all'uguaglianza fra riferimenti di Java
- L'uguaglianza fra riferimenti è espressa dal metodo eq (final)

Esempi

```
val c: Char  = 'a';  //ok                val c2 = new Char('b');          //errore: class Char is abstract
def uguale (a : Int, b: Int) = a==b     (uguaglianza fra valori)
def uguale(a: Any, b: Any) = a==b       (uguaglianza fra oggetti)
"abc" equals new String ("abc")              //true (uguaglianza tra oggetti)
"abc" == new String ("abc")                  //true (uguaglianza tra oggetti)
"abc" eq new String("abc")                   //false (uguaglianza fra riferimenti)
```

BOTTOM TYPES



Scala definisce 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 il sottotipo di chiunque**: non ha istanze, serve a rendere coerente il type system nelle terminazioni anomale – ad esempio, nelle funzioni d'errore

Esempio

In un "if" i cui rami abbiano tipo diverso, e uno dei due sia Nothing, il tipo dell' if è quello dell'altro ramo.

```
def fermaTutto(causa:String): Nothing = throw new Exception(causa)
def divisione(x:Int, y:Int) ) if(y==0) fermaTutto("divBy0") else x/y        → prevale sopratipo Int
```

20.7 – TRATTI E COMPOSIZIONALITÀ

In Java classico (< Java 8), le interfacce non possono contenere codice né dati (al più, costanti) e grazie a questa limitazione, supportano l'ereditarietà multipla.

Scala supera le interfacce, introducendo i tratti (trait):

- Più simili a classi che a interfacce, **possono contenere codice**
- Come classi e interfacce, **definiscono un tipo**
- Supportano una **nuova forma di composizionalità: il MIX-IN. Una classe estende una superclasse** (ereditarietà singola), **ma può mixarsi con un numero arbitrario di tratti**, che vengono composti assieme mediante linearizzazione ottenendo come risultato: **STACKABLE BEHAVIOUR**

CLASSI & TRATTI

Una classe può:

- **ESTENDERE un solo tratto keyword: extends**
  → se la classe estende un tratto, ne **eredita** la classe base
- **COMPORSI con molti tratti keyword: with**
  → se la classe si compone con un tratto, **NON** ne **eredita** la classe base

La chiave è una diversa semantica della keyword **super:**

- I riferimenti super.x a dati o metodi della **superclasse** sono risolti staticamente, a **compile time**
- I riferimenti super.x a dati o metodi di un **tratto**, invece, sono risolti dinamicamente, a **run time**

INTERFACCE vs TRATTI

Un'interfaccia Java 7/C# è un caso particolare di tratto: un'interfaccia è un tratto senza codice, con sole dichiarazioni. In tal caso, è implementata a livello bytecode come interface.
Ma perché non tenersi le interfacce come in Java 7/C#? Dovendo una classe implementare tutti i metodi di un'interfaccia, spesso si definiscono interfacce molto sottili (SAM-un solo metodo!) a scapito di interfacce ricche, a causa della loro scomodità d'uso → tipico patch: definirle in coppia a opportuni adapter.
In Scala, i tratti consentono un diverso trade-off: un tratto con pochi metodi astratti e molti metodi concreti implementati sopra di essi mixabile con classi a piacere.

EREDITARIETÀ MULTIPLA

Le classi hanno noti problemi con l'ereditarietà multipla, mentre **le interfacce la supportano** perché, essendo senza codice, evitano alla radice i noti rischi di duplicazione dati/metodi.
I tratti bypassano il problema perché, **pur potendo contenere dati o metodi:**

- **Non avendo costruttori, non possono inizializzare dati**
- **C'è sempre e solo un unico costruttore primario** (single entry point) – quello dell'unica classe da cui un tratto eredita
- **La linearizzazione** stabilisce una forma prestabilita e predicibile di composizione di comportamenti e dati in modo stackable

SINTASSI DEI TRATTI

Sintatticamente, un tratto è analogo a una classe, cambia è il modo di concepire il sistema software:

- Parola chiave **trait** anziché class
- **Possibilità di definire dati e metodi, ridefinire metodi, etc.**
- **Niente parametri di classe**, perché non ci sono costruttori

```
abstract class AnimaleTerrestre(n:String) extends Animale(n){
  val vive = "sulla terraferma"
  override def toString = super.toString + " vive " + vive
}
```

```
trait AnimaleTerrestre {
  val vive = "sulla terraferma"
  override def toString = super.toString + " vive " + vive
}
```
Niente parametri di classe

COMPOSIZIONALITÀ DEI TRATTI

Un tratto può estendere una classe (o un altro tratto) → in assenza di specifiche, s'intende Any.
È una specifica di vincolo: potrà essere **mixato solo con sottoclassi della classe che lui stesso estende.**

Ad esempio, se ho il tratto AnimaleTerrestre, dove non è specificata nessuna extends, posso mixarlo con qualunque classe. Se invece ho il tratto Raddoppia che extends Coda[Int], posso mixarlo solo con classi che estendono anch'esse Coda[Int].

Esempio

```
abstract class Animale(nome:String) {
  def vive : String
  def siMuove : String
  override def toString = nome
}
```
Metodi astratti
Chiusura

```
abstract class AnimaleTerrestre(n:String) extends Animale(n){
  val vive = "sulla terraferma"
  override def toString = super.toString + " vive " + vive
}
```
Metodo implementato con un valore
Risolto staticamente

```
class Cavallo(n:String) extends AnimaleTerrestre(n){
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove "
                          + siMuove
}
```
Risolto staticamente

```
MAIN:   println(new Cavallo("Furia del West"));
Furia del West vive sulla terraferma e si muove cavalcando
```

Variante: Animale Terrestre modellato ***con un tratto***

```
abstract class Animale(nome:String) {
  def vive : String
  def siMuove : String
  override def toString = nome
}
```
Metodi astratti
Chiusura

```
trait AnimaleTerrestre {
  val vive = "sulla terraferma"
  override def toString = super.toString + " vive " + vive
}
```
Risolto dinamicamente

```
class Cavallo(n:String) extends Animale(n)
                                with AnimaleTerrestre {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove "
                          + siMuove }
```
Risolto staticamente

```
MAIN:   println(new Cavallo("Furia del West"));
Furia del West vive sulla terraferma e si muove cavalcando
```

Novità: il *quadrupede* modellato con un tratto…

```
trait Quadrupede {
  override def toString = super.toString + " ha 4 zampe"
}
```
Risolto dinamicamente

.. e due tipi-cavalli *composti in modi diversi*

```
class CavalloA(nome:String) extends Animale(nome)
                    with AnimaleTerrestre with Quadrupede {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove " + siMuove
}
```
Risolto staticamente

```
class CavalloB(nome:String) extends Animale(nome)
                    with Quadrupede with AnimaleTerrestre {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove " + siMuove
}
```
Risolto staticamente

L'*ordine di composizione* dei tratti conta → ***stackable behaviour***

Stackable behaviour

```
class CavalloA(nome:String) extends Animale(nome)
                    with AnimaleTerrestre with Quadrupede {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove " + siMuove
}
```
Risolto staticamente

```
class CavalloB(nome:String) extends Animale(nome)
                    with Quadrupede with AnimaleTerrestre {
  val siMuove = "cavalcando"
  override def toString = super.toString + " e si muove " + siMuove
}
```
Risolto staticamente

```
MAIN:   println(new CavalloA("Furia del West"));
        println(new CavalloB("Furia del West"));
```

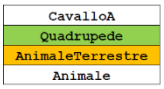
```
Furia del West vive sulla terraferma ha 4 zampe e si muove cavalcando
Furia del West ha 4 zampe vive sulla terraferma e si muove cavalcando
```

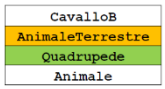
- **Entrambi i cavalli riusano la `toString` di `Animale`**
- ***ma la arricchiscono in modo diverso*** (A+AT+Q+C vs A+Q+AT+C)

Variante: cavalli che *non ridefiniscono* `toString`

```
class CavalloA(nome:String) extends Animale(nome)
                    with AnimaleTerrestre with Quadrupede {
  val siMuove = "cavalcando"
  // NON ridefinisce più toString
}
```

```
class CavalloB(nome:String) extends Animale(nome)
                    with Quadrupede with AnimaleTerrestre {
  val siMuove = "cavalcando"
 // NON ridefinisce più toString
}
```






```
Furia del West vive sulla terraferma ha 4 zampe
Furia del West ha 4 zampe vive sulla terraferma
```
Sparisce l'ultima frase

LINEARIZZAZIONE

Le chiamate a super sono risolte per linearizzazione:

- **Si percorre la sequenza di extends / with** con un criterio prestabilito, **ottenendo una lista di strati**
- **Si compongono le chiamate a super seguendo tale lista**

Nei casi semplici, senza "intrecci", il risultato è uno stack di strati dall'ultimo al primo

Esempio

class CavalloA extends Animale with AnimaleTerrestre with Quadrupede

println(new CavalloA("Furia del West"));

behaviour specifico di CavalloA:	e si muove cavalcando	La frase viene stampata
Quadrupede.toString	ha 4 zampe	in ordine inverso!
AnimaleTerrestre.toString	vive sulla terraferma	
Animale.toString	Furia del West	

Il meccanismo di linearizzazione implementa il mix-in di classi e tratti in modo trasparente ma controllato.
Funziona perché **i tratti non hanno costruttori:**

- Il fatto che un tratto non possa invocare costruttori della classe base permette di **linearizzare l'albero di derivazione**
- **Degli N genitori nominali di un tratto, uno (il primo) può prevalere sugli altri** perché può essere una classe
- È comunque possibile fornire argomenti ai tratti per altra via

Dunque, **una classe Scala non "eredita" da più classi, ma eredita comunque da una sola classe "mixandola" al contempo con altri contributi, che però rimangono su un diverso piano logico e concettuale.**

Esempio

Estendiamo e riorganizziamo l'esempio degli animali, introducendo tratti per catturare singole proprietà rilevanti ed esprimiamo gli animali di interesse combinando i tratti.

```
class 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 {
  def vive = "sulla terraferma"
}
```

```
trait Gambuto extends Animale {
  override def arti = "gambe"
}
```

```
trait Zamputo extends Animale {
  override def arti = "zampe"
}
```

Tratti che catturano animali con mix di proprietà:

```
trait Quadrupede extends Animale {
  override def toString = super.toString + " e ha 4 " + arti
  def siMuove = "correndo"
}
```

```
trait Bipede extends Animale {
  override def toString = super.toString + ", ha 2 " + arti
  def siMuove = "camminando"
}
```

```
trait Altissimo extends Bipede {
  abstract override def arti = super.arti + " lunghissime"
}
```

Un bipede con arti lunghissimi

La doppia qualifica abstract override rende legale la chiamata super.arti anche se tale proprietà è astratta in Bipede.

L'USO di questo tratto, però, sarà ammesso solo in composizioni che includano a un livello sottostante una valida implementazione per arti.

Ora costruiamo un po' di animali reali, mixando tratti:

println(new Animale("Homo3") with AnimaleTerrestre with Bipede with Altissimo with Gambuto)
→ COMPOSIZIONE ERRATA: Altissimo specializza un arti che non esiste prima che sia definito da Gambuto!

```
println(
 new Animale("Zombie") with AnimaleTerrestre with Bipede with Zamputo )
println(
 new Animale("Godzilla") with AnimaleTerrestre with Bipede with Gambuto )
println(
 new Animale("Homo1") with AnimaleTerrestre with Bipede with Gambuto )
println(
 new Animale("Homo2") with AnimaleTerrestre with Bipede
                      with Gambuto with Altissimo)
println(
 new Animale("Fauno") with AnimaleTerrestre with Bipede
                      with Zamputo with Altissimo)
```

COMPOSIZIONI CORRETTE: `Altissimo` **specializza un** `arti`
che esiste perché definito a un livello sottostante

```
Zombie vive sulla terraferma, si muove camminando, ha 2 zampe
Godzilla vive sulla terraferma, si muove camminando, ha 2 gambe
Homo1 vive sulla terraferma, si muove camminando, ha 2 gambe
Homo2 vive sulla terraferma, si muove camminando, ha 2 gambe lunghissime
Fauno vive sulla terraferma, si muove camminando, ha 2 zampe lunghissime
```

ERRORE DI COMPILAZIONE: Overriding method arti in trait Altissimo of type => String
Method arti in trait Gambuto of type => String needs `abstract override' modifiers

Il messaggio d'errore è giusto, ma la cura suggerita no: in questo caso, infatti, etichettando come astratto il metodo arti in Gambuto non ci sarebbero più implementazioni valide, con conseguenti ulteriori errori.

Esempio - Una coda di messaggi configurabile

Una classe base astratta e una possibile implementazione:

```
class MyMsgQueue extends MsgQueue {
  val buf = new scala.collection.mutable.ArrayBuffer[String]
  def get = buf.remove(0)
  def put(s:String) { buf += s}
  def isEmpty = buf.isEmpty
}
```

```
class abstract class MsgQueue {
  def get : String
  def put(s:String):Unit
  def isEmpty : Boolean
}
```

Per quanto sia un po' "basic", l'implementazione funziona:
val q1 = new MyMsgQueue; q1.put("ciao"); q1.put("mondo"); while (!q1.isEmpty) println(q1.get);
//stampa una riga con “ciao” e una riga con “mondo”

Ora definiamo due mini-tratti che specializzano il comportamento: uno inserisce i messaggi trasformandoli in maiuscolo, l'altro invece filtra (fa passare) solo le frasi tutte minuscole.

```
trait Capitalizing extends MsgQueue {
  abstract override def put(s:String) { super.put(s.toUpperCase() )}
}
```

```
trait Filtering extends MsgQueue {
  abstract override def put(s:String) { if (isLowerCase(s)) super.put(s)
}
```

L'approccio "stackable behaviour" permette di comporre questi comportamenti in tutte le quattro combinazioni possibili:

- Coda che inserisce in messaggi in maiuscolo
- Coda che filtra i messaggi minuscoli
- Coda che inserisce i messaggi in maiuscolo dopo averli filtrati
- Coda che inserisce i messaggi filtrandoli dopo averli "maiuscolizzati"

```
val q2 = new MyMsgQueue with Capitalizing
q2.put("ciao");  q2.put("mondo")
while (!q2.isEmpty) println(q2.get);
```

```
CIAO
MONDO
```

```
val q3 = new MyMsgQueue with Filtering
q3.put("ciao");  q3.put("Mondo");  q3.put("crudele")
while (!q3.isEmpty) println(q3.get);
```

```
ciao
crudele
```

```
val q4 = new MyMsgQueue with Filtering with Capitalizing
q4.put("ciao");  q4.put("Mondo");  q4.put("crudele")
while (!q4.isEmpty) println(q4.get);
```

```

```

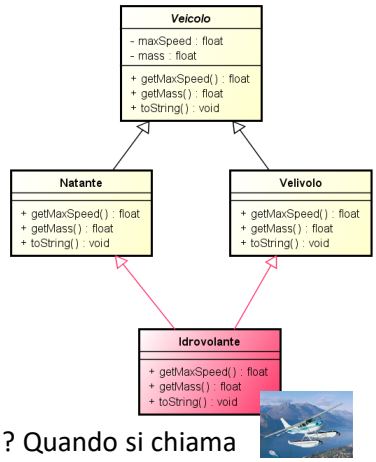
La terza non stampa niente perché non resta niente, dato che prima trasforma tutto in maiuscolo, ma poi fa passare solo quelli minuscoli!

```
val q5 = new MyMsgQueue with Capitalizing with Filtering
q5.put("ciao");  q5.put("Mondo");  q5.put("crudele")
while (!q5.isEmpty) println(q5.get);
```

```
CIAO
CRUDELE
```

DIAMOND INHERITANCE

La diamond inheritance è il tipico esempio dei problemi dell'ereditarietà multipla fra classi.



Esempio

Classe base **Veicolo**

- con proprietà: mass (massa), vmax (velocità max)
- con metodi: getMass, getMaxSpeed, toString

Sottoclassi **Velivolo** e **Natante** → ogni velivolo/natante ha una mass e una vmax

L'idrovolante ha una mass, una vmax o due? Quante vevrsioni di metodi ha? Quando si chiama un metodo, chi viene eseguito?

Possibile scelta per un idrovolante "sensato":

- Una mass, due vmax (quando vola e quando "rulla" sull'acqua)
- Metodi gestiti di conseguenza:
  - getMass non duplicato (ereditato da Veicolo)
  - getMaxSpeed duplicato (due copie ereditate da Velivolo e Natante)
  - toString da decidere (probabilmente, una terza versione ad hoc)

APPROCCI e MECCANISMI

- **C++ usa classi**, permettendo di specificare se duplicare o no ogni elemento – ma obbliga a specificarlo nella classe base!
- **Java e C# usano interfacce**, con codice in classi accessorie che "probabilmente" non ereditano una dall'altra (se non marginalmente)

- **Scala usa tratti e classi, MA ereditando da una sola classe** e quindi componendo codice per linearizzazione dell'albero di derivazione, in modo trasparente e prestabilito

Esempio idrovolante

Una possibile architettura (prima versione)

- **Classe base astratta Veicolo**
  - Tre parametri di classe nome, mass e vmax
  - Una proprietà pubblica nomeclasse (a beneficio di toString)
  - Definisce toString sulla base di questi
- **Due tratti Velivolo e Natante che estendono Veicolo**
  - Estendono Veicolo per ridefinire la proprietà pubblica nomeclasse
  - Definiscono metodi accessori
- **Due classi concrete Idrovolante e Volanteidro che mixano Veicolo con Velivolo e Natante in due modi diversi** non aggiungendo né ridefinendo alcuna proprietà o metodo.

Classe base

```
abstract class Veicolo(nome:String, mass: Float, var vmax: Float) {
  override def toString = classe + " " + nome + " di massa " + mass +
                                  "t e velocità max " + vmax + "km/h"
  val classe: String = "Veicolo"
}
```

```
trait Velivolo extends Veicolo {
  override val classe: String = "Velivolo"
  def setVmaxAsVelivolo(v:Float) = {vmax = v}
}
```

```
class Idrovolante(nome:String, mass: Float, vmax: Float)
           extends Veicolo(nome, mass, vmax) with Velivolo with Natante

class Volanteidro(nome:String, mass: Float, vmax: Float)
           extends Veicolo(nome, mass, vmax) with Natante with Velivolo
```

```
trait Natante extends Veicolo {
  override val classe: String = "Natante "
  def setVmaxAsNatante(v:Float) = {vmax = v}
}
```

Possibile main:

println(new Idrovolante("B737", 40F, 800F)); // Natante B737 di massa 40.0t e velocità max 800.0km/h

println(new Volanteidro("B737", 40F, 800F)); // Velivolo B737 di massa 40.0t e velocità max 800.0km/h

Piccola variazione sul tema: costruiamo un Idrovolante (più Natante che Velivolo) e manipoliamone la velocità massima vmax.

val v1 = new Idrovolante("B737", 40F, 800F); println(v1) //Natante B737 di massa 40.0t e velocità max 800.0km/h

v1.setVmaxAsNatante(10F); println(v1) //Natante B737 di massa 40.0t e velocità max 10.0km/h

v1.setVmaxAsVelivolo(900F); println(v1) // Natante B737 di massa 40.0t e velocità max 900.0km/h

Nel secondo e nel terzo caso sta usando la stessa vmax, quindi - **coerentemente con la specifica di Idrovolante come estensione di Veicolo (pur mixata con Velivolo e Natante) - c'è un'unica istanza della proprietà vmax!**

Refactoring dell'esempio:

```
abstract class Veicolo(nome:String) {
  override def toString = classe + " " + nome + " di massa " + mass +
                                  "t e velocità max " + vmax + "km/h"
  val classe: String = "Veicolo"
  val mass: Float     // abstract val
  val vmax: Float     // abstract val
}
```

```
abstract class Veicolo(nome:String) {
  override def toString = classe + " " + nome + " di massa " + mass +
                                  "t e velocità max " + vmax + "km/h"
  val classe: String = "Veicolo"
  val mass: Float     // abstract val
  val vmax: Float     // abstract val
}
```

```
class Idrovolante(nome:String, mass1: Float, vmax1: Float, vmax2: Float)
           extends Veicolo(nome) with Velivolo with Natante {
  val mass = mass1;
  val vmax = vmax1; // più Natante che Velivolo → vmax as Natante
  val vmaxAsNatante  = vmax1;  // interpreto vmax1 come vmaxAsNatante
  val vmaxAsVelivolo = vmax2;  // interpreto vmax2 come vmaxAsVelivolo
  override def toString = super.toString +
                          ", come velivolo " + vmaxAsVelivolo
  def this(nome:String, mass1: Float, vmax1: Float) =
                          this(nome, mass1, vmax1, 790F)
}
```

valori concreti al posto di quelli astratti

vmax (velivolo) di default

```
class Volanteidro(nome:String, mass1: Float, vmax1: Float, vmax2: Float)
           extends Veicolo(nome) with Natante with Velivolo {
  val mass = mass1;
  val vmax = vmax1; // più Velivolo che Natante → vmax as Velivolo
  val vmaxAsVelivolo = vmax1;  // interpreto vmax1 come vmaxAsVelivolo
  val vmaxAsNatante  = vmax2;  // interpreto vmax2 come vmaxAsNatante
  override def toString = super.toString +
                          ", come natante " + vmaxAsNatante
  def this(nome:String, mass1: Float, vmax1: Float) =
                          this(nome, mass1, vmax1, 10F)
}
```

valori concreti al posto di quelli astratti

vmax (natante) di default

Costruiamo ora 4 Idrovolante (più Natante che Velivolo) e 4 Volanteidro (più Velivolo che Natante):

val i1 = new Idrovolante2("B737", 40F, 10F); println(i1) // Natante B737 di massa 40.0t e velocità max 10.0km/h, come velivolo 790.0

val i2 = new Idrovolante2("B737", 40F, 10F, 820F); println(i2) // Natante B737 di massa 40.0t e velocità max 10.0km/h, come velivolo 820.0

val i3 = new Idrovolante2("B737", 40F, 10F); println(i3) // Natante B737 di massa 40.0t e velocità max 10.0km/h, come velivolo 790.0

val i4 = new Idrovolante2("B737", 40F, 20F, 850F); println(i4) // Natante B737 di massa 40.0t e velocità max 20.0km/h, come velivolo 850.0

val v1 = new Volanteidro2("B737", 40F, 800F); println(v1) // Velivolo B737 di massa 40.0t e velocità max 800.0km/h, come natante 10.0
val v2 = new Volanteidro2("B737", 40F, 800F, 15F); println(v2) // Velivolo B737 di massa 40.0t e velocità max 800.0km/h, come natante 15.0
val v3 = new Volanteidro2("B737", 40F, 900F); println(v3) // Velivolo B737 di massa 40.0t e velocità max 900.0km/h, come natante 10.0
val v4 = new Volanteidro2("B737", 40F, 850F, 20F); println(v4) // Velivolo B737 di massa 40.0t e velocità max 850.0km/h, come natante 20.0

ALGORITMO 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 poi i vari tratti dall'ultimo al primo** (quindi, qui, 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 volta sola**

La risoluzione delle chiamate **super** segue tale lista.

Esempio

class Idrovolante extends Veicolo with Velivolo with Natante
class Volanteidro extends Veicolo with Natante with Velivolo

- Si considera la classe base Veicolo
  - Veicolo → AnyRef → Any
- Si considerano poi Natante e Velivolo singolarmente e si ripete il procedimento escludendo le classi già considerate
  - Natante → Veicolo → AnyRef → Any (già inserite)
  - Velivolo → Veicolo → AnyRef → Any (già inserite)
- Liste linearizzate finali usate per la risoluzione di super:
  - Idrovolante → Natante → Velivolo → Veicolo → AnyRef → Any
  - Volanteidro → Velivolo → Natante → Veicolo → AnyRef → Any

Per vedere all'opera l'algoritmo di linearizzazione in un caso meno ovvio, supponiamo che IdroVolante e VolanteIdro si mixino non già con Velivolo, ma con una sua nuova **estensione Jet.**

```
trait Jet extends Velivolo  {
  override val classe: String = "Jet"
}
```

OSSERVA: un tratto che estende un altro tratto!
(in realtà, estende la sua classe, `Veicolo`)

```
class Idrovolante(nome:String, mass1: Float, vmax1: Float, vmax2: Float)
                    extends Veicolo(nome) with Jet with Natante { ... }

class VolanteIdro(nome:String, mass1: Float, vmax1: Float, vmax2: Float)
                    extends Veicolo(nome) with Natante with Jet { ... }
```

Output:
Jet B737 di massa 40.0t e velocità max 800.0km/h, come natante 10.0
Jet B737 di massa 40.0t e velocità max 800.0km/h, come natante 15.0
Jet B737 di massa 40.0t e velocità max 900.0km/h, come natante 10.0
Jet B737 di massa 40.0t e velocità max 850.0km/h, come natante 20.0

Linearizzando:

- class Idrovolante extends Veicolo with Jet with Natante
- class Volanteidro extends Veicolo with Natante with Jet

Classe base come prima: Veicolo → AnyRef → Any
Natante e Velivolo come prima:

- Natante → Veicolo → AnyRef → Any (già inserite)
- Velivolo → Veicolo → AnyRef → Any (già inserite)

Il nuovo tratto Jet:

- Jet → Velivolo → Veicolo → AnyRef → Any (già inserite)

Liste linearizzate finali usate per la risoluzione di super:

- Idrovolante → Natante → Jet → Velivolo → Veicolo → AnyRef → Any
- Volanteidro → Jet → Velivolo → Natante → Veicolo → AnyRef → Any

Ciò spiega il cambiamento nelle stampe dei Volanteidro ma non negli Idrovolante.

TRATTI VS CLASSI

- Se il comportamento non sarà riusato: **classe concreta**
- Se potrebbe essere riusato in classi scorrelate: **tratto**
- Se una classe Java dovrà ereditare: **classe astratta**. Esprimere da Java l'ereditarietà da un tratto Scala è poco pratico, perché il mapping non è diretto
- Se altri potrebbero estendere (non solo "usare as is") il codice, di nuovo meglio **classe astratta** perché se cambia il codice di un tratto, occorre ricompilare non solo lui, ma anche tutte le classe che lo estendono (anche se formalmente non modificate) → impatto alto
- Se l'efficienza conta molto: meglio una **classe concreta** perché la JVM è più veloce a invocare una classe che un'interfaccia
- Negli altri casi meglio un **tratto** (si potrà cambiare dopo al massimo)

20.8 – SCALA COLLECTIONS

Le Scala collection sono fornite in doppia versione:

- **Mutable** "gentilmente disincentivate"
  - package scala.collection.mutable (non importato di default)
- **Immutable favorite sotto ogni punto di vista**
  - package scala.collection.mutable (importato di default)
  - Nomi corti
  - Operatori shortcut
- **Meno "boilerplate code" rispetto a Java:**
  - Nessuna necessità di ricordare nomi di interfacce e di classi
  - **Uniformità di trattamento rispetto agli array**
  - **Metodo mkString** per ottenere una stringa con tutti gli elementi di una collection, separati da un separatore configurabile

Collezioni disponibili: Liste, Tuple, Set, Mappe

LISTE IMMUTABILI

List rappresenta una lista immutabile: è una classe, non un'interfaccia come in Java, e ha **modello classico: cons, head, tail + costante emptylist.**

- emptylist: Nil
- cons: operatore :: (target object a destra)
- head, tail: metodi head, tail

Operatori e metodi di utilità:

- **Costruzione**: notazione array-like (es. List(2,3,4))
- **Accesso per posizione**: notazione array-like (es. mylist(2))
- **Concatenazione**: operatore ::: (target object a destra)
- E ancora length, count, exists, filter, foreach, forall, map, mkString, remove, reverse, sort, … **ma non append**, perché sarebbe inefficiente su liste immutabili

API riprogettate con first-class functions → semplificati idiomi di uso comune: potenza in poche righe di codice

Esempio

Gli operatori :: e ::: sono metodi dell'oggetto alla loro destra. A ciò si aggiunge una **convenzione sull'associatività: gli operatori sono di norma associativi a sinistra, tranne quelli il cui nome termina con :**

```
val l1 = List("Paperino", "Pippo", "Pluto");    //identica al caso Array, inferisce tipo List[String]
for (i <- 0 to 2) println(v(i))
val l2 = "Qui" :: "Quo" :: "Qua" :: Nil; //Costruzione (cons)
val l3 = l1 ::: l2; //Concatenazione
for (s <- l3) println(s)
val l2 = "Qui" :: "Quo" :: "Qua" :: Nil;        equivale a      val l2 = Nil.::("Qua").::("Quo").::("Qui");
val l3 = l1 ::: l2;                              equivale a      val l3 = l2.:::(l1);
```

METODO mkString

Il metodo mkString produce una stringa con tutti gli elementi di una collection, separati da un separatore **configurabile**: il separatore è l'argomento del metodo.

Esempio

```
l2.mkString(“, ”)        //res10: String = Qui, Quo, Qua
l2.mkString(“,”)         //res11: String = Qui, Quo, Qua
l2.mkString              //res13: String = QuiQuoQua
```

Esempio - Append su Liste

La append è troppo costosa nelle liste immutabili: il tempo cresce in proporzione la dimensione della lista. Possibili alternative:

- Usare cons + reverse
- Usare una **lista mutevole** (ListBuffer) come appoggio, ottenendone una copia immutabile solo alla fine (via toList)
- Convertire la **lista immutabile** in Buffer (via toBuffer), operare su essa (tramite l'operatore di append +=) e poi tornare a List alla fine

```
(“Paperone” :: l1.reverse ). reverse      //res18: List[String] = List(Paperino, Pippo, Pluto, Paperone)
l1.toBuffer += “Paperoga”                 //scala.collection.mutable.Buffer[String]=ArrayBuffer(Paperino,…,Paperoga)
(l1.toBuffer += “Paperoga”).toList        //List[String] = List(Paperino, Pippo, Pluto, Paperoga)
```

TUPLE

La classe Tuple rappresenta una tupla immutabile:

- Entità con elementi di tipi diversi → Costruzione: operatore (…), Accessor: operatore _index
- **Gli elementi di una tupla sono numerati a partire da 1** per una motivazione storica (Haskell e ML)
- **Il loro tipo è una delle classi da Tuple2 a Tuple22**

L'operatore di accesso è _index e non (index) come nelle liste perché i vari elementi di una tupla sono, in generale, di tipo diverso e quindi richiedono metodi diversi con tipi di ritorno differenziati.

Esempio

```
val eta = (“Giovanni”, 22)      //eta: (String, Int) = (Giovanni, 22)
eta._1                          //res25: String = Giovanni
eta._2                          //res26: Int = 22
```

Scala offre molti operatori furbetti su array e tuple.
Capita di avere array da riaccoppiare in modo diverso → da (1,2,3) e ("a","b","c") a ((1,"a"),(2,"b"),(3,"c"))
L'operatore **Array.zip** serve allo scopo e se le dimensioni dei due array sono diverse, vale la minore.

SET

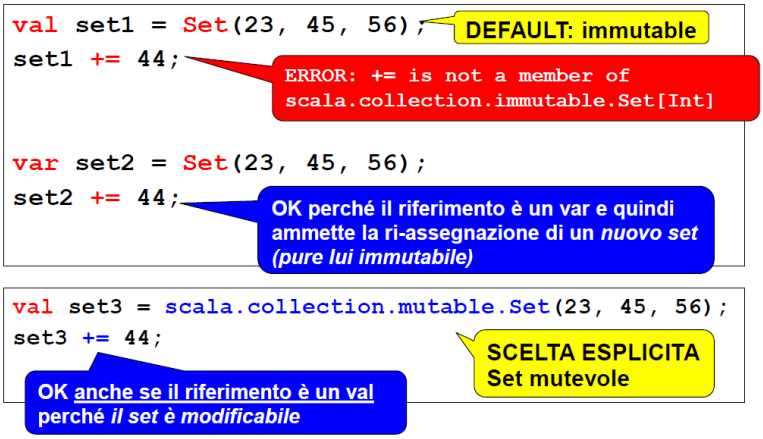
Il tratto Set esiste in doppia versione → **costruzione**: operatore (…), **aggiunta elementi:** operatore +=
La semantica dell'operatore +=:

- **Nei set mutevoli** esiste un vero operatore +=
- **Nei set immutabili** invece non esiste, ergo += è interpretato come shortcut per la creazione di un nuovo set (come nelle stringhe Java), che è corretto purché il riferimento sia dichiarato var

Set è implementato dalla **classe HashSet** e per passare dalla versione mutable a immutable (o viceversa), basta cambiare la import.

Esempio
Ma se Set è un tratto, com'è possibile che si possa istanziare?

val set1 = Set(23, 45, 56);
val set3 = scala.collection.mutable.Set(23, 45, 56); set3 += 44;



In realtà, non stiamo "istanziando" un'interfaccia: stiamo soltanto invocando un factory method dell'oggetto companion! **La frase Set(…) è uno shortcut linguistico per l'invocazione del metodo apply del companion object di Set.** → val set1 = Set(23, 45, 56); **equivale a** Set.apply(23, 45, 56)

MAPPE

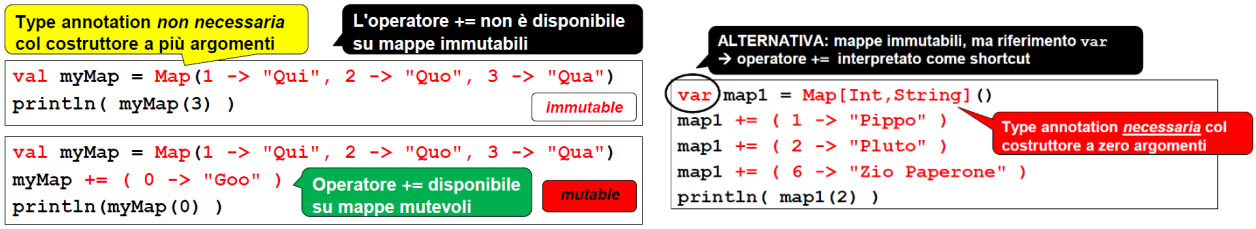
Il tratto Map esiste in **doppia versione** → **costruzione**: operatore (…), **aggiunta elementi**: operatore +=

- Stesse precauzioni di Set sulla semantica dell'operatore "aggiunta":
- **Implementato dalla classe HashSet**
- Anche qui Map(…) è uno shortcut per il factory method apply del companion object
- Anche qui, per passare da mutable a immutable (o viceversa), basta cambiare la import
- **Aggiunta elementi tramite tuple speciali key -> value**

Occorre però **attenzione ai tipi:**

- I set ospitano oggetti omogenei → tipo deducibile (type inference)
- **Nelle mappe, il tipo di chiavi e valori non è sempre inferibile** → può essere necessario specificarlo esplicitamente

Esempio



Poiché in Scala ogni operatore è un metodo, **la sintassi key -> value equivale a key.->(value)**
myMap += 0 -> "Goo" equivale a myMap += (0).->("Goo") //ritorna map += (5). -> (“Gee”)
Le parentesi servono ad evitare che 5 . sia inteso come Double

20.9 – ALTRE FEATURE INTERESSANTI

VARIANZA & COVARIANZA

Soluzione più sofisticata rispetto a Java:

- In **Java**: soluzione semplificata (wildcards)
- In **Scala**:
  - lower bounds [U >: T] per tipi covarianti [+T]
  - upper bounds [U <: T] per tipi controvarianti [-T]

Il compilatore verifica che l'utilizzo dei bound sia coerente con la specifica di varianza del tipo.

class Coda[+T](...) { def append[U >: T](element: U) {..} ... }

ARGOMENTI AVANZATI

- Case classes e pattern patching
- Extractors
- Interoperabilità Scala/Java
- Modello ad attori per la concorrenza
- Supporto per Domain Specific Languages nella forma di parser combinators
- Funzioni e oggetti definiti in Scala
- Usabili come "building blocks" per un parser

GRAFICA

Scala NON ha un suo supporto, si appoggia su Java (es. ScalaFX) o altre librerie e progetti (github)