LINGUAGGI E MODELLI COMPUTAZIONALI M

APPUNTI DI FRANCESCO PENNELLA
ANNO 2016/2017

1ª PARTE: TEORIA E RICONOSCIMENTO DEI LINGUAGGI

00. INTRODUZIONE

Analisi sintattica → verifica della correttezza strutturale dei comandi

Analisi semantica → verifica del significato dei comandi

Spazio concettuale → l'insieme di metafore e concetti introdotti da un linguaggio (oltre alle regole sintattiche e semantiche) costituisce il suo spazio concettuale. Il diverso spazio concettuale di ogni linguaggio è la ragione di fondo per l'esistenza di tanti linguaggi.

> Ad esempio: *un linguaggio può essere più adatto di un altro come strumento per uno specifico settore e un altro può esprimere meglio le entità di un certo dominio applicativo.*

Stile di programmazione → ogni modello computazionale promuove uno specifico stile di programmazione, caratterizzante per i linguaggi di tale famiglia. L'adozione i uno stile di programmazione ha conseguenze sulle proprietà dei programmi, sulla metodologia di soluzione dei problemi, sulle caratteristiche dei sistemi software.

Programmazione multi paradigma → proprio perché ogni modello computazionale ha i suoi punti di forza, può aver senso usarne più di uno. Parti diverse di applicazione potrebbero beneficiare di approcci diversi e stili diversi.

01. LINGUAGGI E MACCHINE ASTRATTE

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 (istruzioni) 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. Consiste nella formulazione testuale di un algoritmo in un dato linguaggio di programmazione.

Automa esecutore → una macchina astratta capace di eseguire le azioni dell'algoritmo. L'esecuzione delle azioni specificate nell'algoritmo porta ad ottenere a partire dai dati in ingresso la soluzione del problema. Deve ricevere dall'esterno una descrizione dell'algoritmo e deve essere capace di interpretare un linguaggio

Macchina base <I,O,mfn>
Automa a stati finiti <I,O,S,mfn,sfn>
Macchina di turing <A,S,mfn,sfn,dfn>

Tesi di Church Turing → non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella risolta dalla macchina di turing

Macchina di turing universale → l'algoritmo cablato sulla MDT si limita a leggere dal nastro una descrizione dell'algoritmo che serve (si comporta come un Loader), andando a caricare ogni volta un nuovo programma dal nastro. In conclusione la UMT non è altro che un interprete di linguaggio (come l'automa esecutore).

UMT e macchina di Von Neumann → La UMT non cattura tutti gli aspetti dell'architettura di Von Neumann. La UMT infatti elabora prendendo dal nastro dati e algoritmo e scrivendo sul nastro i risultati, ossia opera solo sulla memoria RAM, poiché non ha il concetto di mondo esterno (non ha istruzioni di I/O). La UMT è pura computazione: non modella la dimensione dell'iterazione, che invece esiste nella macchina di Von Neumann.

Problema irresolubile → se neanche la macchina di Turing riesce a risolvere un problema, quel problema è irresolubile. (la MDT non si ferma)

Problema risolubile → un problema la cui soluzione può essere espressa da una MDT. (la MDT riesce a computare la funzione caratteristica di quel problema)

Funzione caratteristica di un problema → dato un problema P e X l'insieme dei dati in ingresso e Y l'insieme delle risposte corrette, si dice funzione caratteristica del problema P la funzione f: X -> Y

Funzione computabile → una funzione f: A -> B è computabile se esiste una MDT che data sul nastro una rappresentazione di A, dopo un numero finito di passi produce sul nastro f(x) appartenente a B

Procedimento di Godel → data una collezione di numeri naturali, questa p esprimibile con un unico numero naturale, grazie alla scomposizione in fattori primi.

> Siano N1, N2, .. Nk i numeri naturali dati
> Siano P1, P2, .. Pk i primi k numeri primi
> Sia R un nuovo numero naturale definito da $\mathbf{R = P1^{N1} * P2^{N2} * .. * Pk^{Nk}}$

Problema dell'HALT della MDT → stabilire se data una macchina di turing con un generico ingresso, si ferma oppure no. Questo problema è perfettamente definito ma non è computabile.

> $f_{HALT}(m,x) = 1$, se m con ingresso x si ferma
> $f_{HALT}(m,x) = 0$, se m con ingresso x non di ferma

Questa funzione caratteristica è perfettamente definita ma non è computabile poiché tentare di calcolarla conduce a un assurdo.

Se f_{HALT} è computabile allora esiste una MDT capace di calcolarla. Sia allora n il numero naturale che tramite il procedimento di Godel identifica quella macchina. Definiamo ora una nuova funzione $g_{HALT}(n)$:

> $g_{HALT}(n) = 1$, se $f_{HALT}(n,n) = 0$
> $g_{HALT}(n) = |$, se $f_{HALT}(n,n) = 1$

La funzione g si ferma e risponde 1 quando la MDT n con ingresso n non si ferma. Al contrario la g non si ferma ed entra in loop, quando la MDT n con ingresso n si ferma.

Insieme numerabile → è un insieme i cui elementi possono essere contati, ossia che possiede una funzione biettiva che mette in corrispondenza i numeri naturali con gli elementi dell'insieme.

Insieme ricorsivamente numerabile → un insieme è ricorsivamente numerabile (semi-decidibile) se tale funzione è computabile.

Insieme decidibile → un insieme è decidibile se la sua funzione caratteristica è computabile. Quindi esiste una MDT che è capace di rispondere SI o No senza entrare mai in un ciclo infinito se un certo elemento appartiene all'insieme.

> **Teorema 1** → se un insieme è decidibile è anche semi decidibile ma non viceversa.
>
> **Teorema 2** → un insieme S è decidibile se e solo se sia S che il suo complemento N-S sono semi decidibili.

Sintassi → l'insieme delle regole formali per la scrittura di programmi di un linguaggio, che dettano le modalità per costruire frasi corrette nel linguaggio stesso.

Semantica → insieme dei significati da attribuire alle frasi (sintatticamente corrette) costruite nel linguaggio

Interprete → un interprete per un linguaggio L accetta in ingresso le singole frasi di L e le esegue una per volta. Il risultato è la valutazione della frase

Compilatore → un compilatore per un linguaggio L accetta in ingresso un intero programma scritto in L e lo riscrive in un altro linguaggio. Il risultato è dunque una riscrittura della macro-frase (il programma)

Analisi lessicale → consiste nella individuazione delle singole parole (token) di una frase. L'analizzatore lessicale è detto scanner, e data una sequenza di caratteri li aggrega in token.

Analisi sintattica → consiste nella verifica che la frase intesa come sequenza di token, risetti le regole grammaticali del linguaggio. L'analizzatore sintattico è detto parser, e data la sequenza di token prodotta dallo scanner, genera una rappresentazione interna della frase.

Analisi semantica → consiste nel determinare il significato di una frase. L'analizzatore semantico data la rappresentazione interna prodotta dal parser, controlla la coerenza logica della frase.

Chiusura A* di un alfabeto A → è l'insieme di tutte le stringhe con simboli di A

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

Chiusura positiva A+ di un alfabeto A → è l'insieme infinito di tutte le stringhe non nulle composte con simboli di A.

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

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 e S è un particolare simbolo non-terminale detto scopo della grammatica.

Forma di frase → una qualsiasi stringa comprendente sia simboli terminali sia meta simboli, ottenibile dallo scopo applicando una o più regole di produzione.

Frase → una forma di frase comprendente solo simboli terminali

Derivazione → si dice che β deriva direttamente da α (α->β) se le stringhe α, β si possono decomporre in: $\alpha = \eta A \delta$ $\beta = \eta\gamma\delta$ ed esiste la produzione A->γ. Si dice quindi che β deriva da α. Se la derivazione da α a β si svolge in più passaggi allora parliamo di derivazione indiretta

$S \rightarrow \sigma$ derivazione con un solo passo

S +-> derivazione in uno o più passi
S *-> derivazione in zero o più passi

Grammatiche equivalenti → due grammatiche si dicono equivalenti se generano lo stesso linguaggio.

Grammatica di tipo 0 → nessuna restrizione sulle produzioni.

È il caso più generale in cui le regole di derivazione le puoi fare come vuoi, quindi non ci sono restrizioni, però questo può voler dire che non esista nessuna MDT in grado di riconoscerla. Tra le regole è anche possibile accorciare la lunghezza della stringa poiché vengono tolti alcuni elementi.

Grammatica di Tipo 1 (dipendenti dal contesto) → produzioni vincolate alla forma $\beta A\delta \rightarrow \beta\alpha\delta$ con $\beta,\delta,\alpha \in (VT \cup VN)^*$, $A \in VN$ e $\alpha \neq \varepsilon$

Quindi A può essere sostituito con α solo se si trova nel contesto βAδ. In questo caso le riscritture non accorciano mai la forma di frase corrente.

DEFINIZIONE ALTERNATIVA: $\alpha \rightarrow \beta$ con $|\beta| \geq |\alpha|$
Esprime lo stesso concetto ma in un modo più pratico poiché non esplicita l'idea di contesto.

Grammatica di Tipo 2 (context free) → produzioni vincolate alla forma $A \rightarrow \alpha$ con $\alpha \in (VT \cup VN)^*$, $A \in VN$.

Qui A può sempre essere sostituita da α, indipendentemente dal contesto, poiché non esiste più l'idea stessa di contesto. Sul lato sinistro (A) c'è un simbolo solo e non più una sequenza di caratteri, quindi non è più consentito lo scambio tra i simboli.

Nelle grammatiche di tipo 0 e 1 era importante evitare la cancellazione poiché c'erano regole complesse, nel Topo 2 invece sono tutte regole molto semplici e quindi la cancellazione non provoca danni.

Grammatica di Tipo 3 (grammatiche regolari) → produzioni vincolate alle forme lineari: a destra $A \rightarrow \sigma$, $A \rightarrow \sigma B$ oppure a sinistra $A \rightarrow \sigma$, $A \rightarrow B\sigma$.

Si intende che le produzioni di una data grammatica devono essere tutte o lineari a destra o lineari a sinistra, non mischiate.

Una frase di questa grammatica può crescere solo in fondo se è lineare a destra o solo all'inizio se è lineare a sinistra. Se la grammatica cresce un po' a destra e un po' a sinistra allora sarà sicuramente del Tipo 2.

Stringa vuota → le grammatiche di tipo 1 non ammettono la stringa vuota ε sul lato destro delle produzioni poiché accorcerebbero la lunghezza delle frasi. Viceversa le grammatiche di tipo 2 e di tipo 3 la ammettono.

Teorema: le produzioni di grammatiche di tipo 2 (e quindi anche di tipo 3) possono essere riscritte in modo da evitare la stringa vuota: al più possono contenere la regola $S \rightarrow \varepsilon$

Teorema: se G è una grammatica context free (3) con produzioni della forma $A \rightarrow \alpha$, cioè α può essere uguale a ε, allora esiste una grammatica context free G' che genera lo stesso linguaggio ma le cui produzioni hanno o la forma $A \rightarrow \alpha$ con $\alpha \in V^+$ oppure in forma $S \rightarrow \varepsilon$ ed S non compare sulla destra di nessuna produzione.

Self-embedding → una grammatica contiene self-embedding quando una o più produzioni hanno la forma A *-> α1 A α2 con α1,α2 appartenenti a V+. il self embedding è la caratteristica che rende una grammatica di tipo" diversa da una di tipo 3.

Il self-embedding introduce una ricorsione in cui si aggiungono elementi contemporaneamente sia a destra che a sinistra, procedendo di pari passo.

Riconoscibilità dei linguaggi → i linguaggi generati da grammatiche di tipo 0 possono in generale non essere riconoscibili. Al contrario i linguaggi generati da grammatiche di tipo 1 (e quindi di tipo 2 e 3) sono riconoscibili.

Macchine e linguaggi → per i linguaggi di tipo 2 verranno utilizzate le Push Down Automaton (PDA) cioè un Automa a stati finiti (ASF) con lo stack, mentre per i linguaggi di tipo 3 verranno utilizzato un automa a stati finiti.

Grammatica BNF → in una grammatica BNF le regole di produzione hanno la forma α::=β, mentre i meta simboli X appartenenti a VN hanno la forma <nome>. Il meta simbolo | indica l'alternativa.

Grammatica EBNF → la notazione Extended BNF è una forma estesa della notazione BNF rispetto a cui introduce alcune notazioni compatte per alleggerire la scrittura delle regole di produzione.

X::=[a]B	->	X::=B|aB
X::={a}^{n}B	->	X::=B|aB| .. | a^{n}B
X::={a}B	->	X::=B|aX
X::=(a|b)D|c	->	X::=aD|bD|c

Alberi di derivazione → per le grammatiche di tipo 2 si introduce il concetto di albero di derivazione, dove: ogni nodo è associato a un simbolo, la radice dell'albero coincide con lo scopo S. Se a1, a2, .. ak sono i figli di X allora esiste la produzione X::=A1 A2 .. Ak, con A_i simbolo associato ad a_i

Grammatica ambigua → se esiste almeno una frase che ammette due o più derivazioni canoniche sinistre distinte. Ciò ammette due alberi di derivazione diversi anche se utilizzo la stessa tecnica di derivazione.

Forma normale di Chomsky → produzioni della forma A->BC|a con A,B,C ∈ VN e α ∈ VT con anche ε

Forma normale di Greibach → produzioni della forma A->aα con A,B,C ∈ VN e a ∈ VT con anche ε e α ∈ VN*(per linguaggi privi di ε)

Trasformazioni importanti → per facilitare la costruzione dei riconoscitori è spesso rilevante poter trasformare la struttura delle regole di produzione per renderle più adatte allo scopo.

Sostituzione: consiste nell'espandere un simbolo non terminale che compare nella parte destra di una regola di produzione, sfruttando a tale scopo un'altra regola di produzione.

Raccoglimento a fattor comune: consiste nell'isolare il prefisso più lungo comune a due produzioni.

Eliminazione della ricorsione sinistra: è una trasformazione sempre possibile divisa in due fasi. FASE!: eliminazione dei cicli ricorsivi a sinistra e FASE2: eliminazione della ricorsione sinistra diretta.

Pumping lemma → Il Pumping lemma fornisce una condizione necessaria ma non sufficiente perché un linguaggio sia di tipo 2 (o 3). In un linguaggio infinito, ogni stringa sufficientemente lunga deve avere una parte che si ripete, cioè essa può essere pompata un qualunque numero di volte ottenendo sempre altre stringhe del linguaggio.

Pumping lemma per linguaggi di tipo 2 → 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
- La parte centrale vwx ha lunghezza limitata
- V e x non sono entrambe nulle
- V e x possono essere pompate quanto si vuole ottenendo sempre frasi del linguaggio in questione. Ossia xy^nwx^mz

Pumping lemma per linguaggi di tipo 3 → se L è un linguaggio di tipo 3 eiste un intero N tale che per ogni stringa z di lunghezza almeno pari a N:

- Z può essere riscritto come: z=xyw
- La parte centrale xy ha lunghezza limitata
- Y non è nulla
- La parte centrale può essere pompata quanto si vuole ottenendo sempre frasi del linguaggio in questione. Ossia xy^nw

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 appartenente ad A è una espressione regolare
- Se X e T sono espressioni regolari lo sono anche, X+Y (unione), X.Y (concatenazione), X* (chiusura).

Espressioni e linguaggi regolari → i linguaggi generati da grammatiche regolari (tipo 3) coincidono con i linguaggi descritti da espressioni regolari

La grammatica è COSTRUTTIVA cioè dice come si fa, non cosa si ottiene, mentre le espressioni regolari sono DESCRITTIVE cioè dicono cosa si ottiene e non come si fa.

Dalla grammatica all'espressione regolare → si risolvono le cosiddette equazioni sintattiche

Per passare dalla grammatica all'espressione regolare si interpretano:

- le produzioni come equazioni sintattiche in cui, i simboli terminali sono i termini noti
- i linguaggi generati da ogni simbolo non terminale sono le incognite e si risolvono con le normali regole algebriche.

Dall'espressione regolare alla grammatica → si interpretano gli operatori dell'espressione regolare in base alla loro semantica (sequenza, ripetizioni, alternativa) mappandoli in opportune regole).

Per passare dall'espressione regolare alla grammatica si interpretano gli operatori in base alla loro semantica:

- Sequenza → simboli accostati nella grammatica
- Operatore + → simbolo di alternativa nella grammatica
- Operatore * → regola ricorsiva nella grammatica

03. AUTOMI RICONOSCITORI

Riconoscibilità dei linguaggi → i linguaggi generati da grammatiche di tipo 0 possono in generale non essere riconoscibili. Al contrario i linguaggi generati da grammatiche di tipo 1 (e quindi di tipo 2 e 3) sono riconoscibili

Perché il traduttore possa essere realizzato in modo efficiente conviene adottare linguaggi generati da grammatiche di tipo 2. Per ottenere poi maggio efficienza in sotto parti di uso estremamente frequente, si adottano spesso per esse linguaggi generati da grammatiche di tipo 3.

> Tipo 2 -> Push Down Automaton (Automa a stati finiti + stack)
> Tipo 3 -> Automa a stati finiti (ASF)

Automa a stati finiti → un linguaggio regolare è riconoscibile da un Automa a stati finiti. Un automa a stati finiti si definisce come una quintupla <I,O,S,mfn,sfn> dove I è l'insieme dei simboli in ingresso, O è l'insieme dei simboli in uscita, S è l'insieme degli stati, mfn (IxS -> O) è la funzione macchina e sfn (IxS -> S) è la funzione di stato.

Riconoscitore a stati finiti → rappresenta una specializzazione di un ASF, e viene definito da una quintupla <A,S,S0,F,sfn*> cove A è un alfabeto, S è l'insieme degli stati, S0 è lo stato iniziale, F è l'insieme degli stati finali e la funzione sfn* (A*xS -> S) è la funzione di stato.

> Sfn* definisce l'evoluzione dell'automa a partire dallo stato iniziale S0 in corrispondenza a ogni sequenza di ingresso x appartenente ad A*.

Frase accettata da un RSF → una frase x che porta il riconoscitore a partire dallo stato iniziale in uno stato finale.

Teorema1 (caccia allo stato finale) → un linguaggio ® è non vuoto se e solo se il riconoscitore R accetta una stringa x di lunghezza Lx minore del numero di stati N dell'automa.

Teorema2 (caccia al ciclo) → un linguaggio L® è infinito se e solo se il riconoscitore R accetta una stringa x di lunghezza N >= Lx < 2N con N numero si stati dell'automa.

Dai riconoscitori ai generatori → si può automatizzare la costruzione di un RSF a partire dalla grammatica, o viceversa, risalire a una grammatica dato un RSF. Si può definire un mapping tra:

- stati <-> simboli non terminali
- transizioni <-> produzioni
- scopo <-> uno stato particolare

L'approccio intuitivo è quello di immaginare un osservatore che stando in ogni stato guarda in avanti (grammatica regolare a destra) o indietro (grammatica regolare a sinistra. Nel primo caso vede la freccia e lo stato successivo, nel secondo caso vede la freccia e lo stato precedente.

Riconoscitori top-down e bottom-up → se la grammatica è regolare a destra si ottiene un automa riconoscitore top-down, al contrario se la grammatica è regolare a sinistra si ottiene un automa riconoscitore bottom-up.

Riconoscitori top-down → data una grammatica regolare lineare a destra il riconoscitore ha tanti stati quanti sono i simboli non terminali, ha come stato iniziale lo scopo S. Inoltre per ogni regola del tipo X->xY l'automa con ingresso x si porta dallo stato X a Y. Mentre per ogni regola del tipo X->x l'automa con ingresso x, si porta dallo stato X a quello finale.

Riconoscitori bottom-up → data una grammatica regolare lineare a sinistra il riconoscitore ha tanti stati quanti sono i simboli non terminali, ha come stato finale lo scopo S. Inoltre per ogni regola del tipo X->Yx l'automa con ingresso x si porta dallo stato Y a X. Mentre per ogni regola del tipo X->x l'automa con ingresso x, riduce lo stato iniziale I allo stato X.

Dall'automa alle grammatiche → dato un automa riconoscitore se ne possono trarre: una grammatica regolare a destra, interpretandolo top-down e una grammatica regolare a sinistra interpretandolo bottom-up.

Nel caso bottom-up, in presenza di più stati finali:

- si assume come sotto-scopo Sk uno stato finale alla volta
- si scrivono le regole bottom-up corrispondenti
- si esprime poi il linguaggio complessivo come unione dei vari sotto-linguaggi definendo lo scopo globale come S->S1|S2|..|Sn

Implementazione di RSF deterministico → un riconoscitore a stati finiti deterministico è facilmente realizzabile in un linguaggio imperativo.

- Ciclo while con serie di if
- Ciclo while con switch
- Ciclo while con tabella separata (Hash map o oggetto). In questo modo separo la business logic dal funzionamento, quindi è riutilizzabile e l'automa non è più cablato nel codice.

Automa non deterministico → certe grammatiche posso portare ad un automa non deterministico cioè nella cui tabella di transizioni compaiono più stati futuri per una stessa configurazione.

Riconoscitore non deterministico → è meno efficiente di uno deterministico, e deve disporre di strutture dati interne per ricordare la strada fatta e poterla disfare se necessario per esplorarne un'altra.

In un linguaggio imperativo bisogna costruirsi tutto a mano, mentre Prolog ha già questa capacità innata.

Da automa non deterministico a deterministico → un automa non deterministico può sempre essere ricondotto a un automa deterministico equivalente.

1. Si definisce un automa i cui stati corrispondono a dei set di stati dell'automa originale (non deterministico)
2. Si costruisce la tabella delle transizioni del nuovo automa aggiungendo righe via via che si analizzano nuovi casi.

Espressioni e linguaggi regolari → l'insieme dei linguaggi riconosciuti da un ASF coincide con l'insieme dei linguaggi regolari, ossia quelli descritti da espressioni regolari. Espressioni regolari e automi a stati finiti sono metodi descrittivi appartenenti a due differenti categorie.

- ASF sono un metodo di descrizione operazionale perché evidenziano i passi computazionali da compiere per riconoscere le frasi
- Le espressioni regolari sono invece un metodo di descrizione denotazionale

Dalle espressioni regolari agli automi → ogni * diventa un ciclo (ε-rule o un auto-anello nel caso più semplice. Mentre ogni + diventa un percorso alternativo.

Grammatica, Espr. Regolari e Automa → si può passare da uno qualunque dei tre formalismi ad un altro poiché sono tre punti di vista della stessa realtà. Però potrebbe essere più complicato passare dalle Espr. Regolai alla Grammatica e quindi conviene passare dalle Espr. Regolari all'Automa e poi alla Grammatica.

04. RICONOSCITORI PER GRAMMATICHE CONTEXT FREE (TIPO 2)

Un RSF non può riconoscere un linguaggio di tipo 2, poiché ha un limite intrinseco strutturale e prefissato, alla propria capacità di memorizzazione. Ergo non riece a riconoscere frasi che richiedano per la loro verifica di memorizzare una parte di lunghezza non nota a priori,

> Esempio: il bilanciamento delle parentesi. In questo caso non sapendo quante volte vado ad aprire le parentesi non so neppure quante volte devo andarle a chiudere

Push Down Automaton → per riconoscere un linguaggio di tipo 2 serve un nuovo tipo di automa, che super il limite di memoria finita del RSF. Tale automa quindi dovrà appoggiarsi a una struttura dati esterna ad esso, espandibile quanto serve, cioè uno stack. PDA = RSF + STACK.

PDA <A,S,S0,snf,Z,Z0>

- A: alfabeto
- S: insieme degli stati
- S0: stato iniziale
- sfn: AxSxZ -> Q
- Z: alfabeto dei simboli interni (simboli usati sullo stack)
- Z0: simbolo iniziale sullo stack

Linguaggio accettato da un PDA → può essere definito in due modi equivalenti:

- **criterio della stato finale**: il linguaggio accettato è l'insieme di tutte le stringhe di ingresso che portano il PDA in uno stato finale (come RSF).
- **Criterio dello stack vuoto**: il linguaggio accettato è definito come l'insieme di tutte le stringhe di ingresso che portano il PDA nella configurazione di stack vuoto.

Funzione sfn → dati un simbolo di ingresso (A), lo stato attuale (S) e il simbolo interno attualmente al top dello stack (Z) può operare 3 operazioni:

- Effettua una POP dallo stack (preleva il simbolo al top dello stack)
- Porta l'automa nello stato futuro
- Effettua una PUSH sullo stack di zero o più simboli interni

Fasi si un PDA → il funzionamento di un PDA si divide in due fasi: FASE1 (memorizzare) cioè una fase di accumulo in cui memorizzo. Quando arriva all'elemento centrale termina la FASE1 e poi incomincia la FASE2 (verifica) che è una fase di controllo in cui verifico se ci sono tante chiusure quante aperture.

PDA non deterministici → anche un PDA può essere non deterministico, in tal caso la funzione sfn produce insiemi di elementi di Q. Il PDA quindi con lo stesso ingresso, stesso stato e stesso stack può finire in stati futuri differenti.

> Il non determinismo può emergere sotto due aspetti: l'automa in un certo stato Q0 con simbolo interno in cima allo stack z e con ingresso x può portarsi in uno qualunque degli stati futuri disponibili, oppure l'automa in un certo stato Qi con simbolo interno in cima allo stack z e con ingresso x può leggere o non leggere il simbolo di ingresso. Ciò accade se sono definite entrambe le mosse sfn(Qi,x,Z) e sfn(Qi,ε,Z) di cui la seconda è una ε-mossa.

Teorema1 → la classe dei linguaggi riconosciuti da un PDA non deterministico coincide con la classe dei linguaggi context-free (tipo 2). Qualunque linguaggio context free può sempre essere riconosciuto da un opportuno PDA.

Teorema2 → esistono linguaggi context-free riconoscibili soltanto da PDA non deterministici

PDA deterministico → per ottenerlo bisogna evitare che il PDA si trovi in una delle due condizioni considerate in precedenza.

Passando da PDA non deterministico a deterministico però vengono meno alcune proprietà:

- Il criterio dello stack vuoto p meno potente del criterio degli stati finali
- Una limitazione sul numero degli stati interni o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- L'assenza di ε-mosse riduce l0insieme ei linguaggi riconoscibili

Mix tra linguaggi deterministici e regolari → Se L è un linguaggio deterministico e R uno regolare, il linguaggio quoziente L/R (ossia l'insieme delle stringhe di L private di un suffisso regolare) è deterministico, come anche il concatenamento L.R (ossia l'insieme delle stringhe di L con un suffisso regolare) è deterministico.

Realizzazione PDA deterministico → la differenza tra RSF e PDA è la presenza o meno dello stack. Quindi invece di creare uno stack, potrei utilizzare lo stack del sistema operativo, che è fatto in assembler ed è perfettamente funzionate (svolge operazioni PUSH quando chiama una funzione e POP quando la funzione finisce)

Analisi ricorsiva discendente → si introduce una funzione per ogni metasimbolo della grammatica e la si chiama ogni volta che si incontra quel metasimbolo. Ogni funzione copre le regole di quel metasimbolo, ossia riconosce il sotto-linguaggio corrispondente.

Separare motore e grammatica → 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 svolgerò le singole azioni consultando di volta in volta la tabella di parsing.

Grammatiche LL(k) → le grammatiche che sono analizzabili in modo deterministico, procedendo left to right, applicando la left-most derivation e guardando avanti al più di k simboli. Nelle grammatiche LL(1) basta guardare in avanti di un solo simbolo per poter operare in modo deterministico.

Starter set → sono i simboli iniziali di un dato meta-simbolo (o di una specifica sua riscrittura) ricavati anche da più produzioni se non sono immediatamente evidenti al primo livello

Starter ste del non terminale A l'insieme:
$SS(A) = \{ a \in VT \mid A \text{ +-> } \alpha\beta \}$, con $\beta \in V^*$

Starter set della riscrittura α l'insieme:
$SS(\alpha) = \{ a \in VT \mid \alpha \text{ *-> } a\beta \}$, con $\alpha \in V+$ e $\beta \in V^*$

Condizione necessaria LL(1) → perché una grammatica sia LL(1), gli starter set dei metasimboli con cui iniziano le parti destre di produzioni alternative siano disgiunti (non devono avere gli stessi simboli inziali).

Problema della stringa vuota → se in una regola è presente come primo simbolo ε allora può far scomparire dei pezzi delle nostre regole, il che non permette in alcuni casi di capire subito qual sia l'iniziale di una espressione.

> Soluzione: o si elimina la stringa vuota, agendo per sostituzione, oppure si amplia in un qualche modo la nozione di starter set.

Director symbol set → definiamo director symbol set della produzione A->α l'unione di due insieme, lo starter symbol set e il nuovo following symbol set.

> DS(A->α) = SS(α) unito FOLLOW(A) se α*->ε
>
> Dove FOLLOW(A) denota l'insieme dei simboli che nel caso A generi ε possono seguire la frase generata da A.
>
> Se A non genera mai ε allora il director symbol set non è altro che il semplice starter set.

Condizione necessaria e sufficiente LL(1) → perché una grammatica sia LL(1), è sufficiente che i director symbol set relativi a produzioni alternative siano disgiunti.

05. DAI RICONOSCITORI AGLI INTERPRETI

Riconoscitore puro → accetta in ingresso una stringa di caratteri e riconosce se essa appartiene al linguaggio. Risponde solo si o no

Interprete → è più di un riconoscitore puro, poiché non solo riconosce se una stringa appartiene al linguaggio, ma esegue azioni in base al significato della frase.

> Un interprete è di solito strutturato come uno **Scanner** e un **Parser.** Lo scanner analizza le parti regolari del linguaggio fornendo al parser singole parole già aggregate. Il parser riceve dallo scanner i singoli token per valutare la correttezza della loro sequenza

Analisi lessicale → viene fatta raggruppando i singoli caratteri dell'input secondo le produzioni regolari associate alle diverse possibili categoria lessicali (ad esempio id, numero, etc.) L'analizzatore può categorizzare i token mentre li analizza osservando lo stato finale del suo RSF.

Tuttavia cablare ogni dettaglio nella struttura del RSF non è una strategia vincente, poiché un linguaggio contiene parole chiave o simboli speciali e per tenerne conto dovrebbe avere delle regole specifiche.

Per passare da un puro riconoscitore a un interprete occorre propagare qualcosa di più di un sì o no, come ad esempio un VALORE se l'obiettivo è la valutazione immediata, o un ALBERO se l'obiettivo è la valutazione differita (compilatore o interprete a più fasi).

Dalla grammatica al parser

Dal parser al valutatore

Abstract Syntax Tree (AST) → ogni operatore è un nodo con due figli (albero binario), il figlio di sinistra è il primo operando, il figlio di destra è il secondo operando, mentre i valori numerici sono le foglie dell'albero.

> La struttura dell'albero fornisce anche intrinsecamente l'ordine corretto di valutazione dell'espressione, senza ambiguità

Valutazione degli alberi → la teoria degli alberi introduce il concetto di visita, che può essere di 3 tipi: Pre-order, Post-order e In-order.

- Pre-order: radice, figli (da sinistra a destra) [notazione PREFISSA]
- Post-order: figli (da sinistra a destra), radice [notazione POSTFISSA]
- In-order: figlio sinistro, radice, figlio destro [notazione INFISSA]

> La notazioni POSTFISSA è adatta a un elaboratore poiché fornisce prima gli operandi che possono essere caricati nei registri solo dopo comanda l'esecuzione dell'operazione. Lo stesso principio è usato anche nei compilatori.

06. STILI DI INTERPRETAZIONE

Il valutatore valuta un AST in un dato dominio secondo la sua funzione di interpretazione.

Il valutatore quindi incorpora la funzione di valutazione. Deve visitare l'albero applicando in ogni nodo la semantica prevista per quel tipo di nodo, inoltre deve discriminare che tipo di nodo sta visitando.

Implementazione funzione di valutazione

- Con una funzione statica (eval) che mangia un albero e restituisce un risultato, questo caso un intero.

  Per girare sull'albero è altamente probabile che ci siano ricorsioni o in ogni operazione della ricorsione bisogna che si analizzi in quale tipo di nodo ci troviamo e in base a ciò svolgere un'operazione diversa. Per discriminare il tipo di nodo utilizza delle catene if ... else

- Con un metodo eval, che potrebbe essere del PARSER o dell'AST (in questo caso cambia totalmente l'approccio)

  Se il metodo eval è del PARSER, ogni volta che devi aggiungere una valutazione devi andare a modificare il parser e rifare il build di tutto. Sennò posso creare una classe generica che possiede il metodo eval e per ogni tipo di nodo il metodo viene implementato in maniera differente.

Ci sono due tipi di approccio: approccio funzionale, dove è presente una sola funzione che esegue tanti if, uno per ogni nodo, il secondo approccio è molto più object oriented e sfrutta il polimorfismo per spezzare la funzione (che è sempre una sola) ma è divisa tra i vari nodi e associa ad ogni nodo le sue responsabilità.

Metodologia funzionale → funzione di valutazione "all in one" esterna all'AST

Separa nettamente la sintassi e l'interpretazione. Si base su una funzione di interpretazione esterna alle classi che determina la classe dell'oggetto, opera un cast sicuro e accede al contenuto informativo del nodo per calcolarne il valore

PRO: facilita l'inserimento di nuove interpretazioni, perché basta scrivere una nuova funzione.

CONTRO: rende più oneroso introdurre una nuova produzione, poiché devo modificare il codice di ogni funzione.

Metodologia object oriented → funzione di valutazione come metodo dell'AST

Sfrutta il polimorfismo dei linguaggi a oggetti. Si basa su un metodo di interpretazione (interno alle classi) che è specializzato da ogni classe, e grazie al polimorfismo invocando il metodo sull'oggetto viene eseguita la versione appropriata.

PRO: facilita l'aggiunta di nuove produzioni, poiché basta definire una sottoclasse con il relativo metodo di valutazione

CONTRO: rende più oneroso introdurre nuove interpretazioni, perché bisogna definire il nuovo metodo in ogni classe.

Arithmetic Stack Machine → è una macchina stratta di nostra invenzione senza registri, che opera solo sullo stack ed ha 4 operazioni aritmetiche: SUM, SUB, MUL e DIV, più due operazioni di trasferimento da stack: PUSH e PUB.

Per definizione le operazioni aritmetiche prelevano dallo stack i due operandi (con due POP) e pongono sullo stack il risultato (con una PUSH)

Visitor come interprete → per incapsulare la logica di interpretazione serve un opportuno oggetto che sostituisca il costrutto funzionale come "contenitore" capace di assicurare l'unitarietà concettuale e fisica della funzione interpretazione. È esattamente un Visitor.

> Il Visitor realizza la logica di interpretazione in modo coerente all'approccio a oggetti. Si scrive un Visitor per ogni interpretazione richiesta, di fatto la logica è concentrata tutta in un unico luogo, ma non è più una funzione. Ora all'interno del Visitor ho tante implementazioni del metodo visit quante sono le classi della tassonomia.

Double dispatch → si attiva la valutazione chiedendo all'espressione di accettare la visita di un certo visitor, ad esempio expression.accept(visitor). L'espressione serve la richiesta rimpallando l'azione sul visitor e passando se stessa come oggetto da visitare

Visitor per le espressioni → deve prevedere tante versioni specializzate di visit quanti i tipi di espressioni da visitare.

08. INTERPRETE ESTESO: ASSEGNAMENTI, AMBIENTI E SEQUENZE

Tutti i linguaggi di programmazione introducono un qualche livello di nozioni di variabile e assegnamento. 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** perché il suo significato dipende dalla posizione del simbolo rispetto all'operatore di assegnamento.

L-value e R-value → è per questo motivo che si distingue tra L-value e R-value, che rispettivamente sono:

- L-value: il nome della variabile indica la variabile in quanto tal (tipicamente la cella di memoria associata)
- R-value: il nome di variabile indica il contenuto della variabile (ossia il valore associato a quel simbolo)

L'assegnamento è intrinsecamente LL(2) perché per capire cos'è x devo prima guardare avanti di un passo. In bash invece ad esempio devo poter distinguere se parlo di L-value o di R-value fin da subito.

Assegnamento distruttivo → si può cambiare il valore associato in precedenza a un simbolo di variabile. (linguaggi imperativi)

Assegnamento non distruttivo → non si può cambiare il valore associato in precedenza a un simboli di variabile e nel caso restituisce errore. (linguaggi logici)

Environment → per esprimere la semantica dell'assegnamento occorre introdurre il concetto di environment inteso come insieme di coppie (simbolo, valore)

Assegnamento → l'assegnamento modifica l'environment causando un effetto collaterale secondo la seguente semantica: x = valore

- se non è già presente una coppia con primo elemento x si inserisce nell'environment la coppia (c, valore)
- se invece esiste già una coppia (x, v) ci 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 da luogo a un errore.

Environment multipli → nei linguaggi imperativi l'assegnamento è distruttivo e produce effetti collaterali nell'environment. L'environment p spesso suddivisio 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

Assegnamento multiplo → l'assegnamento è un'istruzione che vuole produrre un effetto collaterale nell'environment. Tuttavia ciò rende impossibile comporre un assegnamento multiplo poiché bisognerebbe interpretare l'assegnamento come espressione.

L’assegnamento è necessariamente associativo a destra, perché il valore è l’ultimo elemento. La frase x=y=z=valore deve quindi essere interpretata come x=(y=(z=valore)))

14. RICONOSCITORI LR(0)

La debolezza dell'analisi LL sta nella sua semplicità, poiché costruisce l'albero top-down, deve riuscire a identificare la produzione giusta avendo visto solo i primi k simboli di tale produzione

L'analisi LR invece costruisce l'albero bottom-up, cioè pospone le scelte fino a quando ne sa abbastanza, ossia finché non ha visto abbastanza input da coprire tutta la parte destra della produzione.

Ogni grammatica LL(k) è anche LR(k). Però a parità di k LR batte sempre la LL.

L'analisi LR però è molto complessa quindi sono state sviluppate delle tecniche semplificate che riducono la complessità:

- SLR Simple LR
- LALR(1) Look-Ahead LR

Tecniche LR → l'analisi LL opera top-down, quindi parte dal simbolo iniziale della grammatica e dopo cerca di costruire l'albero partendo dalla radice, coprendo la frase data. Deve però sapere da che parte andare ad esempio LL(1) guarda avanti di un simbolo.

L'analisi LR opera bottom-up, quindi parte dalla frase da riconoscere e cerca di ridurla allo scopo S. a tal fine ad ogni passo deve decidere se proseguire la lettura dall'input (SHIFT) o costruire un pezzo di albero senza leggere input (REDUCE).

Architettura parser LR → il parser LR richiede al suo interno la presenza di un oracolo, cioè un componente che in base al contesto corrente gli dica in ogni istante se proseguire nella lettura o costruire un pezzo di albero.

Un parser LR è costituito da:

- Un oracolo, che dice se fare SHIFT o REDUCE
- Uno stack in cui conservare l'input o l'albero
- Un controller che governa il tutto

Oracolo → per decidere chi abbia prodotto una sottostringa α non basta che la grammatica contenga una produzione A->α, perché α potrebbe derivare da più produzioni, quindi occorrono informazioni di contesto.

L'oracolo è un **riconoscitore di contesti.**

- Si calcolano le informazioni di contesto in ogni produzione
- L'oracolo riconosce il contesto attuale e in base a ciò risponde
- Quando riconosce un contesto di riduzione ordina la mossa di riduzione giusta

La scelta operata dall'oracolo dipende dal pezzo di albero che hai costruito, da ciò che hai sullo stack e dall'input, e queste vengono chiamate informazioni di contesto.

Analisi LR(0) → nell'analisi LL il caso più semplice è LL(1) visto che il caso LL(0) sarebbe stato troppo semplice. La maggior complessità della tecnica LR suggerisce invece di partire dal caso più semplice possibile cioè LR(0). LR(0)

significa che è possibile scegliere la mossa giusta da fare sena dover mai guardare il prossimo simbolo di input.

I passi da seguire per l'analisi LR(0) sono:

- Calcolare il contesto LR(0) di ciascuna produzione
- Verificare se ci sono collisioni tra i contesti relativi a produzioni diverse. Una **collisione** si ha quando una stringa appartenente a un contesto è un prefisso proprio di una stringa di un altro contesto. Un **prefisso proprio** invece è una stringa che è prefisso di un'altra e ciò che segue è un terminale.
- Se non ci sono collisioni si possono usare i contesti LR(0) per guidare l'analisi e la grammatica è LR(0)

Analisi LR(0): calcolo dei contesti → il calcolo dei contesti LR(0) si basa sul fatto che essi sono definiti da un'opportuna grammatica che è sempre regolare a sinistra.

Contesti LR(0) → formalmente il contesto LR(0) di una produzione A->αè così definito:

LR(0)ctx(A->α) = { γ | γ=βα, Z*->βAw->βαw, w appartiene a VT* }

il contesto LR(0) della produzione A->α è l'insieme di tutti i prefissi βα di forma di frase che usi la produzione A->α all'ultimo passo di una derivazione canonica a destra.

Tutte le stringhe del contesto LR(0) della produzione A->α hanno la forma βα e differiscono quindi solo per il prefisso β che dipende solo da A.

CONSEGUENZA: si può esprimere il contesto LR(0) come prodotto tra il contesto sinistro di A e il suffisso α.

Il contesto sinistro di A: leftctx(A) = { β|Z*->βAw, w appartiene VT*}

Da cui ottengo il contesto LR(0) della produzione A->α:

LR(0)ctx(A->α) = leftctx(A)*{α}

Calcolo dei contesti sinistri → poichè uno scopo Z per definizione non compare mai nella parte destra di alcuna produzione, leftctx(Z) = {ε}

Inoltre data una produzione B->γAδ i prefissi che possono essere davanti ad A ono quelli che potevano essere davanti a B seguiti dalla stringa γ. Quindi il primo contributo a leftctx(A) è leftctx(B)*{γ}. Nel complesso leftctx(A) si ottiene unendo tutti i contributi di tutte le produzioni in cui compaia A nella parte destra.

Automa ausiliario → questo RSF è la macchina caratteristica della grammatica iniziale. Ogni stato finale è etichettato con una produzione (quella che il parser LR deve usare per fare una REDUCE in questo contesto) e ogni volta si riparte dallo stato iniziale per sapere in ogni momento che mossa devo fare.

L'automa ausiliario svolge il ruolo dell'oracolo .

- SHIFT quando l'automa aggiunge uno stato non finale (leggo dall'input)
- REDUCE quando l'automa caratteristico raggiunge uno stato finale e quindi si applica quella produzione per effettuare una riduzione. (non leggo dall'input).

Condizione LR(0) → ogni stato di riduzione dell'automa ausiliario sia etichettato da una produzione unica e non abbia archi di uscita etichettati da terminali.

L'automa caratteristico può essere ottenuto senza calcolare né i contesti sinistri né i contesti LR(0) applicando un procedimento meccanizzabile.

- Non calcoliamo i contesti per poi dover sintetizzare l'automa
- Piuttosto, partiamo dalla regola di top-level Z->S$ e analizziamo via via le situazioni che si presentano.
    - Scriviamo ogni situazione in un diverso rettangolo
    - Quando una regola ne usa un'altra la includiamo nel rettangolo
    - Introduzione l'astrazione cursore "." Per indicare dove siamo all'interno di ogni regola.
- Studiamo le evoluzioni possibili di ogni rettangolo e costruiamo direttamente l'automa dal basso, caso per caso.

Parser LR(0) → ogni arco corrisponde a un'azione SHIFT mentre ogni stato finale corrisponde a un'azione REDUCE. Si costruisce allora una tabella di parsing che oltre allo stato futuro indica anche quale azione bisogna compiere.

15. RICONOSCITORI LR(1), SLR, LALR(1)

L'analisi LR(0) è troppo semplice per gli usi reali, il caso di maggior interesse quindi è LR(1). L'analisi LR(k) opera come l'analisi LR(0) ma guarda avanti di k simboli (lookahead symbols)

Contesti LR(k) → formalmente il contesto LR(k) di una produzione A->α è così definito:

$$LR(k)ctx(A\to\alpha) = \{\gamma|\gamma=\beta\alpha u,\ Z^*\to\beta Auw\to\beta\alpha uw,\ w \in VT^*,\ u \in VT^k\ \}$$

La stringa u appartiene all'insieme $FOLLOW_k(A)$ delle stringhe di lunghezza k che possono seguire il simbolo non terminale A

Contesto sinistro LR(k) → il contesto sinistro di un non terminale A è:

$$leftctx(A,u) = \{\beta|Z^*\to\beta Auw\to\beta\alpha uw,\ w \in VT^*,\ u \in VT^k\ \}$$

dove il contesto LR(k) della produzione A->α diventa:

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

Automa caratteristico LR(k) → anche in questo caso di procede nello stesso modo del caso LR(0). Si calcolano le espressioni regolari per i contesti LR(k) e si usano per costruire l'automa caratteristico.

Tuttavia tale approccio che era già complesso nel caso LR(0) ora è ancora più complesso

Procedimento operativo LR(k) → per adattare il procedimento opertivo in ogni stato si deve ora tenere conto anche del simbolo successivo. A fianco di ogni regola quindi si specifica anche il lookahead set, ossia i possibili simboli che rendono valida l'azione

Approssimazione parsing LR(1) → l'esperienza conferma che è praticamente impossibile costruire un parser LR() per un vero linguaggio poiché le sue dimensioni sarebbero intrattabili. Per questo si introducono delle versioni semplificate dell'analisi LR(1):

- **SLR** (Simple LR) → si rifiuta di calcolare tutto LR(1), quindi calcola LR(0) che costa sicuramente meno e poi calcola tutti i simboli che possono seguire uno stato anche se ci sono situazioni che posso creare dei conflitti
- **LALR(1)** (Look-Ahead LR) → è un LR che prescinde la look-ahead quindi voglio creare un automa con meno stati. LALR segue l'analisi LR(1) ma poiché si ottengono molti stati che sono simili tra di loro, li accorpa insieme.

Contesti SLR → il contesto SLR(k) di una produzione A->α è così definito:

$$SLR(k)ctx(A\to\alpha) = LR(0)ctx(A\to\alpha) * FOLLOWk(A)$$

È possibile dimostrare che il contest SLR è un pò più grande di quello LR completo e perciò è più esposto a potenziali conflitti.

Procedimento operativo SLR → il modo più immediato per ottenere il aprser SLR è quello di prendere l'automa LR(0) e togliere certe riduzioni cioè quelle incompatibili con il lookahead set.

SLR parte da LR(0) e poi tiene conto di quello che c'è dopo. Perciò prima facciamo LR(0) e poi creiamo la tabella di parsing come nel caso LR(0), dopodiché tutte le celle che non appartengono ai FOLLOW allora le vado a cancellare poiché rappresentano situazioni incompatibili.

Parser LALR(1) →approccio alternativo che consiste nel collassare in un solo stato tutti quegli stati che nel parser LR(1) completo sono identici a meno dei lookahead set. Può essere implementato calcolando prima LR(0) e poi aggiungendo i lookahead set.

PRO: è una trasformazione sempre possibile spesso molto conveniente perichè il parser LALR ha molti meno stati del LR

CONTRO: possono apparire conflitti reduce/reduce tipicamente gestibili

Bilancio finale → esistono delle grammatiche LR(1) che non sono SLR(k) per nessun valore di k. Inoltre esistono delle grammatiche LL(1) che non sono SLR(1).

Per quanto riguarda SLR ci sono dei risultati positivi, per cui ogni grammatica SLR(k) è anche LR(k), e ogni grammatica LL(k) priva di produzioni inutili e aumentata della produzione Z->S$ è anche LR(k).

Possibile domanda: principio ispiratore di un SLR e LR e come si può ottenere un LR(1) partendo da un SLR.

2ª PARTE: MODELLI COMPUTAZIONALI

16. ITERAZIONE VS RICORSIONE

Obiettivo di questo capitolo quello di concentrarsi sul modello computazionale indipendentemente dalla sintassi, dal linguaggio e dal costrutto linguistico usati. Cosa contraddistingue un processo computazionale iterativo rispetto a uno ricorsivo?

Processi computazionali iterativi → nei linguaggi imperativi il costrutto linguistico che esprime un processo computazionale iterativo è il ciclo. Nello schema di un ciclo c'è sempre una variabile accumulatore che viene inizializzata prima e modificata durante il ciclo, il quale quando termina contiene il risultato.

Solitamente si utilizzano dei blocchi come il ciclo while, do..while, ma a noi interessa sapere solo cosa succede dentro questi blocchi che producono il ciclo

Il processo computazionale computa IN AVANTI

Processi computazionali ricorsivi → al contrario un processo computazionale ricorsivo è espresso da una funzione ricorsiva. In questo schema non c'è un accumulatore, la chiamata ricorsiva ottiene il risultato (n-1)esimo; inoltre il corpo della funzione opera sul risultato (n-1)esimo per sintetizzare il risultato n-esimo desiderato.

La ricorsione risulta essere più comoda del ciclo poiché in alcuni casi può fare cose che non sono possibili con il ciclo. Ma la ricorsione viene utilizzata quando ogni volta si possono aprire diverse possibilità (cosa che non capita con il ciclo in cui le operazioni non hanno alternative). Nella ricorsione visto che non c'è l'accumulatore, quindi ho qualcosa in mano solo al termine della ricorsione.

Il risultato è sintetizzato mentre le chiamate si chiudono, il processo computa quindi ALL'INDIETRO.

Tail recursion → un processo computazionale ricorsivo è espresso da un una funzione ricorsiva, ma un costrutto sintatticamente e formalmente ricorsivo può dare luogo a un processo computazionale iterativo? Ciò accade se la chiamata ricorsiva è l'ultima istruzione della funzione e il risultato parziale viene passato in avanti come argomento. Si chiama tail-recursion poiché la ricorsione avviene in coda.

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

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 effettua la nuova chiamata. Come nell'iterazione dobbiamo quindi portarci dietro sia l'accumulatore che il contatore dei cicli effettuati.

A runtime computa effettivamente in avanti come un ciclo, mentre il return serve solo a passare indietro il risultato che viene accumulato ad ogni passaggio. Nelle tail-recursion inoltre non c'è bisogno di avere un nuovo record di attivazione ad ogni passo poiché posso riutilizzare sempre lo stesso visto che di fatto sto effettuando un assegnamento distruttivo.

Processi computazionali iterativi espressi da costrutti ricorsivi → la ricorsione tail quindi un costrutto ricorsivo che dà luogo a un processo computazionale iterativo. Per questo può essere usata in alternativa ai cicli per esprimere l'iterazione.

- Nei linguaggi funzionali e logici si tende a seguire questo approccio ottimizzando opportunamente a runtime
- Nei linguaggi imperativi invece avendo i cicli di norma non viene ottimizzata la ricorsione tail.

Il ciclo quindi risulta essere superfluo poiché con la ricorsione posso sia effettuare operazioni ricorsive che effettuare operazioni iterative tramite la ricorsione tail. I primi linguaggi avevano i GO TO q quindi è stato naturale introdurre i cicli nei linguaggi di programmazione.

Ottimizzazione della tail recursion (TRO) → può essere ottimizzata stabilendo di allocare il nuovo record di attivazione sopra al precedente, in tal modo l'occupazione di risorse è identico al caso del ciclo.

Due operazioni sono uguali se a runtime sono uguali (hanno la stessa immagine) e non se è scritta nello stesso modo.

- In quasi tutti i linguaggi imperativi (C, Java , C#) la tail recursion non è ottimizzata: questo perché tali linguaggi offrono una sintassi specifica per l'iterazione (while, for) e quindi la ricorsione è usata solo per processi ricorsivi.
- Nei linguaggi logici (Prolog) e funzionali (Lisp) la tail recursion invece è ottimizzata: poiché non offrono una sintassi specifica per l'iterazione quindi la tail recursion è l'unico modo per esprimesse processi computazionali iterativi.

Scala → scala rappresenta un linguaggio blended (ibrido) che unisce gli approcci imperativi e funzionali, poiché assomiglia a java ma è possibile usare un approccio scalabile completamente imperativo (come una shell) oppure completamente funzionale. In scala la tail recursion è ottimizzata.

18. PROGRAMMAZIONE FUNZIONALE

Tutti i linguaggi di programmazione supportano un qualche costrutto per esprimere le funzioni, ma di rado queste sono first-class entities. Infatti non è possibile fare su una funzione tutta una serie di operazioni che faresti con un qualunque altro tipo di dato. Una funzione che sia **first-class entities** quindi deve essere manipolabile come ogni altro tipo di dato:

- Deve poter essere assegnata a una variabile
- Deve poter essere passata come argomento a un'altra funzione
- Deve poter essere restituita da una funzione
- Deve poter essere definita e usata "al volo" come ogni altro valore
- Magari anche senza avere un nome

Funzioni nei linguaggi imperativi → nei linguaggi imperativi tradizionali, una funzione tipicamente è solo un costrutto che incapsula codice, non è manipolabile come ogni altro tipo di dato.

Funzioni nei linguaggi funzionali → nei linguaggi funzionali non è solo un costrutto che incapsula codice, ma è un tipo di dato manipolabile come ogni altro. Puoi fare con le funzioni tutto quello che faresti con una stringa o un intero.

Funzioni come first class entities → passare una funzione come argomento a un'altra significa poter definire funzioni di ordine superiore, che manipolano altre funzioni. Bisognerà poi definire per il tipo funzione un metodo () che produce una call, che esegue un comportamento.

Variabili libere → se un linguaggio ammette le definizioni di funzioni con variabili libere (cioè non definite localmente) tali funzioni possono dipendere dal contesto circostante. Nei linguaggi classici le variabili o sono dentro le funzioni o sono globali, ma chiaramente quando possiamo chiamare una funzione dentro a un'altra funzione e ciascuna di quelle utilizza le sue variabili diventa un casino.

Funzioni e chiusure → se il linguaggio supporta funzioni come first-class entities, la presenza di variabili libere nelle funzioni determina la nascita della notazione di chiusura.

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

Invocando ff non viene calcolato nulla, ma si ottiene come risultato un oggetto funzione che incorpora al proprio interno I riferimenti alle due variabili f e x della funzione ff. Quindi la funzione genitore mette dentro al figlio qualcosa di se, magari il genitore muore ma il figlio può essere utilizzato lo stesso.

Chiusure e tempo di vita delle variabili → in presenza di chiusure il tempo di vita delle variabili di una funzione non coincide necessariamente con quello della funzione che lo contiene. Il modello deve tenere conto che quelle variabili non si possono più allocare sullo stack (ma nell'heap).

- Una variabile locale che è indispensabile al funzionamento della funzione più interna deve sopravvivere
- Il tempo di vita delle variabili locali che fanno parte di una chiusura è pari a quello della chiusura

A cosa servono le chiusure? → le chiusure permettono di rappresentare e gestire uno stato privato e nascosto, di creare una comunicazione nascosta, di definire nuove strutture di controllo e di riprogettare/semplificare API e framework di largo uso.

Chiusure negli altri linguaggi → Le chiusure sono già da alcuni anni supportate in linguaggi come Javascript, Scala e Ruby, ma si stanno diffondendo anche in Java8, C# e C++ seppur in forma adattata.

- Java8 ha introdotto le Lambda expressions
- C# offre lambda expressions e metodi anonimi
- C++ offre le chiusure a partire dal recente standard 11

Chiusure in C# e Java8 → L'approccio utilizzato è quello di utilizzare le chiusure come lambda expressions.

- Notazione C#: { **params** => **body** }
- Notazione Java8: (**params**) -> {**body**}

params è una lista di parametri formali con tipo, separati da una virgola, mentre **body** è una lista di istruzioni di cui l'ultima può essere una espressione (che denota il valore della chiusura).

La semantica molto semplice che permette di creare funzioni compatte, che possono essere passate come argomento. In pratica l'esecuzione della chiusura crea un oggetto contenente il codice del body più il contenuto lessicale.

Chiusure in Scala e Javascript → offrono funzioni come first-class entities, infatti è possibile definire un generico function literal e se ci sono variabili libere, al momento dell'uso nasce una chiusura.

Criteri di chiusura: chiusura lessicale vs chiusura dinamica → bisogna stabilire un criterio sul come e dove cercare una definizione per le variabili libere da chiudere. Di seguito c'è un esempio:

```
var x = 20;
function provaEnv(z) { return z + x; }
function testEnv() {
      var x = -1;
      return provaEnv(18);
}
```

- Se si considera lessicalmente il testo del programma la definizione di x più vicina (ossia precedente l'uso) è quella in alto. (In questo caso z + x risulterebbe essere 38)
- Se invece si considera l'ordine delle chiamate le cose sono diverse perché la definizione di x più vicina è quella locale di testEnv. (In questo caso z +x risulterebbe essere 17)

Catena lessicale: più leggibile ma il comportamento è bloccato. In questo caso si dice che il modello computazionale adotta una chiusura lessicale.

Catena dinamica: meno leggibile ma il comportamento è dinamico. In questo caso di dice che il modello computazionale adotta una chiusura dinamica.

Poiché la comprensibilità è cruciale praticamente tutti i linguaggi di programmazione adottano il criterio di chiusura lessicale.

Modelli computazionali per la valutazione di funzioni → ogni linguaggio che prevede funzioni deve prevedere un modello computazionale per la loro valutazione. Tale modello deve stabilite QUANDO si valutano i parametri, COSA di passa alla funzione e COME si attiva la funzione.

Modello applicativo (chiamato Call by value) → tradizionalmente è il più usato e stabilisce:

- QUANDO si valutano i parametri: subito prima di ogni cosa
- COSA si passa alla funzione: di solito il valore die parametri
- COME si attiva la funzione: chiamandola e cedendo il controllo

Il modello applicativo è molto usato perché è semplice e nel complesso efficiente, tuttavia valutando i parametri a priori può causare inefficienze e soprattutto fallimenti non necessari. Valutando sempre i parametri infatti può fare lavoro inutile.

Modello Call by name → è un approccio pigro ma a volte vincente, carica i parametri solo se necessari e quindi solo se sono realmente utili.

- QUANDO si valutano i parametri: solo al momento dell'effettivo utilizzo
- COSA si passa alla funzione: non si passano valori o indirizzi ma bensì oggetti computazionali
- COME si attiva la funzione: chiamandola e cedendo il controllo

Nonostante i suoi vantaggi concettuali è meno efficiente nei casi normali: infatti valuta i parametri più volte per ogni chiamata e richiede più risorse a runtime dovendo gestire oggetti computazionali anziché semplici valori o indirizzi.

Simulare il modello call by name → il modello call by name può essere simulato nei linguaggi che hanno funzioni come first class entities. In questo modo vengono ricreati a mano tutti i passi che la call by name avrebbe svolto da solo come:

- Passare alla funzione non dei valori ma oggetti computazionali
- Sostituire ogni valore di parametro attuale con una funzione che ritorni tale valore
- Sostituire ogni uso di un parametro normale dentro alla funzione con una chiamata alla funzione stessa

19. JAVASCRIPT

Javascript adotta un approccio funzionale con funzioni e chiusure, inoltre adotta anche un modello object-based senza classi basato sul concetto di prototipo (prototype-based).

È un linguaggio interpretato e può essere utilizzato per creare applicazioni multi-paradigma

I motori Javascript più diffusi sono: V8, Nashorn (Java8) e Rhino (java < 8)

- Le variabili in Javascript sono loolesly typed quindi è possibile assegnare alla stssa vairabile prima un valore di un tipo e poi un valore di un altro tipo.
- La dichiarazione di una variabile può essere implicita (senza parola var) o esplicita (con parola var). Una variabile avrà scope locale in caso di dichiarazione esplicita dentro alle funzioni, mentre globale in tutti gli altri casi
- L'operatore typeof restituisce il tipo di un espressione

IL LATO FUNZIONALE

Le funzioni sono introdotte dalla keyword function. Javascript ammette anche funzioni anonime, di fatto analoghe alle lambda expression.

Le funzioni sono invocate tramite il classico operatore di chiamata (), fornendo i parametri attuali. A differenza di java il numero dei parametri attuali può non corrispondere con il numero dei parametri formali (quelli in più vengono ignorati, quelli in meno sono undefinded).

In javascript una funzione è una first-class entities, pienamente manipolabile come ogni altro tipo di dato:

- Può essere assegnata a variabili
- Può essere definita e usata al volo
- Può essere passata come argomento a un'altra funzione
- Può essere restituita da una funzione generatrice

Function expression → assegna una funzione a una variabile

```
var f = function g(x){ return x/10; }
g(32) //ERRORE: il nome g è indefinito
```

Function declaration → introduce una funzione senza assegnarla a una variabile

```
function g(x){ return x/10; }
g(32); //IL NOME G È NOTO NELL'AMBIENTE
```

Funzioni innestate e chiusure → si può definire una funzione dentro l'altra, in questo modo nascono delle chiusure. In particolare nasce una chiusura quando una funzione innestata fa riferimento a una variabile della funzione esterna.

Chiusure in javascript

- **Rappresentare uno stato privato e nascosto**: in Javascript tutte le proprietà degli oggetti sono pubbliche, se vuoi ottenere una proprietà privata la devi inserire in una chiusura. Questo stato è inaccessibile dall'esterno poiché è interno alla funzione e l'unica che aveva in mano quell'x era la funzione che ora è morta.
- **Realizzare un canale di comunicazione privato**: si ottiene ciò mettendo dentro alla chiusura sia lo stato che i metodi accessor.
- **Realizzare nuove strutture di controllo**: una funzione di secondo livello incapsula il controllo mentre argomenti-funzione rappresentano le azioni da svolgere.

Chiusure e binding delle variabili → per catturare il valore di una variabile che cambia occorre fotografarlo in una variabile ausiliaria, ad esempio tramite una funzione intermedia che svolge il ruolo di tampone.

```
function fillFunctionsArray(myarray){
      for (var i=4; i<7; i++)
      myarray[i-4] = function(){ return i;};
}

function fillFunctionsArray(myarray){
      function aux(j){ return function(){ return j;}}
      for (var i=4; i<7; i++)
      myarray[i-4] = aux(i);
}
```

IL LATO AD OGGETTI

Javascript adotta un modello object-based senza classi basato sulla nozione di prototipo.

Un oggetto quindi non è un'istanza di una classe ma una collezione di proprietà (dati o funzioni) pubbliche accessibili tramite dot notation che possono variare in maniera dinamica. Un oggetto è costruito tramite una new da un costruttore che ne specifica la struttura iniziale. È associato dal costruttore ad un oggetto padre detto prototipo di cui eredita le proprietà.

La classe blocca la struttura degli oggetti mentre qui il costruttore di Javascript da solo la struttura generale.

Costruttore → è una normale funzione che però specifica le proprietà e i metodi dell'oggetto da creare mediante la parola chiave **this**. Viene invocata indirettamente tramite l'operatore **new**.

Il costruttore si chiama come la classe ma è l'unica cosa che rimane della classe (poiché non esistono più).

Object literals → è anche possibile specificare oggetti nella forma di object literal, elencandone le proprietà mediante una sequenza di coppie nome:valore.

```
p3 = { x:10, y:15 };
```

Accesso alle proprietà → poiché tutte le proprietà sono pubbliche, sono anche liberamente modificabili.

Costrutto with → può essere usato per accedere a più proprietà di uno stesso oggetto in modo rapido.

```
with (p1) x=22, y=3 //EQUIVALE A p1.x=22, p1.y=3
```

Aggiunta e rimozione di proprietà → le proprietà specificate nel costruttore non sono le uniche che un oggetto può avere, infatti è possibile aggiungere dinamicamente nuove proprietà semplicemente usandole. È anche possibile rimuovere dinamicamente proprietà con l'operatore **delete**.

Proprietà constructor → ogni oggetto tiene traccia del costruttore che lo ha creato mediante la proprietà constructor.

Il costruttore è l'unica cosa che è rimasta delle classi e quindi se voglio mettere da qualche parte delle proprietà che devono essere comuni a tutti gli oggetti di un tipo, le devo mettere nel costruttore. In questo modo se aggiungo al costruttore delle proprietà queste saranno comuni a tutti gli oggetti costruiti da quel costruttore da lì in avanti.

Le proprietà di classe possono essere modellate come proprietà della funzione costruttore

Proprietà e motodi privati → le proprietà in generale sono pubbliche, ma possono essere modellate tramite chiusure con apposite funzioni accessor.

```
Point = function(i,j){
      this.getX = function(){ return i; }
      this.getY = function(){ return j; }
}
```

In java quando muore un costruttore muoiono anche le variabili I e j ma essendo dentro delle chiusure in Javascript non muoiono poiché servono a getX() e getY() e visto che non sono altrimenti raggiungibili sono private.

Prototipi di oggetti → ogni oggetto è associato a un oggetto padre, detto prototipo di cui eredita le proprietà. Ogni oggetto referenzia il suo prototipo tramite una proprietà (nascosta) chiamata __proto__. Dentro __proto__ mette le proprietà dell'oggetto padre.

Prototipi di oggetti → chi sia l'antenato padre di un oggetto dipende da chi costruisce tale oggetto. Esiste però un antenato comune a tutti, le cui proprietà sono ereditate da tutti: lo chiameremo The One. Esso è antenato diretto di ogni object literal e antenato indiretto di ogni altro oggetto.

I prototipi sono il mezzo offerto da Javascript per esprimere relazioni tra oggetti. Esiste un prototipo predefinito per ogni categoria di oggetti (array, funzioni, numeri, etc)

La tassonomia degli oggetti in java dipende dall'ereditarietà, in Javascript invece dipende da cullo che c'è scritto dentro alla proprietà del prototipo.

- typeof → per scoprire il tipo di un oggetto
- isPrototypeOf → per risalire la catena dei prototipi

Il padre di tutte le funzioni è la funzione **function Empty(){}**.

Prototipo di costruzione → il prototipo di un oggetto dipende da chi lo costruisce, quindi gli oggetti costruiti da uno stesso costruttore condividono lo stesso prototipo.

Il prototipo che un costruttore attribuisce agli oggetti da lui creati è espresso dalla proprietà **prototype**, che è una proprietà pubblica dei soli costruttori, e come tale può essere modificata.

Posso quindi intervenire a runtime per cambiare il processo dic reazione di un oggetto, ma posso anche cambiare il prototipo di un oggetto già esistente. Ad esempio posso cambiare il prototipo del prototipo dell'oggetto in questo modo l'oggetto che avevo creato avrà delle nuove proprietà derivate dal proprio pare.

Object è il costruttore degli oggetti, ossia una funzione e Object.prototype è The One.

- Le funzioni sono costruite dal costruttore **Function** che usa come prototipo **The Function.** (Function.prototype è The Function)
- Gli array sono costruiti dal costruttore **Array** che usa come prototipo **The Array**. (Array.prototype è The Array)
- Gli object literal sono costruiti dal costruttore **Object** che usa come prototipo l'ggetto base **The One**

Prototipi ed effetti a runtime → come detto in precedenza è possibile effettuare modifiche a runtime sulle catene di prototipazione.

- Se vuoi effettuare un cambiamento su un prototipo in modo che sia retroattivo, allora devi modificare il prototipo che avevi creato in precedenza.
- Se invece non vuoi che i cambiamenti siano retroattivi allora devi creare un nuovo prototipo (in modo che il vecchio continui ad esistere) e poi devi associare il nuovo prototipo al costruttore.

Type argumenting → si intende la possibilità di aggiungere proprietà al prototipo esistente. Il che produce un effetto immediato sugli oggetti esistenti e la struttura del sistema cambia a runtime.

Per farlo non è opportuno accedere al prototipo direttamente tramite la proprietà __proto__ ma è più corretto sfruttare la proprietà prototype del costruttore.

Creating and inheriting → In alternativa si può sostituire il prototipo di costruzione usato da un costruttore con un altro. Ciò non ha impatto sugli oggetti già esistenti ma permette di cambiare l'effetto di costruzione da qui in avanti. È il modo per definire una nostra ereditarietà prototipale.

> *Esprimere l'idea che la categoria Studente eredita dalla categoria Persona. Prima di tutto dobbiamo definire il costruttore Persona e il costruttore Studente in accordo con le proprietà di ogni persona e di ogni studente.*
>
> *Per esprimere il fatto che Studente is-a Persona gli oggetti studente che saranno costruiti dovranno avere come prototipo una persona e non The Student.*

1. *Occorre scegliere un oggetto persona che faccia da prototipo per tutti gli studenti*

   protoStudente ? new Persona("zio", 1900);

2. *Impostare su di esso Studente.prototype in modo che tutti gli studenti facciano riferimento ad esso*
   Studente.prototype = protoStudente;

3. *Reimpostare la proprietà constructor dell'oggetto persona prototipo in modo che punti al costruttore Studente.*
   Studente.prototype.constuctor = Studente;

Riferimenti alla classe base → non avendo l'idea di classe Javascrpit non ha modo diretto per potersi riferire alla classe base. Tuttavia il metodo **call** permette di chiamare un oggetto-funzione svolgendo circa lo stesso compito.

```
funzione.call(target, ... , argomenti);
```

Oggetto globale → 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, più tutte le funzioni e le variabili globali definite dall'utente.

Funzione eval → valuta la stringa ricevuta come programma Javascript, in questo modo una stringa di testo diventa codice e da luogo a comportamento.

DOVE I DUE LATI SI INCONTRANO (LATO FUNZIONALE E LATO AD OGGETTI)

Come sappiamo ogni funzione in Javascript è un oggetto.

- Finora abbiamo utilizzato sempre dei function literal introdotti dalla parola chiave **function**. Questo costrutto accetta una lista di argomenti fra parentesi tonde e un corpo racchiuso tra parentesi graffe.

  *square = function(x){ return x*x; }*

- In realtà è anche possibile costruire dinamicamente oggetti-funzione tramite i costruttore **Function**. Rappresenta un costrutto linguistico di meta livello, poiché è un oggetto funzione destinato a creare altri oggetti funzione. I suoi argomenti sono tutte stringhe (i primi N-1 argomenti sono i parametri e l'ultimo è il corpo della funzione).

  *square = new Function("x", "return x*x")*

Il costruttore Function permette di sintetizzare dinamicamente la funzione desiderata, creando al volo un pezzo di comportamento eseguibile. Si rompe la barriera codice/dati.

Metodi invocabili su un oggetto funzione:

- **toString**: una stringa fissa
- **valueOf**: la funzione stessa
- **call** e **apply**: supportano i pattern di chiamata indiretta.

Pattern di chiamata indiretta → una funzione si invoca solitamente per nome in modo diretto tramite l'operatore di funzione (). Esistono però anche forme di chiamata indiretta.

methodname.call(object, argList)

methodname.apply(object, argArray)

le due primitive si differenziano solo per il modo con cui passano gli argomenti args (una lista o un array). L'oggetto target della funzione chiamata è object il primo argomento.

Array Javascript → un array Javascript non è altro che una lista numerata costruita sulla base del costruttore Array o con [..]. DI fatto gli array costituiscono il supporto per gli oggetti, infatti ogni oggetto sotto è organizzato come un array (una lista di proprietà).

Infatti la notazione array-like [..] è utilizzabile anche per accedere per nome alle proprietà di un oggetto.

Introspezione → la possibilità di aggiungere/togliere proprietà a un oggetto pone il problema di scoprire quali proprietà esso abbia in un dato istante

Intercessione → per accedere alle proprietà per nome serve la notazione array-like.

APPLICAZIONI MULTI-PARADIGMA

Un interprete Javascript interoperabile con Java con supporto alla programmazione multi-linguaggio e multi-paradigma. Questo componente è già inserito all'interno del JDK quindi posso benissimo costruire fin da subito applicazioni Java che sfruttano Javascript e viceversa.

Rhino: fino a Java 7

Nashorn: a partire da Java8

Per eseguire uno script Rhino in Java occorrono: un contesto per mantenere le informazioni thread-specific (oggetto Context) e uno scope per mantenere le informazioni script-specific (oggetto Scope).

Devo quindi prima creare il contesto e lo scope in modo tale da crear le cose minime per garantire interoperatività.

20.INTRODUZIONE AL LAMBDA CALCOLO

L'idea è quella di sviluppare il minimo formalismo capace di descrivere qualunque algoritmo (Turing-equivalente).

Un solo concetto: funzione

Un solo paradigma: applicazione di funzione

SINTASSI	$L ::= \lambda x.L \mid x \mid LL$
SEMATICA	$(\lambda x.L_b)L_a \rightarrow L_b[L_a/x]$

L può esprimere una qualsiasi struttura dati o un qualsiasi algoritmo, e rappresenta una trasformazione atomica.

Idea semplice: sostituzione del testo. Il risultato è il termine L_b con al suo interno tutte le occorrenze di x sostituite da L_a

Un **lambda termine** può essere:

- λx.L → una **funzione** senza nome (chiamata chiusura), con un parametro x e un corpo L che è a sua volta un lambda termine
- x → una **variabile**
- LM → l'**applicazione di una funzione**. L è una funzione e M è il parametro attuale che viene applicato alla funzione (ossia che prende il posto della variabile sostituita)

Lambda calcolo e Javascript → In Javascript le funzioni sono first-class objects, sono presenti le chiusure e le funzioni anonime (come nella notazione lambda)

Lambda calcolo vs C# e Java8 → In C# e Java8 per indicare le lambda expressions viene utilizzata la => e -> simboli che in analisi vengono utilizzati per le funzioni.

Definizione di metodi:

- λx.x si può leggere come

  Object m1(Object x){ return x; }

- 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 metodo che contiene il codice corrispondente a λx.x

Ciò che la funzione lambda prende si trova dopo la lambda (λ) mentre ciò che la funzione restituisce si trova dopo il punto.

Per disambiguare parentesi e/o associatività, l'applicazione di funzioni è associativa a sinistra e i corpi delle funzioni si estendono il più possibile.

yLxM si legge come ((yL)x)M)

λx.λy.xyx si legge come λx.(λy.xyx) che diventa λx.(λy.(xy)x)

Definizione della semantica → la semantica di un lambda termine è definita tramite regole di trasformazione: (λx.L)M -> L[M/x]

I passi computazionali sono i seguenti:

- **identificazione di un pattern** del tipo (λx.L)M, cioè l'applicazione della funzione λx.x al parametro M
- **sua trasformazione** nel lambda termine L[M/x] sostituendo nel corpo della funzione L ogni occorrenza di x con M

applicare una funzione a un argomento significa valutarne il corpo dopo aver sostituito al parametro formale il parametro attuale.

Funzione con più argomenti → una funzione con più argomenti può essere rappresentata come una funzione di funzione. Infatti una funzione con due argomenti non è altro che una funzione con un argomento che poi esegue un "search and replace" con il secondo argomento della funzione

λx.λy.xy si legge come λx.(λy.xy) ovvero una funzione in x il cui corpo è una funzione in y

Funzioni notevoli:

- I::= λx.x sostituzione
- K::= λx.λy.x selezione di un elemento in una struttura dati
- S::= λn.λz.λs.s(nzs) somma tra numeri
- ω::= λx.xx duplicazione dell'argomento.
- Ω::= ωω =(λx.xx)(λx.xx) loop infinito

Forma normale → una lambda termine è in forma nomale quando non si possono più applicare altre riduzioni

- **Fortemente normalizzabile**: se qualunque sequenza porta a una forma normale. Rappresenta una computazione che termina.
- **Debolmente normalizzabile**: se esiste almeno una sequenza di riduzioni che porta a una forma normale. (in questo caso è fondamentale la strategia di esplorazione utilizzata, perché potrebbe portare alla soluzione o meno). Rappresenta una computazione che termina solo se si sceglie il giusto ordine di valutazione.
- **Non normalizzabile**: se non si arriva mai a una forma normale. Rappresenta computazioni che non terminano in alcun caso

Teorema di Church-Rosser → ogni lambda ha al più una forma normale (che essendo unico può essere chiamato risultato)

Computare con le lambda → nel lambda calcolo un primo termine L descrive l'algoritmo, un altro lambda termine D descrive i dati in input e si esprime la computazione applicando L a D (cioè LD) e riducendo il più possibile. Il risultato è il nuovo lambda termine R.

Significato vs rappresentazione → anche cose semplici come i booleani sono rappresentati da sintassi strane come λx.λy.x e λx.λy.y. La stranezza del lambda calcolo è che anche le costanti sono termini funzione (non riducibili)

Strategie di riduzione → l'idea di ridurre il più possibile introduce il concetto di strategia di risoluzione, infatti adottare una strategia piuttosto che un'altra può fare la differenza.

Due principi ispiratori alternativi:

- **Eager evaluation**: privilegiare la valutazione dell'argomento rispetto all'applicazione dell'argomento alla funzione (valuta l'argomento prima possibile)
- **Lazy evaluation**: privilegiare l'applicazione dell'argomento alla funzione rispetto alla valutazione dell'argomento (valuta l'argomento solo quando serve).

Le strategie quindi si dividono in **strategie Eager** e **strategie Lazy**.

Strategie Eager:

- **Applicative order** → prima valuta gli argomenti e poi li applica alle funzioni. Tenta anche di ridurre il più possibile i termini dentro la funzione prima di applicare gli argomenti.
- **Call by value** → simile alla precedente ma non riduce i termini dentro la funzione prima di applicare gli argomenti.
- **Call by reference** → ulteriore variante del modello applicativo, la funzione riceve un riferimento implicito all'argomento del chiamante anziché la copia dello stesso
- **Call by copy-restore** → caso particolare della call by reference che viene usata nei modelli multi thread dov ogni funzione riceve una copia privata dell'argomento e poi alla fine lo ricopia indietro.

Strategie Lazy:

- **Normal order** → applica gli argomenti alle funzioni prima di ridurli, riducendo per prima la lambda più a sinistra e considerando come corpo ciò che compare a destra. Se esiste una forma normale la trova.
- **Call by name** → funziona come la normal order ma considera irriducibili le lamda abstraction (λx.x).
- **Call by need** →è una versione efficiente della call by name che non rivaluta gli stessi termini più volte, memorizzandone il risultato.
- **Call by macro** → simile alla call by name, si basa su sostituzioni testuali (viene utilizzata dal C proprio per le macro).

Il problema della ricorsione → per esprimere la ricorsione occorrono due ingredienti: la capacità di definire funzioni e la capacità di richiamare funzioni per nome (ma le funzioni lambda sono anonime!)

A questo provvede il combinatore di punto fisso Y, che non è altro che il copiatore del contenuto della funzione.

$$Y = \lambda f(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Applicandolo a una funzione F l'effetto è YF -> F(YF), ossia si ottengono esattamente i due effetti desiderati: eseguire il corpo della funzione (F()) e rigenerare in qualche modo la funzione stessa (YF)

Perché questa cosa funzioni bisogna utilizzare l'approccio Lazy, in caso contrario farebbe esplodere tutto. Quindi presuppone l'utilizzo della strategia **call by name** perché se usato con la strategia **call by value** diverge.

Il combinatore Z → il combinatore Y modificato per funzionare con la call by value viene anche chiamato combinatore Z.

$$Z = \lambda f(\lambda x.f(\boldsymbol{\lambda v.((x\ x)v)}))(\lambda x.f(\boldsymbol{\lambda v.((x\ x)v)}))$$

In sostanza viene piazzato in mezzo v che non fa altro che ritardarlo.

21. INTRODUZIONE A SCALA

Scala è un linguaggio scalabile (da piccoli script a grandi sistemi) e java based (compila in bytecode e gira su JVM). È un linguaggio blended cioè è sia object-oriented (stile imperativo) sia funzionale (con chiusure e funzioni first-class entities)

Tipi → non esistono tipi primitivi, sostituiti da tipi valore Int, Float (simi ai Wrapper) e rimappati sui tipi primitivi Java a livello di bytecode. Alla radice della gerarchia **Any** non è Object. Any è il padre generale di **AnyRef** e **AnyValue:** AnyRef è il vecchio Object di Java mentre AnyValue è il padre dei valori immodificabili

Valori e variabili → valori (immodificabili) e variabili (modificabili) sono enti distinti introdotti da specifiche parole chiave: **val** e **var**. la specifica di tipo è postfissa (anziché prefissa come in C e Java), inoltre non è obbligatoria se può essere inferita. I tipi Java sono comunque tutti disponibili in Scala.

Funzioni → nuova sintassi introdotta dalla parola chiave def. Specifica di tipo postfissa (che è possibile omettere se si può inferire).

Ottimizzazione della tail recursion e funzioni intese come first-class entites, in grado di generare vere chiusure.

Classi → costruttore primario come single point of entry definito a partire dai parametri di classe. Costruttori ausiliari che si rimappano sul costruttore primario. La sintassi in Scala è più compatta poiché non viene utilizzato il concetto di this come in Java.

Singleton Objects → nuova sintassi introdotta dalla parola chiave **object** che permette di definire oggetti singleton.

Companion objects → la parte statica di una classe è spostata nel suo oggetto compagno, cioè un singleton omonimo definito nello stesso file.

Tratti → evoluzione dell' interfacce, possono contenere codice. Una classe che eredita da un tratto ne implementa l'interfaccia ma contemporaneamente ne eredita anche il codice.

Oltre l'ereditarietà multipla → rimane vero che una classe Scala può ereditare sa una sola classe, ma può mixarsi tra più tratti, questo perché nei tratti non c'è il costruttore e quindi si passerà sempre dallo stesso costruttore in maniera controllata.

Pattern matching → una nuova sintassi basata sulle parole chiave **match** e **case** permette di fare pattern matching su oggetti e classi.

Case Classes → nuova sintassi tramite il costrutto **case class** che permette di definire classi speciali (case classes) che godono di una gestione particolare, largamente automatizzata e semplificata.

Strutture di controllo → scala ha solo 5 strutture di controllo predefinite: if, while, for, try e match. Le strutture di controllo denotano valori poiché sono espressioni e non più istruzioni.

Il tipo Unit → una funzione che non ritorna valori (procedura) ha per convenzione il tipo di ritorno **Unit**. È anche il tipo di ritorno degli assegnamenti (che vengono considerati anch'essi espressioni).

Main → il main in Scala non sta in una classe ma in un oggetto singleton – un object. Di conseguenza cambia leggermente la dichiarazione che diventa:

def main(args: Array[String])

for → il for in scala nonostante la somiglianza sintattica è ben diverso da quello in Java. L'argomento s in effetti è un valore e non una variabile ma soprattutto l'espressione interna è un generatore (cioè genera una serie di valori su cui iterare)

for (s <- args) { println("Argomento: " + s) }

yield → spesso lo scopo di un'iterazione è di accumulare in qualche struttura dati I risultati parziali via via calcolati e questo si può fare facilmente con la keyword **yield**

sintassi infissa → 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

```
Console.println(5)          Console println 5 //ILLEGGIBILE
```

Ancora più in generale ogni operatore infisso è una chiamata di metodo.

Parentesi come shortcut → in generale una lista di valori fra parentesi usata come R-value sottintende sempre una chiamata implicita al metodo apply dell'oggetto ricevente.

```
pippo(a,b,c,...)            pippo.apply(a,b,b,...)
```

Analogamente una lista di valori fra parentesi usata come L-value sottointende una chiamata implicita al metodo updata dell'oggetto ricevente.

```
pippo(a,b,...) = q          pippo.update(a,b,...,q)
```

visibilità → a differenza di java in Scala i membri delle classi sono pubblici per default, mentre i parametri di classe sono privati.

Sintassi speciali per identificatori speciali → il backtick (`) rende legale qualunque sequenza di caratteri come identificatore. Inoltre una sequenza di caratteri operatore è un unico operatore. L'underscore ha un significato particolare perché introduce gli identificatori misti.

Identificatori misti → il carattere underscore introduce i mixed identifiers. PAROLA + UNDERSCORE + OPERATORE

Eccezioni → le eccezioni in Scala sono simili a Java ma come in C# non esiste la clausola throws, anche l'espressione throw ha il tipo di ritorno Nothing e la clausola catch opera per pattern matching.

Espressioni match → il costrutto switch è sostituito in Scala dalla più potente espressione match. Match non opera solo su valori scalari o stringhe ma anche su pattern.

Letterale funzione → un letterale funzione (lambda) ha la forma generale

(argomenti) => blocco

Il blocco può essere costituito da un'unica espressione, ma se è costituito da più di una espressione il tipo e il risultato sono quelli dell'espressione più a destra. Un letterale tecnicamente è compilato su classi Function0, Function1, le cui istanze rappresentano a run time valori-funzione di quell'arità.

Può essere eseguita direttamente con l'operatore di chiamata () oppure indirettamente tramite il metodo apply della classe FunctionN.

Chiusure → il pieno supporto alle funzioni come first-class objects comporta un altrettanto pieno supporto alle chiusure. Differenza tra Javascript e Scala: le variabili qui sono chiuse su riferimenti alle variabili esterne e quindi percepiscono i loro cambiamenti nel tempo.

Nuove strutture di controllo → scala fornisce gli strumenti per definire nuovi costrutti anziché definirne a priori un insieme fisso e prestabilito.

- **Block like syntax**: per passare un argomento usando parentesi graffe anziché tonde in modo che non sembri una funzione (in questo modo sembreranno built-in di Scala e non creati dall'utente)
- **Call by name**: nel passaggio degli argomenti. Valuta un argomento al momento dell'uso non a priori. Inoltre in scala non è necessario emularla, perché un argomento passa automaticamente by name se preceduto da =>
- **Currying**: possibilità di applicare una funzione a più argomenti una serie di argomenti in sequenza, uno dopo l'altro anziché tutti insieme nella classica lista di argomenti.

Funzioni senza argomenti → a differenza di java scalla ammette funzioni senza lista di argomenti (da non confondere con le funzioni con lista di argomenti vuota).

- Le **funzioni senza lista di argomenti** deve essere invocata senza l'operatore (). Queste funzioni supportano il principio di accesso uniforme (un cliente non deve sapere se una data proprietà sia una funzione o un dato. Principalmente vengono usate per i puri accessor, ossia funzioni che leggono dati senza modificarli.
- Una **funzione con lista di argomenti** vuota invece può essere invocata con o senza l'operatore ().

Classi astratte e ereditarietà → una classe astratta in Scala è molto simile a Java (parola chiave **abstract** all'inizio della classe). Una classe derivata si definisce come in java (parola chiave **extends** nella dichiarazione).

Tassonomia Scala → la classe base **Any** definisce i metodi generali, mentre le due sottoclassi **AnyVal** e **AnyRef** fanno da base rispettivamente ai valori e ai riferimenti.

Le sottoclassi di AnyVal sono tutte astratte e finali. AnyRef invece è la radice dei tipi-riferimento.

Bottom type → scala definisce dei botto 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
- **Nothing** è il sottotipo di chiunque: non ha istanze serve a rendere coerente il type system nelle terminazioni anomale.

Tratti e composizionalità → scala supera le interfacce introducendo i tratti (**trait**), che sono più simili a classi che a interfacce e possono contenere codice.

Una classe può estendere un solo tratto (**extends**) e comporsi di molti tratti (**with**, in questo modo forma un mix-in).

Un tratto può estendere una classe (o un altro tratto) e inoltre è una specifica di vincolo che un tratto potrà essere mixato con sottoclassi della classe che lui stesso estende.

Linearizzazione → le chiamate a super sono risolte per linearizzazione. Si percorre quindi la sequenza di extends/with con criterio prestabilito, ottenendo una lista di strati. Ciò funziona poiché i tratti non hanno costruttori.

Scala collections → la scala collection sono fornite in doppia versione: mutable e immutable (migliori sotto ogni punto di vista)

- **List**: rappresenta una lista immutabile. È una classe (non un'interfaccia)
- **Tuple**: la classe rappresenta una tupla immutabile, i cui elementi sono numerati a partire da 1 (e non da 0)
- **Set**: è un tratto e esiste in doppia versione. È implementato dalla classe HashSet
- **Map**: anche Map è un tratto ed esiste in doppia versione. È implementato dalla classe HashSet.

Essendo scala compilato in bytecode, l'interoperatività Scala/Java è garantita, infatti molte cose restano identiche, altre cose invece si riescono a mappare 1-1 (valori primitivi). Però ci sono anche alcune novità però che richiedono un mapping ad hoc:

- Oggetti singleton → classe con metodi statici
- Tratti → interfaccia + classe accessoria
- Annotazioni → doppio processing (Scala, poi Java)
- Tipi generici → molto simili ma con dettagli diversi

Da scala si possono usare classi Java così come sono, scrivere classi Scala che stendano classi Java, istanziare classi java, accedere a metodi o campi-dati anche statici e sfruttare framework come Swing.

Da Java invece si possono usare classi Scala così come sono, usare oggetti Scala, usare classi Scala che rappresentino funzioni e chiusure, e ingenerale fare quasi tutto.

Conversioni automatiche di strutture dati Java/Scala → nonostante i nomi analoghi le classi-collection Scala non sono le stesse definite da Java, perché sono a tutti gli effetti tipo diversi, che quindi non possono essere passati uno al posto dell'altro. L'oggetto **collections.JavaConversions** fornisce una serie si conversioni implicite pronte all'uso.

Iterator	<->	java.utili.Iterator
Iterator	<->	java.util.Enumeration
Iterable	<->	java.lang.Iterable
Iterable	<->	java.util.Collection
Mutable.Buffer	<->	java.util.List
Mutable.Set	<->	java.util.Set
Mutable.Map	<->	java.util.Map
Mutable.ConcurrentMap	<->	java.util.concurrent.ConcurrentMap

Altre conversioni invece funzionano in un solo verso:

Seq	->	java.utili.List
Mutable.Seq	->	java.util.List
Set	->	java.lang.Set
Map	->	java.util.Map

Il motivo per cui non operano a rovescio è che Scala definisce quelle classi in doppia versione (mutevole e immutabile)

Interoperatibilità Scala/Java → Java prevede wildcard e tipi "raw" mentre Scala no e questo potrebbe essere un problema. Per gestirli Scala definisce i tipi esistenziali.

Case classes → Scala introduce case classes come modo compatto per esprimere oggetti su cui fare pattern matching. Una case class è sintatticamente introdotta dal codificatore **case**, mentre semanticamente è gestita in modo speciale dal compilatore.

Le case classes sono il presupposto naturale su cui Scala definisce il meccanismo di pattern matching.

Espressione match: *selector* ***match*** *{ alternatives }*

Dove le alternatives sono: ***case*** *pattern => expressions*

Pattern matching per definire letterali funzione → il costrutto case è più potente di quanto visto fino ad adesso, infatti una sequenza di case tra graffe costituisce un function literal.

Se accade che una serie di case non copra tutti i casi reali allora abbiamo un'esplosione a runtime. Però si può controllare il rischio definendo le funzioni parziali, in questo caso il compilatore avviserà se ci sono casi non coperti.

Inoltre i costrutto **for** si sposa perfettamente con il pattern matching (lavora in un modo molto simile all'unificazione Prolog).

Classe sealed → una classe sealed ammette come uniche sottoclassi quelle definite nello stesso file della classe-base. Questo meccanismo è utile nel pattern matching proprio per essere certi di aver coperto tutti i casi.

Tipo Option → il tipo speciale **Option** permette di esprimere parametri opzionali e può avere due valori possibili: **None** e **Some(qualcosa)**

Proprietà e metodi accessor → in java le proprietà non esistono mentre in Scala esistono senza richiedere una sintassi speciale. Semplicemente i metodi accessor sono generati in automatico.

Extractor → i pattern richiedono case classes, altrimenti la notazione Costruttore(args) non si può usare. Sarebbe bello poter usare tale approccio sempre e a questo scopo Scala offre gli estrattori, cioè oggetti che definiscono un metodo **unapply** che fa match con un valore e lo estrae (concettualmente è il duale del metodo **apply** che invece lega valori assieme).

L'extractor viene usato spesso nelle Scala collection, infatti tutte le notazioni che permettono di accedere ad array e liste come Array(), List(), ... sono tutte realizzate con extractor.

L'extractor viene utilizzato anche nelle espressioni regolari. In Scala infatti permettono di fare match su pezzi di stringhe descritte da espressioni regolari.

Grafica → la grafica Scala si appoggia su Swing. Stesse astrazioni e stesso approccio solo che i nomi sono stati ripuliti dalla "J" iniziale.

Però **scala.swing** definisce un livello extra. La classe **SimpleSwingApplication** definisce l'essenziale di una applicazione grafica, cioè un main frame con alcune proprietà.

- In Java Swing tipicamente si crea un pannello e gli si attribuisce un layout, a quel punto si aggiungono componenti al pannello
- In Scala la classe Panel è astratta, le sottoclassi concrete adottano un layout (BorderPanle, BoxPanel, FlowPanel, ...) e si aggiungono componenti alla proprietà **contents**

Parser combinators → Scala permette di specificare in modo diretto sintassi e semantica di un linguaggio tramite combinatori

- ~ esprime concatenazione
- Rep(..) esprime ripetizione del contenuto
- | esprime alternativa
- parentesi vengono usate per raggruppare

la propria classe deve estendere **JavaTokenParsers** e per scatenare il parser bisogna invocare la funzione **parseAll**.

Di default il parser generato è in grado di supportare il non determinismo. Se la grammatica è LL(1) allor conviene specificarlo con l'apposito combinatore ~! (da notare il simbolo ! del cut in Prolog) in tal caso il parser sarà deterministico.

Per inserire azioni semantiche invece bisogna specificare una funzione Scala accanto alla regola sintattica usando l'apposito combinatore ^^.