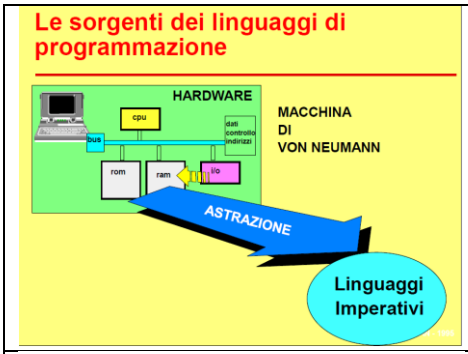
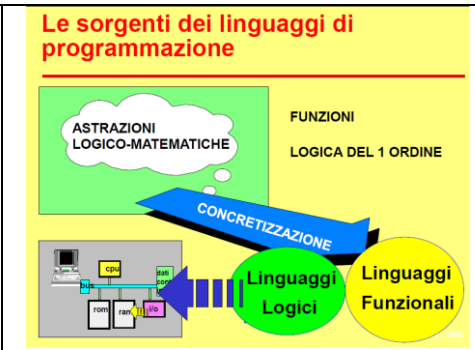
Riassunto di Linguaggi e Modelli Computazionali (E. Denti) A.A. 2016-17

Contents

Capitolo 0 – Introduzione

Storicamente il linguaggio di programmazione è lo strumento che consente di eseguire le operazioni desiderate ad un livello più alto di quello consentito dall'hardware, viene lasciato ai compilatori e interpreti sottostanti il ruolo di colmare il divario. Oggi tra linguaggio di programmazione e macchina ci sono diversi livelli formati dalle macchine virtuali che consentono un'analisi sintattica e semantica dei comandi prima dell'esecuzione.
Solitamente vengono adottate due metodologie risolutive per risolvere problemi che ci vengono presentati: Top-Down e Bottom-Up.
Spesso oltre a queste due metodi risolutivi applichiamo anche inconsciamente concetti del linguaggio che utilizziamo (es. le classi, thread e oggetti in java; processi in C), per questo motivo ci sono diversi linguaggi di programmazione: l'introduzione di nuovi concetti che ogni linguaggio può introdurre e che facilitano la risoluzione di un particolare problema in un certo dominio di applicazione.

Ci sono poi 2 tipologie di linguaggi:

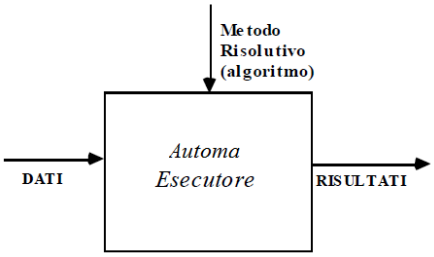
	
***Imperativi**: danno al processore/hardware gli ordini sulle istruzioni da eseguire. L'obiettivo è assegnare una variabile il valore che andrà restituito come risultato.*	***Funzionali**: definiscono funzioni da eseguire consecutivamente e in maniera ricorsiva. Qui gli assegnamenti alle variabili non sono concessi, l'obiettivo è ottenere un risultato sfruttando funzioni e sotto-funzioni.*

Vista la varietà di modelli computazionali utilizzabili, per sfruttare le comodità che offre ognuno di questi linguaggi può aver senso sfruttarne i punti di forza di ogni linguaggio con una programmazione **multi-paradigma**.

Capitolo 1 – Linguaggi e macchine astratte

Alcune **domande**: Quali istruzioni esegue un computer? Può risolvere qualunque problema ci venga in mente? Quali problemi non può risolvere? Quanto conta il linguaggio di programmazione in tutto questo?

Automa Esecutore: macchina astratta capace di eseguire le azioni di un algoritmo dategli in pasto.

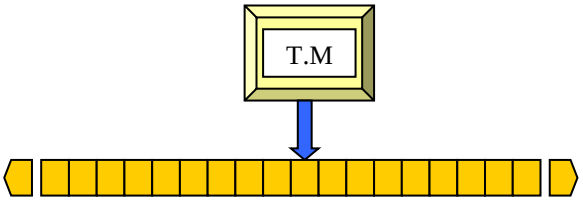


Formalizzando la definizione di automa ottengo il concetto di computabilità. Ottengo dall'esterno un algoritmo che l'automa deve essere in grado di interpretare con un qualche linguaggio e devo dare in uscita una risposta, un risultato.

Stabiliamo una **gerarchia** di macchine astratte e le analizziamo per capire quali sono i problemi che non potremo mai risolvere.

GERARCHIA DI MACCHINE
- macchine base (combinatorie)
- macchine (automi) a stati finiti
- **macchina a stack (PDA)**
- macchina di Turing

Macchina base: combinatoria, senza memoria, collega un ingresso ad un'uscita.
Automa a stati finiti: contiene anche uno stato che le dà un minimo di memoria in più rispetto alla macchina base.
Macchina a stack (PDA): ha un supporto di memorizzazione (es. un nastro) su cui andare a leggere/scrivere solo in un senso e può solo azzerare l'indice e partire dall'inizio.
Macchina di Turing: può leggere/scrivere in entrambi i sensi, come preferisce. Raggiunto l'halt del programma si ferma.



Stabilita la macchina di Turing come la macchina più potente che abbiamo (come specificato nella tesi di Church-Turing) desidereremmo però non avere solo una macchina special purpose, che non è riutilizzabile e si occupa di 1 solo compito, ma sarebbe buono avere una macchina UNIVERSALE, general purpose.
Potremmo per esempio crearla inserendo l'algoritmo da analizzare sul nastro in modo da inserire in input sempre un algoritmo.
È possibile confrontare la macchina di Turing con quella di Von Neumann, alla base di calcolatori generici: le operazioni sono rispettivamente analoghe fra loro.

Le due macchine si distinguono in maniera sostanziale dalla limitatezza della macchina di Turing ad approcciarsi col mondo esterno: opera nel proprio nastro (o memoria RAM), ma non è in grado di ricevere istruzioni di I/O.

In pratica:	corrisponde a:
♦ leggere / scrivere un simbolo dal / sul nastro	♦ lettura / scrittura dalla / sulla memoria RAM / ROM
♦ transitare in un nuovo stato interno	♦ nuova configurazione dei registri della CPU
♦ spostarsi sul nastro di una (o più) posizioni	♦ scelta della cella di memoria su cui operare (indirizzo contenuto nell'*Address Register*)

Alla MdT manca coordinazione ovvero la possibilità di ottenere informazioni I/O dal/verso il mondo esterno. Possiede invece facoltà di computazione per analizzare l'algoritmo ed eseguirlo.

Teoria della computabilità

Problema Risolubile→ problema la cui soluzione può essere espressa da una MdT o un formalismo equivalente.

Per riuscire a capire qual è il nostro limite di computabilità do la definizione di ***Funzione Caratteristica***: è quella funzione che dato in ingresso un valore, mi associa in uscita la corrispondente risposta corretta.

Detto questo dobbiamo anche distinguere le funzioni che siamo in grado di computare con una MdT da quelle che siamo solo in grado di definire.
Prendendo in considerazione come esempio il solo insieme di numeri naturali per semplicità, scopriamo che (tramite il **procedimento di Gödel**) gran parte delle funzioni definibili non è computabile. Infatti se associamo ad ogni MdT un numero naturale, sappiamo che l'insieme N non è enumerabile, mentre l'insieme delle MdT e quindi delle funzioni computabili è enumerabile. Otteniamo così il risultato che la maggior parte delle funzioni non può essere calcolata.
Scopro così grazie al **Problema dell'Halt della MdT**, che esistono problemi irrisolubili.
Concludo che è INDECIDIBILE capire se una MdT con ingresso generico si fermi oppure no.

Ci interessa approfondire l'argomento decidibilità vs generabilità, così ottengo che:

- Un insieme è semi-decidibile (o ricorsivamente enumerabile) se la funzione è computabile e può computando generare ad uno ad uno gli elementi dell'insieme. Con la semi-decidibilità posso stabilire se un elemento generato appartiene ad un insieme, ma non so se "non appartiene" a tale insieme;
- Un insieme è decidibile se riesco a stabilire se un elemento appartiene o no ad un insieme;
- Un insieme è decidibile se riesco a stabilire se un elemento appartiene o non appartiene a tale insieme senza entrare in un ciclo infinito;

Questo è importante perché nei linguaggi di programmazione abbiamo un alfabeto da rispettare e vorremmo che un compilatore ci dicesse se una frase è giusta o sbagliata senza entrare in un loop infinito che non ci porterebbe risposta.

Capitolo 2 – Linguaggi e Grammatiche

Alcune domande: in un linguaggio quali sono le frasi lecite? Si può stabilire se una frase appartiene al linguaggio? Come si stabilisce il significato di una frase? Quali elementi linguistici primitivi ha?

Un linguaggio ha sintassi e semantica, vediamo quindi di precisare cosa sono e che ruolo hanno questi 2 aspetti del linguaggio:

Sintassi→ regole formali per costruire frasi corrette nel linguaggio (la esprimo con diagrammi sintattici, BNF e EBNF);

Semantica→ significato da attribuire alle frasi costruite nel linguaggio (la esprimo con parole, azioni, funzioni matematiche, funzioni logiche);

precisiamo anche la differenza tra un interprete e un compilatore:

Interprete→ prende in input le singole frasi di un linguaggio e ne valuta la correttezza;

Compilatore→ prende in input un intero programma e lo riscrive in un linguaggio più semplice;

sapendo quindi che si possono costruire frasi sintatticamente corrette, ma con semantica errata o senza senso siamo portati ad analizzare i vari tipi di correttori di errori:

Scanner/Lexer→ analizzatore lessicale, individua le singole parole (token) di una frase;

Parser→ verifica che la sequenza di token rispetti le regole grammaticali del linguaggio;

Analizzatore semantico→ data la rappresentazione del parser determina il significato di una frase, che non è facile da determinare a priori;

per dare senso alla semantica è necessario definire un **Alfabeto**, ovvero un insieme di simboli atomici come parole, simboli o frasi.

Combineremo sequenze di simboli dell'alfabeto per formare Stringhe e sarà importante ricordare che la lettera ε identificherà la stringa vuota.

È utile ricordare anche altre 2 notazioni

1. Chiusura A^*→ che è l'insieme infinito di tutte le stringhe composte coi simboli dell'alfabeto A;
2. Chiusura positiva A^+→ è l'insieme infinito di tutte le stringhe non nulle dell'alfabeto;

Definito quindi l'alfabeto del linguaggio è necessaria una grammatica formale per specificare quali combinazioni di simboli sono sintatticamente corretti. Definiremo quindi come **Grammatica Formale** la quadrupla <VT,VN,P,S> → VT sono i simboli facenti parte dell'alfabeto (terminali), VN sono i meta-simboli per le diverse categorie sintattiche (non terminali), P sono le produzioni (o regole di riscrittura) infine S è lo scopo della grammatica.

Derivazione ➔ β deriva da α se esiste una sequenza di N derivazioni dirette che da α possono infine produrre β.

La **sequenza di derivazione** è il numero di produzioni necessarie per derivare dallo scopo S la frase σ, ci sono varie notazioni per indicare il numero di passi necessari per ottenere σ:

$S \Rightarrow \sigma$ indica la derivazione di σ da S con 1 sola produzione necessaria di distanza

$S \overset{+}{\Rightarrow} \sigma$ indica la derivazione con 1 o più produzioni necessarie

$S \overset{*}{\Rightarrow} \sigma$ indica la derivazione con 0 o più produzioni necessarie

In linea teorica possiamo specificare come equivalenti due grammatiche che generano lo stesso linguaggio, in realtà però stabilire se due grammatiche sono equivalenti è un problema indecidibile.

CLASSIFICAZIONE DI CHOMSKY

Le grammatiche possono essere classificate in 4 tipi e al crescere del numero del tipo si ha una grammatica sempre più restrittiva:

Tipo 0→ nessuna restrizione sulle produzioni: permette di accorciare la forma di frase perché permette anche la stringa vuota nella trasformazione (questo è un problema);
Tipo 1→ produzioni vincolate alla forma (dipendenti dal contesto), non accorciano mai la forma di frase;
Tipo 2→ produzioni vincolate alla forma (libere dal contesto 1 sola sostituzione alla volta). In certe situazioni è utile poter scambiare pezzi, questo non sarà mai possibile con un Tipo2 (context-free);
Tipo 3→ produzioni che devono essere tutte o lineari a destra o lineari a sinistra (grammatiche regolari);

The Hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursive Enumerable	Turing Machine
Type-1	Context Sensitive	Context Sensitive	Linear-Bound
Type-2	Context Free	Context Free	Pushdown
Type-3	Regular	Regular	Finite

Esempi di tipologie di grammatiche:

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

Problema della stringa vuota:
possiamo notare che la grammatica di Tipo 1 non ammette la stringa vuota ε, mentre le grammatiche di Tipo 2 e Tipo 3 la ammettono. Considerando la gerarchia sempre più restrittiva questa è una contraddizione! Che si spiega solo grazie ad un teorema che dice che "le produzioni di tipo 2 e 3 possono sempre essere riscritte in modo da evitare la stringa vuota".
Un linguaggio quindi può essere generato da più grammatiche anche di tipo diverso.

Differenze fra le caratteristiche dei tipi sono che dall'1 al 2 non posso più scambiare due simboli, dal 2 al 3 non posso avere un meta-simbolo in qualunque posizione ma solo alla fine o all'inizio.

Self-embedding→ aggiunta di simboli in contemporanea sia a destra che a sinistra

Il self-embedding è anche ciò che distingue una grammatica di tipo 2 da una di tipo 3: infatti se una grammatica è di tipo 2 e non aggiunge simboli da entrambi i lati può essere anche di tipo 3, se invece li aggiunge (ha self-embedding) è di tipo 2.

Grammatica che presenta self-embedding: S→aSc S→A A→bAc A→ε

Grazie a regole meno restrittive applicate una dopo l'altra può essere vanificato il vincolo di self-embedding e possiamo ottenere un linguaggio regolare.

Arrivando al dunque, dopo tutta questa distinzione, posso dire che una MdT è in grado di decidere se una frase appartiene o meno a un linguaggio in tutti i casi ad eccetto il Tipo 0.

parentesi sulla ricorsione diretta
$X = u X + \delta$ può essere riscritta come $X = u^* \delta$

Utilizziamo quindi solitamente grammatiche di Tipo 2 per avere una traduzione efficiente e con tempi non troppo lunghi come potrei avere col Tipo 1, anche se è appunto computabile con la MdT. Per essere particolarmente efficienti in pezzi di codice che utilizziamo frequentemente si può utilizzare un linguaggio con grammatica di Tipo 3.

GRAMMATICHE	AUTOMI RICONOSCITORI
• Tipo 0	• *Se L(G) è riconoscibile*, serve una Macchina di Turing
• Tipo 1	• Macchina di Turing *(con nastro di lunghezza proporzionale alla frase da riconoscere)*
• Tipo 2 *(context-free)*	• Push-Down Automaton (PDA) *(cioè ASF + stack)*
• Tipo 3 *(regolari)*	• Automa a Stati Finiti (ASF)

Ci concentreremo quindi su grammatiche di tipo 2 e cambieremo anche la modalità di esprimere la grammatica poiché quella finora usata è un po' scomoda e ambigua.

GRAMMATICA BNF

Regole di produzione→ ::=
Meta-simboli→ <nome>
Meta-simbolo "alternativa"→ |

GRAMMATICA EBNF (BNF estesa)

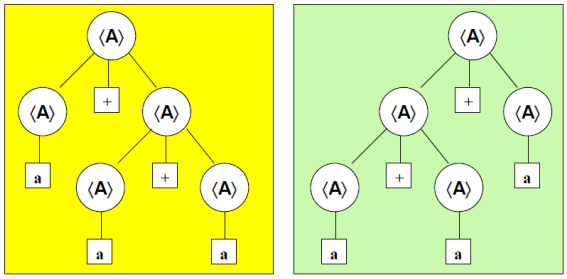
Comparizione di un simbolo 0 o 1 volta→ [ ]
Comparizione di un simbolo da 0 a N volte → $\{\}^N$
Comparizione di un simbolo 0 o più volte → { }
Raggruppamento di categorie sintattiche → ( )

Per le sole grammatiche di tipo 2 (quindi anche per il tipo 3) posso ottenere un albero di derivazione, per le grammatiche di tipo 0 e 1 non posso poiché potrei avere più di un simbolo per il lato sinistro e dunque avrei un generico grafo e non più un albero.

Derivazioni Left e Right-Most→ parto dallo Scopo e riscrivo il simbolo non terminale più a sx/dx

Possiamo classificare una grammatica come Ambigua quando esiste almeno 1 frase che ammette 2 derivazioni canoniche sinistre.
Spesso se una grammatica è ambigua si può trovare una grammatica analoga che non lo sia.

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



2 possibilità per costruire un albero sintattico diverso con la stessa frase

La stringa vuota può far parte delle frasi di una grammatica di Tipo 0 e non degli altri tipi. E come detto prima ricordiamo di non essere in contraddizione poiché col tipo 2 e 3 si può sempre esprimere una grammatica equivalente senza ε-rules.

Possiamo scrivere un linguaggio di tipo 2 facendo in modo che le produzioni abbiano tutte una forma ben precisa. In particolare esistono due standard di **forme normali** applicabili: quella di **Chomsky** e quella di **Greibach**. Quest'ultima ci aiuterà notevolmente a costruire con più semplicità i riconoscitori.

Quella di Chomsky è facilmente ottenibile tramite il banale algoritmo di trasformazione.
Quella di Greibach si può ottenere con trasformazioni notevoli come la sostituzione, il raccoglimento a fattor comune e l'eliminazione della ricorsione sinistra.

Queste trasformazioni sono utili per non far incasinare la macchina nell'elaborazione, roba semplice per noi ma non troppo per il pc che può impiegare ore per fare una cosa che a noi fa perdere solo qualche minuto per farla al posto del pc.

ESEMPIO
S → X a
X → b Q | S c | d

ESEMPIO
S → X a
X → b Q | X a c | d

Sostituzione

ESEMPIO
S → a S b | a S c

ESEMPIO
S → a S (b | c)

ESEMPIO
S → a S X
X → b | c

Raccoglimento a fattor comune

ESEMPIO
A → B a
B → C b
C → A c | p

Si ottiene quindi:
A → B a
B → C b
C → C b a c | p

Ergo, C → C b a c | p diventa
C → p | p Z
Z → b a c | b a c Z

Eliminazione della ricorsione sinistra

Pumping Lemma

Lemma del pompaggio perché per capire se una grammatica è di un tipo o di un altro posso pompare tutte le lettere che voglio. Utilizzando il pumping lemma posso riconoscere per esclusione se un linguaggio è del tipo 3, 2 oppure non è di nessuno dei due tipi, quindi sarà del tipo 1. La condizione del lemma in questo caso è necessaria ma non sufficiente per riconoscere completamente la tipologia di linguaggio utilizzato, perciò vado per esclusione.
La proprietà, che appartiene ai **linguaggi regolari**, è: una stringa qualunque di un dato linguaggio, **deve poter essere divisa in 3 parti** (chiamate x, y e z) tali che la stringa xy^nz ottenuta pompando n volte l'elemento centrale, appartenga ancora ad L. In altri termini, ogni stringa deve contenere una sottostringa che possa essere ripetuta un qualunque numero di volte.
La spiegazione logica del pumping lemma fa leva sul fatto che ogni stringa sufficientemente lunga si ripete, perciò posso pompare una stringa ottenendo sempre altre stringhe del linguaggio.

Ho così linguaggi di tipo 2 se ho self-embedding, ovvero se non mi si ripete la stringa centrale e mi si ripetono invece le stringhe ai lati (es. $L=\{(11)^n(1|0)^m(1)^n,\ n>1,\ m>1\ \}$).
Ho un linguaggio di tipo 1 se ho la ripetizione di 3 caratteri, quindi non posso pompare nessuno dei 3 (es. $L=\{1^n2^{2n}3^n,\ n>1\}$).

Le **Regular Expression** sono uno strumento formale per definire linguaggi con una struttura semplice, non sono invece adatte per definire linguaggi complessi come quelli di programmazione.
Definiamo come regular expression quelle formate da:

1. Gli elementi di un alfabeto;
2. La stringa vuota (ε);
3. La concatenazione, unione e chiusura fra loro di diversi linguaggi;

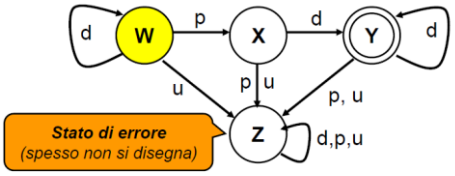
Le espressioni regolari ci interessano perché, paragonate alla grammatica, sono due facce della stessa medaglia. Entrambe infatti mostrano una caratteristica del linguaggio:
grammatica=dice come si fa il linguaggio
espressione=dice cosa si ottiene
si può passare dall'una all'altra notazione utilizzando un algoritmo diverso a seconda che la grammatica sia regolare destra o sinistra. La differenza sta nel raccoglimento a fattor comune e quindi nella posizione finale dei termini.
In effetti guardando un paio di esempi si nota subito che un linguaggio può essere denotato da diverse espressioni regolari equivalenti che sembrano non assomigliarsi neanche un po' es:

$S = (a^* a\ d)^*\ a^* a\ b$

$S = (a\ d + a)^* a\ b$

Capitolo 3 – Automi riconoscitori

Un linguaggio di tipo 3 è detto regolare ed è riconoscibile da un ASF, mentre linguaggi di tipo 2 e 1 sono più complicati da riconoscere e addirittura quelli di tipo 0 non sono riconoscibili. In particolare l'automa utilizzabile per riconoscere un linguaggio di tipo 3 è detto **Riconoscitore a Stati Finiti** ed è una specializzazione di un automa.
In particolare un riconoscitore è formato da una quintupla <A, S, S_0, F, sfn*> → A è l'alfabeto, S è l'insieme degli stati, S_0 è lo stato iniziale, F è l'insieme degli stati finali, sfn è la state function. Il riconoscitore può accettare o non accettare frasi. Se una frase è accettata significa che dallo stato iniziale riesce ad arrivare ad uno stato finale. Posso descrivere gli stati di un automa attraverso il suo diagramma degli stati →

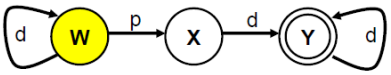


Diamo poi la definizione di **linguaggio non vuoto e linguaggio infinito**:
il linguaggio è non vuoto se il riconoscitore accetta una stringa di lunghezza inferiore al numero N di stati dell'automa.
Un linguaggio è invece infinito se il riconoscitore accetta una stringa di lunghezza(L_x) → $N \leq L_x < 2N$
Con la grammatica scopriamo che è possibile, facendo un mapping fra gli elementi, costruire un riconoscitore o viceversa risalire alla grammatica dal RSF.

Dal diagramma degli stati dell'automa riconoscitore posso ricavare la grammatica utilizzando 2 approcci:

1. Grammatica regolare a destra
2. Grammatica regolare a sinistra

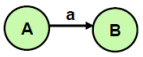
Sono semplicemente due modi di vedere e scrivere lo stesso diagramma: uno guardando avanti e vedendo sempre lo stato successivo e uno guardando invece lo stato precedente e guardando indietro.



Grammatica regolare a destra	Grammatica regolare a sinistra
W → d W | p X	X d | Y d ← Y
X → d | d Y	p | W p ← X
Y → d | d Y	d | W d ← W

Dei casi particolari di bottom-up possono complicare la situazione con più stati finali.

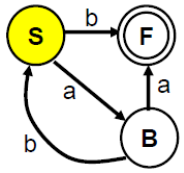
Caso generico:



Top-down: A → a B
Bottom-up: A a ← B

Riconoscitore Top-Down

S → a B | b
B → b S | a

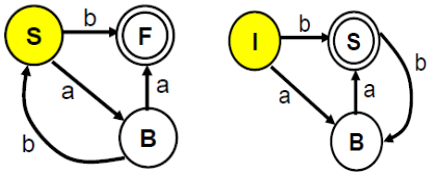


Con questa grammatica lineare a destra posso ottenere il diagramma semplicemente disegnando gli stati e le transizioni da uno stato all'altro.

Se invece voglio ottenere la frase del linguaggio partendo dal diagramma dell'automa posso, partendo dallo scopo iniziale S, "coprire" la frase con le produzioni successive:

$S \rightarrow aB \rightarrow abS \rightarrow abaB \rightarrow abaa$

Parto così dallo scopo (stato iniziale S) e vado verso lo stato finale. Mi fermo quando lo raggiungo.

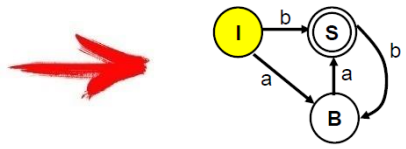


Top-Down Bottom-Up

Riconoscitore Bottom-Up

Dalla grammatica come nel caso top-down posso semplicemente ottenere il diagramma del riconoscitore andando a inserire uno stato per ogni metasimbolo (es. S, B, iniziale I), e disegnandone le transizioni (a, b):

$S \rightarrow Ba \mid b$
$B \rightarrow Sb \mid a$



Posso da qui ottenere la grammatica partendo da una frase del linguaggio:

$aaba \rightarrow Baba \rightarrow Sba \rightarrow Ba \rightarrow S$

Partendo dallo stato iniziale I riduco la frase di volta in volta fino a raggiungere lo scopo (stato finale S).

Considerazioni su Bottom-Up e Top-Down

Come posso ben vedere quindi l'approccio top-down ha uno stato finale F nel diagramma degli stati, che non è presente nello stesso diagramma bottom-up e viceversa nel bottom-up c'è uno stato iniziale I che non è presente nel top-down.

Grammatica regolare a destra ⇔automa riconoscitore "top-down" ⇔ generazione della frase
Grammatica regolare a sinistra ⇔ automa riconoscitore "bottom-up" ⇔ riduzione della frase

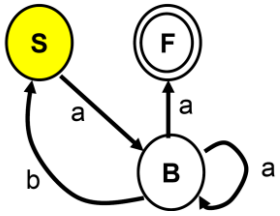
Trasformazione ottenibile da un verso e dall'altro: dal riconoscitore posso ottenere la grammatica e dalla grammatica posso ottenere il riconoscitore.

Un problema in cui ci si può imbattere è la presenza di più stati finali nel bottom-up e di più stati iniziali nel top-down. Si può valutare se:

1. Trattare in modo particolare lo stato iniziale/finale e non avere presenza di ε-rules;
2. Trattare in modo standard gli stati accettando così la presenza delle ε-rules;

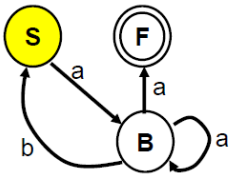
Implementazione di un RSF – distinzione fra non determinismo e determinismo
Implementare un riconoscitore non deterministico non è un problema. Inizia ad esserlo quando il riconoscitore è deterministico e il linguaggio utilizzato non supporta il non determinismo.

Automa non deterministico → è un automa nella quale, all'interno della tabella delle transizioni, ci sono più stati futuri per una stessa configurazione. Nel nostro caso infatti, in corrispondenza dello stato B bisogna sapere se è meglio portarsi in B o in F utilizzando sempre la transizione 'a'.



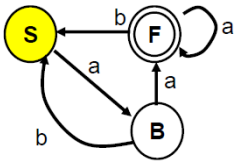
<u>Deterministico vs Non Deterministico</u>

Un riconoscitore deterministico ha solamente in più la capacità di ricordare la strada fatta e se necessario disfarla per esplorarne un'altra. Tutto questo è implementabile anche con un linguaggio che non supporta il determinismo ma significherebbe creare noi delle strutture apposite per ricordare la strada e fare in modo che in caso di errori si possa tornare indietro e cercare di percorrere la strada giusta cancellando la strada sbagliata.



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

Automa non deterministico



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

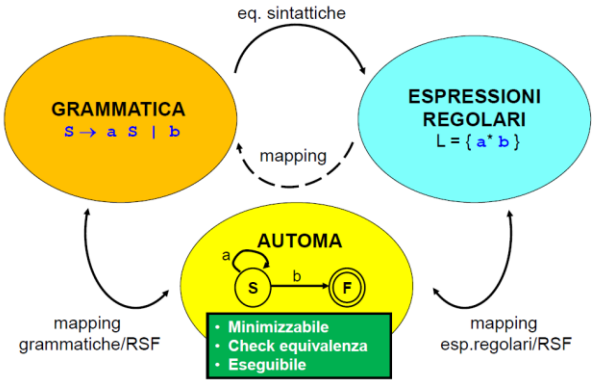
Automa deterministico minimo

Il Prolog

Il Prolog oltre ad essere un linguaggio che supporta il non determinismo è anche un linguaggio dichiarativo e non imperativo come quelli tradizionali che conosciamo. La cosa principale che distingue linguaggio imperativo da dichiarativo è la ricorsione applicata nei linguaggi dichiarativi, è infatti necessario esprimere i comandi imposti in maniera ricorsiva.
Un altro vantaggio del Prolog è che oltre che essere interrogato per riconoscere un linguaggio, può essere interrogato per generare un linguaggio.

I linguaggi che un ASF riconosce sono quelli regolari, ovvero che sono descritti da espressioni regolari. Rendendo deterministico e minimo un automa posso ottenere l'espressione regolare associata e così facendo mi rendo conto che Grammatica, Regular Expression e Automa Riconoscitore sono 3 punti di vista della stessa realtà.

Già avevo visto che dalla grammatica posso ottenere un diagramma dell'automa e viceversa posso passare dal diagramma alla grammatica facendo il giusto mapping. Già avevamo visto anche, nel capitolo precedente, che è possibile passare dalla grammatica alla rispettiva regular expression e mappando la regular expression posso ri-ottenere la grammatica. Ora come ultimo passo mi sono reso conto che col dovuto mapping posso passare dal diagramma dell'automa alla regular expression e viceversa.



Capitolo 4 – Riconoscitori per grammatiche context-free

Per riconoscere linguaggi di tipo 2 abbiamo un problema maggiore, ovvero la grandezza della memoria che potrebbe dover essere illimitata. Per far fronte a questo problema facciamo affidamento ad un automa RSF come quello per riconoscere linguaggi di tipo 3, ma ci appoggiamo ad una struttura che è lo **stack**: chiameremo questo automa **PDA (Push-Down Automaton)**. Lo stack ci dà la possibilità di appoggiarci e avere memoria teoricamente infinita, quindi possiamo valutare anche vincoli meno restrittivi, senza un limite superiore. Questo riconoscitore è realizzato da una sestupla <A, S, S_0, sfn, Z, Z_0> (alfabeto, stati, stato iniziale, state function, alfabeto dei simboli interni, simbolo iniziale sullo stack) e segue due criteri accettabili dall'automa: il primo è quello dello **stato finale**, lo stesso del tipo 3, che richiede per la terminazione della computazione che ci si porti lo stato dall'inizio alla fine. Il secondo criterio è quello dello **stack vuoto**, ovvero portando lo stato a quello finale è necessario che non si mantengano elementi nello stack, ma si arrivi a 0 elementi per ottenere un linguaggio accettato.

Analizzando i PDA non deterministici (quelli che da uno stato hanno più strade da scegliere a cui passare), vediamo che abbiamo ci sono 2 aspetti per classificarlo:

1. Il PDA è non deterministico solo perché gli si presentano più strade da prendere;
2. Il PDA è non deterministico perché può scegliere se accettare l'input che gli viene dato, oppure non accettarlo (ε-mossa);

Con un PDA non deterministico è possibile riconoscere un qualunque linguaggio context-free il problema è la complessità di elaborazione: infatti spesso ci basterà un PDA deterministico che ha una complessità lineare di elaborazione invece che esponenziale.
È necessario sapere (per creare un PDA efficiente) che l'unione, intersezione e concatenamento di linguaggi deterministici non dà luogo a un linguaggio deterministico, il complemento di un linguaggio deterministico invece è ancora deterministico.
Troviamo che tra PDA non deterministico e deterministico vengono a mancare alcune proprietà.
Per realizzare il PDA deterministico potrebbe essere una buona idea manipolare lo stack esattamente come manipoliamo il PDA, sfruttando il linguaggio di programmazione più adatto.

Metodo di realizzazione del PDA deterministico → **Analisi Ricorsiva Discendente** (Top-Down Recursive-Descendent Parsing)
Ad ogni metasimbolo della grammatica associo una funzione. Devo fare attenzione però a non cablare nel codice il comportamento: cerco così di distaccare motore e grammatica costruendo la **Tabella di Parsing** (simile alle transizioni di un Riconoscitore).

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

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

Ci è immediato scrivere il riconoscitore partendo dalla grammatica, ma l'approccio non è sempre possibile, funziona se non ho mai dubbi su quale regola applicare.

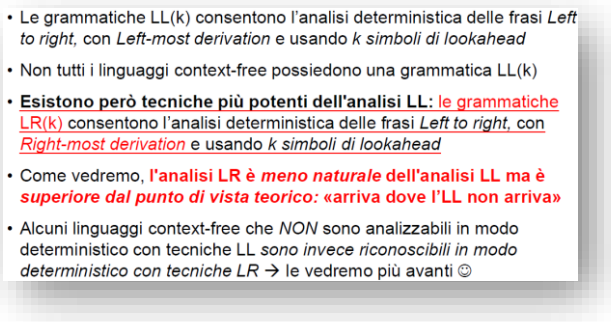
Per rendere deterministica l'analisi top-down è necessario cercare di prevedere quale mossa tra quelle che ci si porranno sarà necessario scartare e quale mossa invece sarà necessario scegliere. Il dubbio sta nello stabilire in che maniera farlo, senza tirare a indovinare: per questo ci vengono in aiuto le grammatiche LL(k) e gli Starter Symbol Set.
Grazie alle **grammatiche LL(k)** è possibile scegliere quale produzione è preferibile tra quelle che vengono poste, infatti la 'k' è il numero di simboli necessari da guardare in avanti per sapere di poter operare in modo deterministico.
Solitamente per le grammatiche LL(k) si opera con la derivazione canonica sinistra e procedendo da sinistra a destra.
Gli **Starter Symbol Set** invece sono i simboli che precedono in particolare alcuni meta-simboli e inserendo un simbolo diverso per ogni produzione è più facile "vedere in avanti".

Per essere sicuri di avere una grammatica LL(1) devo avere gli starter set disgiunti (ovvero tutti diversi) e non devo avere stringhe vuote, infatti la stringa vuota causa problemi e:

1. O la elimino sostituendo;
2. O amplio la nozione di starter set dando la possibilità anche alle stringhe vuote di esserci;

ma mi rendo conto che sostituendo e eliminando le ε-rules posso non avere ancora gli starter set disgiunti.
Nasce da qui il concetto di **blocco annullabile**: stringa che può scomparire e che deve essere prevista anche dove la presenza non appare subito evidente. Infatti visualizzando la Parsing Table mi rendo conto che ci possono essere più celle con la stessa regola, quindi la grammatica non è LL(1) neanche avendo cancellato le ε-rules.

La soluzione alternativa e definitiva per scoprire se una grammatica è LL(1) sarà **ampliare la nozione di starter set**, metto in conto la presenza di stringhe vuote e definisco il **Director Symbol Set** ovvero l'unione tra starter symbol set e following symbol set.
Il Director Symbol Set ci permette di riformulare la condizione per avere una grammatica LL(1): che i directory symbol set siano disgiunti.
Spesso è possibile infatti trovare una grammatica equivalente LL(1) applicando l'eliminazione della ricorsione sinistra e il raccoglimento a fattor comune.
Ma non posso applicare sempre la ricorsione sinistra perché può cambiarmi la semantica della grammatica e a volte anche il raccoglimento non risolve i problemi.

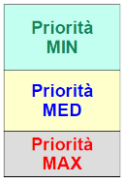

- Le grammatiche LL(k) consentono l'analisi deterministica delle frasi *Left to right*, con *Left-most derivation* e usando *k simboli di lookahead*
- Non tutti i linguaggi context-free possiedono una grammatica LL(k)
- **Esistono però tecniche più potenti dell'analisi LL:** le grammatiche LR(k) consentono l'analisi deterministica delle frasi *Left to right*, con *Right-most derivation* e usando *k simboli di lookahead*
- Come vedremo, **l'analisi LR è *meno naturale* dell'analisi LL ma è *superiore dal punto di vista teorico:* «arriva dove l'LL non arriva»**
- Alcuni linguaggi context-free che *NON* sono analizzabili in modo deterministico con tecniche LL *sono invece riconoscibili in modo deterministico con tecniche LR* → le vedremo più avanti ☺

Capitolo 5 – Dai Riconoscitori agli Interpreti

Finora abbiamo utilizzato riconoscitori, che rispondono solamente sì o no ad una stringa datagli in ingresso esplicitando se la stringa appartiene al linguaggio desiderato. Ma è sicuramente meglio passare ad interpreti e compilatori, che ci permettono anche di costruire immediatamente o successivamente una struttura dati adatta.
Un interprete solitamente è formato da **scanner** (analisi lessicale), fornisce solo le piccole parole aggregate, e **parser** (analisi sintattica e semantica), valuta la correttezza della sequenza di parole.

Lo scanner solitamente categorizza le parole in più battute in modo da non essere rigido riconoscendo subito il linguaggio in maniera specifica ma prima in maniera lieve poi in maniera sempre più precisa e rigorosa identifica le parole chiave e i simboli particolari del linguaggio. Per far questo consulta di volta in volta **tabelle** (delle parole chiave e dei simboli speciali) che gli fanno riconoscere meglio il linguaggio.
Per quanto riguarda il parser invece so di avere un interprete se in uscita ho una valutazione immediata del dominio, mentre ho un albero se l'obiettivo è avere un **compilatore** o un interprete a più fasi.

Bisogna porre attenzione a non creare grammatiche ambigue e delle due sarebbe addirittura meglio cablare nella grammatica gli elementi e le scelte cruciali.
Si introduce, per evitare ambiguità, sintetizzando per prime le entità degli strati più bassi.
Devo fare attenzione a non avere ricorsione a sinistra poiché è incompatibile con l'analisi ricorsiva discendente (tecnica principale di costruzione dei parser).
Può capitare che le soluzioni alternative alla ricorsività a sinistra non siano logicamente accettabili e scomode da usare. Per trovare un parser corretto e comodo so essere un PDA ma voglio riuscire in un qualche modo a rendere LL(1) la grammatica.



Il mio obiettivo è infatti rendere la grammatica LL(1) e mi accorgo di poterlo fare esprimendo senza ricorsioni i sotto-linguaggi che avevo generati, nonostante le ripetizioni, grazie alla notazione **EBNF** che mi fa rimuovere la ricorsione esplicita.
Posso così applicare l'analisi ricorsiva discendente senza problemi (senza far esplodere lo stack).
Applicandola ho che termino la funzione quando finisco i caratteri in input o incontro un simbolo sconosciuto al linguaggio previsto. In uscita ho o un boolean o un oggetto.

Cerchiamo di definire come costruire il parser e quale componente software ci può aiutare: si tenta con java di creare un tokenizer e da lì sia il parser che lo scanner.
Ora facendo il punto della situazione sappiamo trovare la grammatica adatta (grazie all'EBNF) e da lì sappiamo scrivere il puro riconoscitore. Ci manca invece lo scrivere il parser completo e la specifica semantica che il parser dovrà seguire.
Per specificare la semantica ci può venire in aiuto la Funzione di Interpretazione, che intende precisare senza ambiguità il significato di ogni frase del linguaggio.

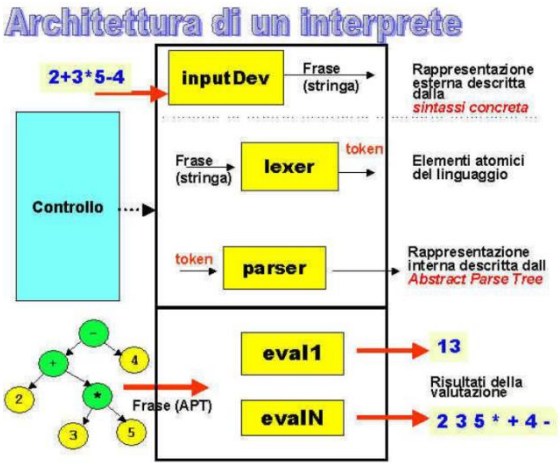
Per costruire la semantica dell'interprete e verificarne la correttezza possiamo pensare di seguire pari pari la sintassi. La semantica che usiamo viene chiamata **Semantica Denotazionale**, ad ogni input si cerca di far corrispondere un output e si specifica un dato comportamento.
L'interprete da noi costruito per Exp, Term e Factor è in grado di analizzare il sotto-linguaggio di pertinenza dividendosi i compiti: un compito specifico ad ogni parser.

Quindi quello fatto finora è la costruzione del parser, ora ci manca la generazione dell'albero sintattico che ci permette di rappresentare le frasi corrette. L'albero più conveniente da usare è l'AST (Abstract Syntax Tree), che è più compatto e contiene solo i nodi indispensabili e non da una rappresentazione ridondante del linguaggio. Infatti si possono escludere le foglie dell'albero che hanno 1 solo figlio, oppure le foglie che non hanno niente di semanticamente significativo, ecc…
La soluzione adottata è spostare dal figlio al padre il simbolo dell'operatore.

Grazie all'AST non c'è ambiguità nell'ordine di esecuzione delle operazioni, il che è positivo. Quindi tutto ciò che dobbiamo fare è cercare di estendere il parser facendogli generare l'AST. Utilizziamo una sintassi astratta per descrivere come sarà fatto l'albero descrivendo i nodi necessari. E seguendo la sintassi si potrà realizzare ad esempio con java la struttura concreta.

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

Il parser che genera un AST non dà più in uscita un int o un boolean, ma un oggetto della classe che stiamo costruendo.
Ottenuto l'AST sappiamo che questo albero non rispecchia più la sintassi del linguaggio, ma solo la sintassi per costruire l'interprete, infatti può essere ambigua e serve solo per fare l'AST.
Dopo aver generato l'albero sintattico ora ci manca solo riuscire a valutarlo, ovvero riuscire a ricomporre la frase originale.



Per valutare l'albero sintattico ci sono sicuramente diverse possibilità: notazione infissa è quella che usiamo di solito (con gli operandi e in mezzo l'operatore), la notazione post-fissa o pre-fissa invece prevedono l'inserimento dell'operatore rispettivamente prima e dopo gli operandi da applicare. Infatti notiamo che spesso la tanto usata notazione infissa crea spesso confusione e richiede l'utilizzo di parentesi.

A seconda di come scegliamo di visitare l'albero chiaramente avremo una notazione diversa utilizzata. In particolare sappiamo che la notazione postfissa è adattissima ad un computer, proprio perché possiamo preparare prima le celle di memoria per i numeri letti, poi applicare l'operazione,

`9-(4-1)`	`- 9 - 4 1`	`9 4 1 - -`
Notazione infissa	Notazione prefissa	Notazione postfissa

esattamente come solitamente si fa in assembler. E se magari non abbiamo troppi registri si può usare uno stack in cui depositare gli operandi e gli operatori.

Con Java Swing è possibile tramite il componente JTree ottenere gli alberi valutarli.
Ciò che ci manca da fare ora è scrivere in Java il componente "valutatore".
Per implementare la funzione di valutazione sarà necessario visitare l'albero e applicare la semantica prevista.

Capitolo 6 – Stili di Interpretazione

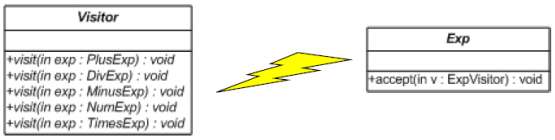
Dobbiamo comporre il valutatore e in particolare scegliere come implementare la funzione di valutazione e usare:

1. Una eval statica (metodologia funzionale);
2. O una eval dell'AST (metodologia Object Oriented);

se dal valutatore poi volessimo ottenere un compilatore basterebbe comporre una riscrittura della frase in un codice intermedio.

Tra compilatore e interprete è questa infatti la differenza: l'**interprete** esegue immediatamente dopo aver valutato la frase, mentre il **compilatore** non esegue mai e restituisce solo una meta-valutazione della frase data in input e si esegue in un secondo momento con un altro comando.
Linguaggi che vengono compilati sono C, C++, Fortran, Pascal; linguaggi invece che vengono interpretati sono Java, Basic, Perl.
La differenza a livello di codice fra compilatore e interprete è che per il compilatore viene creata una funzione **'emit'** che emette il codice macchina corrispondente.

Ci servirebbe e sembrerebbe più adatta come funzione di valutazione la 1° (eval statica), ma non essendo Object Oriented sarà necessario usare il **pattern Visitors** per incapsulare la logica in un opportuno oggetto. La classe base dichiarerà il metodo accept che tutte le classi dovranno implementare.



Capitolo 7 – Multi-Paradigm Programming

Posso ottenere un puro riconoscitore o un interprete completo traducendo la semantica in regole Prolog. In prolog posso scrivere una serie di regole che hanno la forma testa :- corpo (se il corpo è vero allora la testa è vera, se il corpo manca la testa è sempre vera).
Nelle query viene utilizzata l'**unificazione**: una forma di pattern matching che ci permette di utilizzare solo le regole che ci interessano.
Prolog utilizza un suo scanner e un suo parser per interpretare la sua sintassi di regole; dobbiamo cercare di mappare la nostra sintassi su quella del motore Prolog e se riesco tramite le regole a mappare anche la semantica desiderata ho quasi risolto tutti i problemi.
Il problema è che le regole Prolog non contengono semantica ma sono pura sintassi; il significato (semantica) verrà gestito a parte.

RICONOSCITORE PROLOG

```
fExpr(Term) :- fTerm(Term).
fExpr(Exp+Term) :-
  fExpr(Exp), fTerm(Term).
fExpr(Exp-Term) :-
  fExpr(Exp), fTerm(Term).
fTerm(Factor):- fFactor(Factor).
fTerm(Term*Factor) :-
  fTerm(Term), fFactor(Factor).
fTerm(Term/Factor) :-
  fTerm(Term), fFactor(Factor).
fFactor([Exp]) :- fExpr(Exp).
fFactor(Num) :- number(Num).
```

SEMANTICA DENOTAZIONALE

```
fExpr(t) = fTerm(t)
fExpr(e+t) = fExpr(e) + fTerm(t)

fExpr(e-t) = fExpr(e) - fTerm(t)

fTerm(factor) = fFactor(f)
fTerm(t*f) = fTerm(t) * fFactor(f)

fTerm(t/f) = fTerm(t) / fFactor(f)

fFactor((e)) = fExpr(e)
fFactor(num) = valueof(num)
```

Per ottenere un valutatore è necessario restituire un parametro di uscita e ricavare i valori dei due argomenti perciò dovremo sia estendere la testa che il corpo delle regole di Prolog. Bisogna poi considerare che si può delegare al parser Prolog il controllo della sintassi dei numeri dati in input, però è necessario accettare il fatto che il Prolog non ha conoscenza dei numeri naturali ma conosce **solo i numeri reali**!
Predicato speciale **is** per calcolare i risultati delle espressioni, se no crea solo una pura valutazione sintattica.
Notiamo però che non ci conviene usare Prolog per tutto: sarebbe comodo usare java per le stringhe e per gestire l'I/O, però è facile scrivere l'interprete in Prolog. Ci viene così in aiuto la possibilità di utilizzare un'applicazione multi-paradigma, ossia uno metodo risolutivo che ci permetta di utilizzare anche più linguaggi di programmazione per ogni azione che dobbiamo svolgere, essendo così più efficienti (tuProlog).

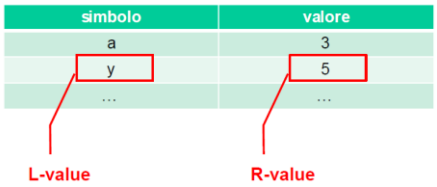
La soluzione adottata sarà quindi avere Java per:

1. visualizzare una finestra di dialogo in cui verrà inserita la funzione da valutare;
2. sostituire le parentesi tonde con le quadre;
3. creare il motore Prolog per la valutazione e interrogarlo;

Faremo quindi poi l'interpretazione e la valutazione in Prolog.
Nuovamente in Java recupereremo il risultato della valutazione e lo mostreremo all'utente.
Grazie ad applicazioni ibride scritte in Java+Prolog riusciamo a risolvere problemi anche molto complessi: ad esempio il "calcolo simbolico delle derivate", che in Java non sapremmo neanche come iniziare a scriverlo, in Prolog è molto più semplice trascrivere le regole e attuarle.

Capitolo 8 – L'Interprete esteso: assegnamenti, ambienti e sequenze

L'operatore di assegnamento '=' in alcuni linguaggi come il Pascal esplicita a quale direzione sia riferito, poiché il significato che ha a destra e a sinistra è differente. Per questo motivo dobbiamo distinguere per la variabile qual è il suo valore a seconda della sua posizione: a destra o sinistra dell'operatore di assegnamento.
l-value è il nome della variabile e indica la cella di memoria associata;
r-value invece indica il contenuto della variabile, ovvero il valore associatole;
Sempre rispetto a questo è da considerare la scelta del particolare linguaggio di programmazione: è possibile che l'assegnamento del valore alla variabile sia distruttivo (Linguaggio imperativo), o non distruttivo (Linguaggio logico).
Introduciamo il concetto di **Environment** →
l'environment è la tabella in figura, la coppia simbolo-valore.



L'environment viene sempre modificato quando c'è l'assegnamento di una variabile. C'è la possibilità di inserire un'istanza se non è già presente il simbolo nell'environment, oppure si può scegliere se sostituire il valore associato al simbolo o non sostituirlo e invece restituire un errore in output.
L'environment può poi essere:

1. **Locale** → se il tempo di vita non è quello del programma ma è associato alla durata di una struttura run-time come una funzione;
2. **Globale** → se le coppie hanno tempo di vita con l'intero programma;

Quindi per introdurre l'assegnamento in un linguaggio bisogna stabilire come si vuol agire in situazioni critiche come assegnamento distruttivo/non distruttivo, assegnamento ad istruzione/espressione, sintassi dei nomi delle variabili.
L'assegnamento può essere ad istruzione nel caso standard, da eseguire e concludere subito, oppure a espressione, indispensabile se si vuole ottenere assegnamento multiplo.
L'assegnamento per essere multiplo deve essere associativo a destra, ovvero l'assegnamento fluisce da destra verso sinistra.

Quindi ricapitolando tutto ciò che ci serve per estendere il nostro interprete è l'espressione di assegnamento e la variabile nelle sue due interpretazioni l-value e r-value.
Per l'AST abbiamo 3 nuovi tipi di nodi e quindi 3 nuove classi Java:

1. Operazione di assegnamento ('=');
2. Nodo L-value;
3. Nodo R-value;

(implementazione in java dell'estensione dell'interprete seguendo il pattern Visitor)

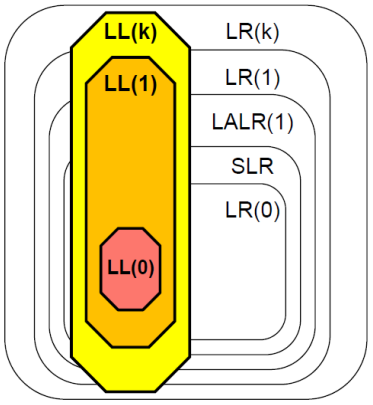
Aggiunta delle espressioni-sequenza
Le espressioni-sequenza sono sequenze di espressioni separate da virgola; ci fanno arrivare sempre più vicini ad un vero linguaggio di programmazione, in cui prima si inizializzano le variabili poi si utilizzano per computare.

Sintassi astratta:
EXP ::= ASSIGN , EXP // *SeqExp*

Aggiunte le espressioni sequenza ora il parser è quasi completo: è in grado infatti di valutare espressioni su interi, variabili e sequenze. Ci mancano invece cose abbastanza semplici da aggiungere: espressioni relazionali (maggiore, minore, diverso) e logiche (and, or, not); le strutture di controllo (come for, while, if); le funzioni e i tipi.

Capitolo 14 – Riconoscitori LR(0)

Nel parsing sappiamo di poter usare l'analisi LL, che è un'analisi semplice utilizzabile praticamente in qualsiasi contesto. Consiste nella costruzione top-down dell'albero, ovvero partendo dal simbolo S iniziale della grammatica e cercando di costruire l'albero coprendo la frase; deve sapere da che parte andare avendo visto solo i primi *k* simboli della parte destra della produzione.
L'analisi che si contrappone a quella LL è la LR che opera invece analizzando bottom-up, ovvero partendo dalla frase da riconoscere e riducendola allo scopo S.



La differenza tra le due analisi è che la LL risulta di semplice applicazione ma proprio per questo è anche molto meno efficiente della LR. Dall'altro lato invece la LR è la macchina per il parsing più potente a nostra disposizione per le grammatiche context-free, ma è anche molto complessa. E proprio perché anche il caso più semplice di LR è molto complesso, sono state sviluppate tecniche che riducono la complessità: SLR (Simple LR), LALR (Look-Ahead LR). Ricordiamo però che ogni grammatica LL(k) è anche LR(k).
La difficoltà del parser LR è che ad ogni passo deve scegliere, in base al contesto corrente, se:

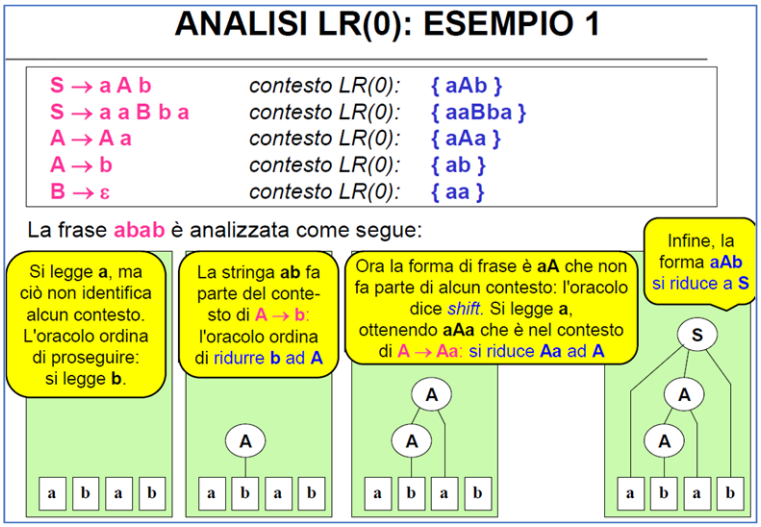
1. Proseguire la lettura dell'input (Shift);
2. O costruire un pezzo di albero senza leggere input (Reduce);

quale azione fare ce lo può dire solo un oracolo, noi conserveremo tutto l'input in uno stack e governeremo il tutto con un controller.
Ciò che faremo quindi sarà una derivazione canonica destra utilizzata al contrario: dalla frase allo scopo. **Derivazione canonica destra a rovescio: aab → Aab → Ab → AB → S**

Per iniziare l'analisi LR(0) dovremo:

1. Calcolare il context di ogni produzione;
2. Verificare eventuali collisioni fra contesti relativi a produzioni diverse;
3. Se non ci sono collisioni posso usare i contesti LR(0) per guidare l'analisi;

Andando ad analizzare come viene realizzato il parsing LR vediamo che l'oracolo non è altro che un riconoscitore di contesto.
Volendo analizzare la maniera in cui opera il parser LR cerchiamo di partire dal caso più facile in assoluto, che ci permetterà di capire le basi della tecnica → LR(0)
Nella realtà questo caso non capita mai anche perché significa poter scegliere la mossa giusta senza dover mai guardare il prossimo simbolo in input, quindi significa che non ci sono collisioni per le diverse produzioni.



La grammatica dei contesti LR(0) è sempre regolare a sinistra, perciò il riconoscimento del contesto può essere svolto da un ASF (Automa a Stati Finiti). Quindi il nostro oracolo che prevedeva che mossa fare è un semplice ASF.
Andando a osservare la definizione del contesto LR(0) vediamo che è complessa e definita formalmente come:

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

Ci sono casi in cui lo scopo 'S' è riutilizzato anche nella parte destra delle produzioni. Noi, visto l'obiettivo di dover ottenere 'S', potremmo essere fuorviati e pensare di aver concluso l'elaborazione. In realtà per evitare di far confusione quindi useremo una **nuova produzione di top-level**: $Z \to S$ e Z è il nuovo scopo.

Per calcolare i contesti LR(0) possiamo applicare un algoritmo che ci permetterà di ottenere il disegno dell'automa ausiliario e sfruttarlo (quell'automa è l'oracolo che decide come operare), oppure otterremo lo stesso risultato andando a sfruttare un procedimento un po' più semplice e operativo, costruendo la tabella di parsing.

Algoritmo

$Z \to S$
$S \to a\,S\,A\,B \mid B\,A$
$A \to a\,A \mid B$
$B \to b$

Utilizzando l'algoritmo, partendo da questa grammatica, ciò che dobbiamo fare è ottenere i contesti sinistri, e lo faccio applicando solo i due postulati:

1) $leftctx(Z) = \{ \varepsilon \}$
2) $B \to \gamma A \delta \quad \Rightarrow \quad leftctx(A) \supseteq leftctx(B) \bullet \{\gamma\}$

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

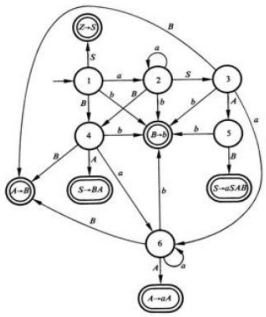
Ottengo così per la prima produzione (Z→S):

$$leftctx(S) \supseteq leftctx(Z) \bullet \{ \varepsilon \} \quad = \quad leftctx(Z)$$

Per la produzione S→aSAB|BA ottengo:

leftctx(S) ⊇ leftctx(S) • { a } (S → a S δ)
leftctx(A) ⊇ leftctx(S) • { aS } (S → a S A δ)
leftctx(B) ⊇ leftctx(S) • { aSA } (S → a S A B)
leftctx(B) ⊇ leftctx(S) • { ε } (S → B δ)
leftctx(A) ⊇ leftctx(S) • { B } (S → B A)

E così via, ottengo prima tutti i contesti sinistri, poi la grammatica regolare ausiliaria, poi le regular expression e infine i contesti LR(0) (non contesti sinistri!).
Grazie ai contesti LR(0) posso costruire il riconoscitore a stati finiti: se non ho collisioni vado liscio.

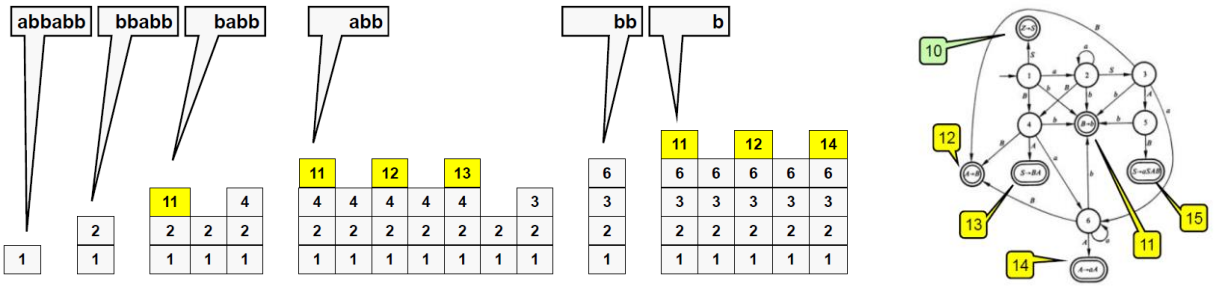


AUTOMA A STATI AUSILIARIO
Arriviamo così al disegno dell'automa, immediato e utile per sapere quale decisione l'oracolo dovrà prendere.
Quindi a seconda di quale lettera viene data in input l'oracolo seguendo l'automa dell'immagine sceglierà come ridurre.

Volendo è inoltre possibile osservare l'evoluzione temporale dello stack per la frase in input → abbabb

							b	B	A		b	B		
			b	B	A		a	a	a	A	A	A		
	b	B	B	B	B	S	S	S	S	S	S	S		
a	a	a	a	a	a	a	a	a	a	a	a	a	S	Z

PARSING OTTIMIZZATO
È possibile **ottimizzare** però la soluzione, poiché ad ogni step di riduzione si riparte a scansionare daccapo la sequenza di caratteri. È un processo un po' verboso che si riesce a rendere più veloce grazie ad un **secondo stack** in appoggio al primo: avremo così il classico stack di input, poi uno stack degli stati, in cui verrà identificato uno stato per ogni combinazione in ingresso rilevante, appunto per non ripartire sempre daccapo.



Arrivati a questo punto siamo in grado di specificare che: **una grammatica è analizzabile mediante analisi LR(0)** (oppure si dice anche che una grammatica è LR(0)), **se** ogni stato di riduzione dell'automa ausiliario è etichettato da una produzione unica e non ha archi di uscita etichettati da terminali.

Procedimento Operativo

In contrapposizione al procedimento algoritmico, un po' complicato e lungo per arrivare a costruire l'automa caratteristico, c'è il procedimento operativo: più semplice, più pratico e sintetico. Partiremo dalla produzione di top-level (Z→S) e analizziamo le situazioni che si presentano; andremo così a costruire dal basso l'automa. Con la notazione del punto (.) si terrà conto del cursore, ovvero "dove siamo"; col simbolo $ invece segnaleremo il terminatore della frase. Arriveremo così a costruire lo stesso automa di prima e con due termini diversi andremo a chiamare l'operazione di **shift** → **goto**; e l'operazione di **reduce** → **accept**.

Si costruisce ora la tabella di parsing:

	a	b	$	S	A	B
1	s2	s11		g10		g4
2	s2	s11		g3		g4
3	s6	s11			g5	g12
4	s6	s11			g13	g12
5		s11				g15
6	s6				g14	g12
10			a			
11	r1	r1	r1			
12	r2	r2	r2			
13	r3	r3	r3			
14	r4	r4	r4			
15	r5	r5	r5			

Legenda

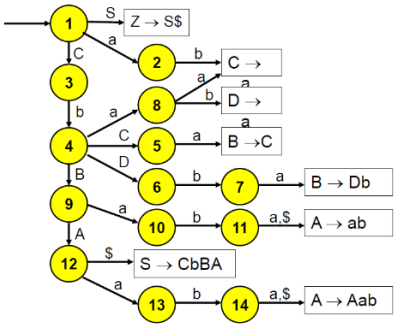
stati di SHIFT	stati di GOTO
stati di RIDUZIONE	stato di ACCEPT

Capitolo 15 – Riconoscitori LR(1)

Per gli usi reali i riconoscitori LR(0) sono troppo semplicistici e non hanno particolare interesse pratico. È così importante considerare i riconoscitori LR(k): riconoscitori analoghi agli LR(0), ma le riduzioni sono ritardate di 'k' simboli e le definizioni di contesto, contesto sinistro e procedimento operativo vengono estese considerando i 'k' simboli successivi e con esse anche i 2 postulati che fanno da pilastro per la formazione dei contesti.

Definisco così il contesto→	$LR(k)ctx(A \to \alpha) = \{ \gamma \mid \gamma = \beta\alpha u, Z \Rightarrow \beta A u w \Rightarrow \beta\alpha u w, w \in VT^*, u \in VT^k \}$
Definisco così il contesto sinistro→	$leftctx(A, u) = \{ \beta \mid Z \overset{*}{\Rightarrow} \beta A u w, w \in VT^*, u \in VT^k \}$
Definisco il primo postulato→	$leftctx(Z, \varepsilon) = \{ \varepsilon \}$
Definisco il secondo postulato→	Se $P \to \gamma Q \delta$ allora $leftctx(Q, u) \supseteq leftctx(P, v) \bullet \{\gamma\}$

Per costruire l'automa caratteristico procedo anche qui idealmente come nel caso LR(0) calcolando le espressioni regolari e sfruttandole. Tuttavia, come già specificato anche prima, qui si complicano e allungano i calcoli. Perciò spesso l'approccio LR(k) non sarà completo ma approssimato, con versioni semplificate.



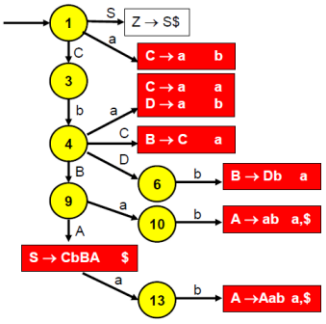
L'automa caratteristico LR(1) è deterministico come nella grammatica LR(0), ovvero ha 1 unico stato futuro possibile per qualsiasi stato e non deve scegliere fra 2 o più stati futuri possibili.

Procedimento operativo

Il procedimento è ancora una volta analogo alla grammatica LR(0), ma la differenza sta nel fatto di tener conto anche del simbolo successivo (**lookahead set**). Ora a fianco ad ogni regola si specificano anche i possibili simboli che validano l'azione.
Evidentemente i calcoli si allungano, perché per ogni regola dovrò calcolare anche i lookahead symbol.
Osservando il procedimento potremo notare che grazie al lookahead set potremo eliminare i conflitti che hanno riduzioni distinguibili e ottimizzeremo così il diagramma dell'automa.

Z → . S$	?
S → . aSAB | . BA	a,=
B → . b	$

ANYTHING è indicato solitamente con ? o *



Ora siamo così riusciti a ricavare l'automa di una grammatica LR(1) capendone un po' i meccanismi, ma sappiamo per esperienza che se per una grammatica così semplice come quella valutata già i problemi aumentano, per un linguaggio vero non sarà possibile utilizzare la grammatica LR(1) pura per fare il parsing.
Perciò come anticipato, introduciamo versioni semplificate della grammatica LR(1): l'idea sarà quella di **ridurre gli stati accorpando** quelli sufficientemente simili.

Parser Simple LR

Il contesto della produzione $A \to \alpha$ della grammatica SLR sarà
$SLR(k)ctx(A \to \alpha) = LR(0)ctx(A \to \alpha) \bullet FOLLOW_k(A)$ e sappiamo che il contesto SLR sarà un po' più grande quindi meno ottimizzato e più esposto a potenziali conflitti. Se effettivamente ci sono potenziali conflitti dovremo rinunciare ad usare l'SLR perché comunque l'automa dovrà essere deterministico, ma se i conflitti non ci sono potremo davvero utilizzare la grammatica SLR al posto della LR.
Come nella grammatica LR però, si può ottenere una grammatica SLR con un **procedimento operativo** più semplice di quello dei contesti → prendendo l'automa LR(0) togliamo le riduzioni incompatibili col lookahead set.
Posso infatti passare da una grammatica LR(0) che non è in grado di guardare avanti e applica nella tabella di parsing l'azione di riduzione solo nelle colonne con i simboli terminali, ad una grammatica SLR che invece inserisce la riduzione $A \to \alpha$ solo nelle colonne dei terminali con essa compatibili.

	a	+	$	E	T
1	s4			gF	g2
F			a		
2	r2	s3,r2	r2		
3	s4			g5	g2
4	r3	r3	r3		
5	r1	r1	r1		

Parsing LR(0)

	a	+	$	E	T
1	s4			gF	g2
F			a		
2		s3	r2		
3	s4			g5	g2
4		r3	r3		
5			r1		

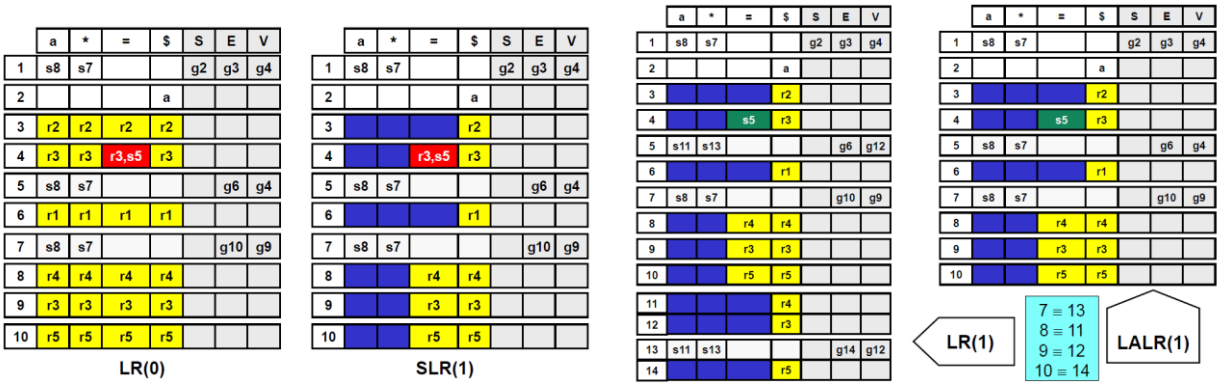
Parsing SLR

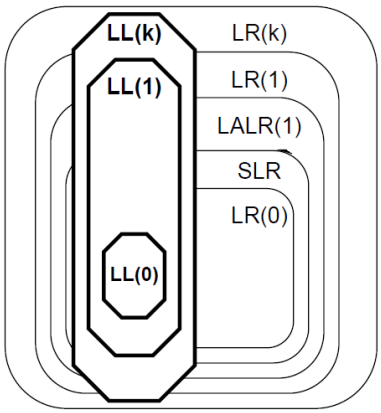
Oltre a questo possibile parsing SLR c'è una variante analoga che però è ricorsiva a sinistra e ha una produzione in più che include l'uso delle parentesi.

Parser Look Ahead LR(1)

Quando non funziona il parser SLR è possibile utilizzare questo approccio alternativo che per semplificare le cose fa collassare gli stati uguali a meno dei lookahead set. Questa soluzione a differenza delle altre è sempre possibile!

Può essere implementato calcolando LR(0) poi aggiungendo i lookahead set





Abbiamo quindi in conclusione che il parser SLR può essere utilizzato per alcune grammatiche LR(k) e diverse grammatiche LL(k) possono essere ricondotte a grammatiche LR(k).

Però abbiamo anche che grammatiche LL(1) non sono SLR(1) e grammatiche LR(1) non sono SLR(k) per nessun valore di k.