Indice

1 — Linguaggi, macchine astratte, teoria della computabilità

Alcune definizioni

- **Algoritmo**: sequenza *finita* di istruzioni che risolve in un *tempo finito* una classe di problemi

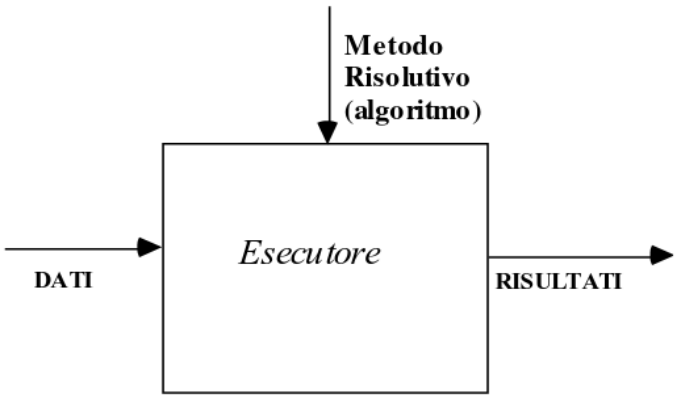
- **Codifica**: descrizione dell'algoritmo tramite un *insieme ordinato di frasi* di un linguaggio di programmazione, che specificano le *azioni* da svolgere

- **Programma**: testo scritto in accordo alla *sintassi* e alla *semantica* di un linguaggio di programmazione

1.0.1 Esecuzione

Un algoritmo esprime la soluzione a un problema, il programma è la formulazione testuale rigorosa in un dato linguaggio; l'esecuzione *ordinata* delle azioni specificate porta a ottenere la soluzione da un insieme di dati in ingresso.

Viene assunta l'esistenza di un **automa esecutore**, una macchina astratta capace di eseguire le azioni dell'algoritmo.



1.1 Macchine astratte

1.1.1 Automa esecutore

Un automa deve poter ricevere dall'esterno una *descrizione* dell'algoritmo, deve quindi essere in grado di *interpretare* un linguaggio, che verrà chiamato **linguaggio macchina**.

Realizzabilità fisica

- le parti che compongono l'automa devono essere in *numero finito*
- ingresso e uscita devono essere denotabili da un *insieme finito* di **simboli**

1.1.2 Gerarchia di macchine astratte

- macchina combinatoria
- automi a stati finiti
- ....
- macchina di Turing

Si individua una gerarchia perché diverse classi di macchina hanno diversa *capacità* di risolvere problemi. Se neanche la macchina più "potente" può risolvere un problema, questo potrebbe essere **non risolubile**.

1.1.3 Macchina base

La macchina base è definita formalmente dalla tripla:

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

dove

- I = insieme finito dei simboli di *ingresso*
- O = insieme finito dei simboli di *uscita*
- $mfn : I \to O$ = funzione di macchina

Limiti

Utilizzare una di queste macchine per risolvere problemi comporta valutare tutte le possibili configurazioni di ingresso.

Esiste inoltre un altro limite, essendo puramente combinatoria, la macchina base non può risolvere problemi che richiedano di ricordare qualcosa dal passato, in quanto privo di memoria interna.

1.1.4 Automa a stati finiti

Nell'automa a stati finiti si introduce il concetto di memoria attraverso l'utilizzo di un numero *finito* di **stati interni**.

Un **automa a stati finiti** è definito dalla quintupla:

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

dove

- I = insieme finito dei simboli di *ingresso*
- O = insieme finito dei simboli di *uscita*
- $mfn : I \times S \to O$ = funzione di macchina
- $sfn : I \times S \to S$ = funzione di **stato**

Limiti

Dato che lo stato funge da memoria interna, le risposte possono essere diverse a parità di dati d'ingresso.

Esistono varie categorie ASF come automi di Mealy o Moore, sincroni o asincroni, minimo numero di stati etc..

Esiste un limite computazionale che rende l'automa inadatto a risolvere problemi che non permette di limitare a priori la lunghezza delle sequenze, dovuto al fatto che la memoria è **finita**.

1.1.5 Macchina di Turing

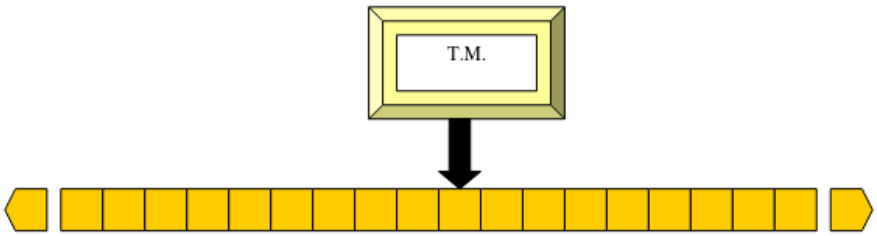
Per ovviare al limite della memoria finita, si introduce un *nastro* di memoria esterno, la **macchina di Turing** è definita dalla quintupla:

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

dove

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

Il deposito dei dati è rappresentato da un natro illimitatamente espandibile



La macchina è dotata di una testina di lettura e scrittura che può:

- leggere un simbolo dal nastro
- scrivere sul nastro il simbolo specificato da $mfn()$
- passare a un nuovo stato interno specificato dalla $sfn()$
- spostarsi sul nastro di una posizione nella direzione indicata da $dfn()$

ipotizzando che quando viene raggiunto lo stato HALT, la macchina si ferma.

Per risolvere un problema con questa macchina, è necessario definire la *rappresentazione dei dati* di partenza (da porre sul nastro) e quelli di uscita (sempre sul nastro), e definire il *comportamento*, ovvero le tre funzioni $mfn()$, $sfn()$ e $dfn()$.

Non è certo che questa macchina arrivi allo stato di HALT.

Tesi di Church-Turing

Esiste una macchina più "potente" della MdT?

La tesi di Church-Turing afferma che

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

1.1.6 Macchine universali

Nella macchina di Turing l'algoritmo è cablato nella macchina.

Si può espandere la MdT alla macchina di Turing **Universale** (UTM), macchina il cui programma è letto dalla macchina direttamente dal nastro.

Questo richiede poter descrivere l'algoritmo richiesto attraverso un *linguaggio* e avere una macchina che lo interpreti.

Le tre operazioni che la UTM effettua sono:

- **fetch**, ricerca delle istruzioni
- **decode**, interpretazione delle istruzioni
- **execute**, esecuzione le istruzioni

UTM vs Macchina di Von Neumann

Macchina di Turing

- leggere/scrivere simbolo da/su nastro
- transitare in un nuovo stato
- spostarsi sul nastro di x posizioni

Macchina di Von Neumann

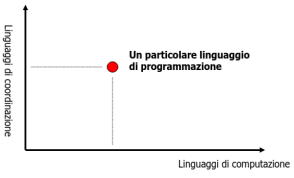
- lettura/scrittura da/su RAM/ROM
- nuova configurazione registri CPU
- scelta cella di memoria su cui operare

La UTM non ha il concetto di "mondo esterno", ne nessuna istruzione di I/O, è puramente computazione senza la dimensione di *interazione*.

Computazione e interazione

Computazione e interazione sono dimensioni *ortogonali*, potenzialmente espressi da due linguaggi distinti:

- linguaggio di **computazione**
- linguaggio di **coordinazione**
  - linguaggio di **comunicazione**



1.2 Teoria della computabilità

Problemi irrisolubili

Secondo la tesi di Church-Turing, non esiste un formalismo più potente della macchina di Turing, quindi se la MdT non riesce a risolvere un problema, quel problema è **irrisolubile**.

Per irrisolubile si intende che la MdT non si ferma, quindi non ritorna un output definito.

Problemi risolubili

Un problema risolubile è un problema la cui soluzione può essere calcolata da una MdT o equivalenti.

1.3 Funzioni

Per valutare la risolubilità di un problema è necessario individuare una **funzione** che lo descrive.

1.3.1 Funzione caratteristica

Come prima cosa si definisce la funziona **caratteristica** di un problema.

> Dato un problema P e detti
>
> - X l'insieme dei suoi dati in ingresso
> - Y l'insieme delle risposte corrette
>
> si dice funzione caratteristica del problema P
>
> $$f_p : X \to Y$$

In questo modo, problema non risolubile equivale a una funzione caratteristica non computabile.

1.3.2 Funzione computabile

Si formalizza ora la classe di funzioni dette **computabili**.

> Una funzione $f : A \to B$ è detta *computabile* se esiste una macchina di Turing

che, data sul nastro una rappresentazione di $x \in A$, dopo un *numero finito* di passi produce sul nastro una rappresentazione di $f(x) \in B$.

1.3.3 Funzioni definibili vs funzioni computabili

É necessario capire se tutte le funzioni sono computabili, o se esistono funzioni definibili ma non computabili.

Funzione computabile su N

Per semplicità, considereremo funzioni sui *numeri naturali*

$$f : N \to N$$

Questa condizione non è limitativa, dato che ogni informazione è necessariamente finita, può quindi essere codificata con una collezione di numeri naturali, la quale a sua volta può essere espressa tramite un *unico* numero naturale (procedimento di Gödel.

Procedimento di Gödel

Data una collezione di numeri naturali, ottenere un unico numero naturale.

- siano $N_1, N_2, \ldots, N_k$ i numeri naturali dati
- siano $P_1, P_2, \ldots, P_k$ i primi k numeri primi

Si definisce R, un nuovo numero naturale così definito

$$R ::= P_1^{N_1} \cdot P_2^{N_2} \cdot \cdots \cdot P_k^{N_k}$$

R rappresenta unicamente la collezione originale $N_1, N_2, \ldots, N_k$.

Poiché che l'insieme $F = \{f : N \to N\}$ delle funzioni sui naturali non è *enumerabile*, e l'insieme delle funzioni computabili è enumerabile (dato che l'insieme dei simboli è finito ogni MdT può essere associata a un numero di Gödel), la gran parte delle funzioni definibili non è computabile.

Interessano però soltanto le funzioni definibili con un linguaggio basato su un alfabeto finito di simboli, sfortunamtamento esistono funzioni definibili con tale alfabeto ma *non computabili*.

1.3.4 Funzioni non computabili

Dire che esistono funzioni definibili ma non computabili, equivale a dire che esistono problemi irrisolubili, basta trovare uno solo di questi problemi.

Problema dell'HALT della macchina di Turing

> Stabilire se una data macchina di Turing T, con un generico ingresso X, si ferma oppure no.

Slide 49-55.

1.3.5 Generabilità e decidibilità

Poiché un linguaggio è un insieme di frasi, è utile indagare la **generabilità** e la **decidibilità** di un insieme.

Si introduce il concetto di *insieme ricorsivamente numerabile*, per decidere che l'insieme sia effettivamente generabile.

Insieme ricorsivamente numerabile

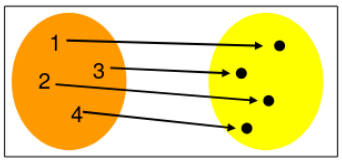
Data la definizione di un insieme numerabile, ovvero un insieme i cui elementi possono essere "contati" cioè che possiede una funzione biettiva:

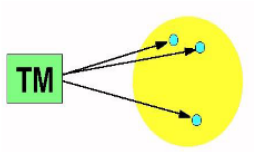
$$f : N \to S$$

che mette in corrispondenza i numeri naturali con gli elementi del sistema.

Si passa alla definizione di insieme **ricorsivamente numerabile** (*semi-decidibile*).

> Un insieme è ricorsivamente numerabile se la funzione biettiva $f : N \to S$ è *computabile*.





Tuttavia, il fatto che l'insieme possa essere costruito, non significa che si possa decidere se un certo elemeto vi appertiene o no.

Decidibilità

In generale, da un insieme ricorsivamente numerabile, potrebbe non essere trovabile un elemento, in questo caso la MdT può entrare in loop:

Un inisieme è **semi-decidibile** se è possibile stabilire se un elemento appartiene a un insieme ma non se un elemento *non appartiene*.

1.3.6 Insiemi decidibili

Occore un concetto più potente della semi-decidibilità.

Un insieme S è **decidibile** (o ricorsivo) se la sua funzione caratteristia è computabile.

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

ovvero se esiste una macchina di Turing capace di rispondere sì o no, senza mai entrare in un *ciclo infinito*, alla domanda 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

- S
- il suo complemento $N - S$

sono **semi-decidibili**.

2 — Linguaggi e grammatiche

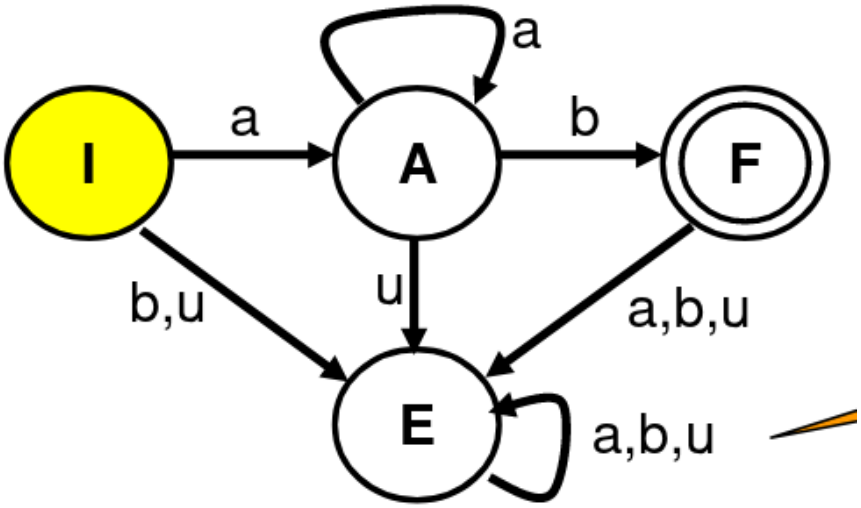
empty

3 — Riconoscitori a stati finiti

...

3.1 Dai riconoscitori ai generatori

Figura 3.1: Automa



Descrivendo a parole le transizioni utili dell'automa:

- nello stato I l'automa può accettare:
  - il simbolo a e portarsi nello stato A
- nello stato A l'automa può accettare:
  - il simbolo b e poi fermarsi
  - il simbolo a e riportarsi nello stato A stesso
- nello stato finale F l'automa puà accettare:
  - nessun simbolo

Se si sostituisce la parola accettare alla parola generare, si nota che l'automa può essere considerato come generatore.

Si può definire un **mapping** tra:

- stati $\longleftrightarrow$ simboli non terminali

- transizioni $\longleftrightarrow$ produzioni
- scopo $\longleftrightarrow$ uno stato particolare

É possibile automatizzare la costruzione di un RSF a partire dalla grammatica o viceversa.

Grammatica regolare a destra:

- scopo = stato iniziale: I
- stato finale: F
  - $S \to a\ A$
  - $A \to a\ A \mid b$

Grammatica regolare a sinistra:

- scopo = stato finale: F
- stato inziale: I
  - $S \to A\ b$
  - $A \to A\ a \mid a$

Lo stato finale F non si considera perché non ha archi uscenti.
Come arrivare a queste grammatiche?

3.1.1 Mapping RSF $\longleftrightarrow$ grammatica

Si immagini un osservatore che, stando in ogni stato, guardi da ogni stato:

- dove si va
  - la freccia col **simbolo terminale**
  - lo stato **successivo**
- da dove viene
  - lo stato **precendente**
  - la freccia col **simbolo terminale**

Mapping tra automa riconoscitore e grammatica

Tra la grammatica e il riconoscitori si riconoscono le seguenti corrispondenze:

- **stati** dell'automa $\longleftrightarrow$ **metasimboli** della grammatica
- **transizioni** dell'automa $\longleftrightarrow$ **produzioni** della grammatica
- **uno stato** dell'automa $\longleftrightarrow$ **scopo** della grammatica

Se la grammatica è regolare a destra si ottiene un automa top-down.
Se la grammatica è regolare a sinistra si ottiene un automa bottom-up.
In modo analogo si ottengono grammatiche destra/sinistra interpretando un RSF in modo top-down/bottom-up
Ad esempio:

In questo caso gli stati iniziale e finale sono anche stati di transito.

(a) Traduzione grammatica

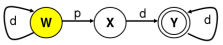
ESEMPIO di RSF

Sia:

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

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

(b) Riconoscitore



Per passare dall'automa alla grammatica si analizzano i collagamenti degli stati e si ricavano le trasformazioni:

$$W \to d\ W \mid p\ X \qquad Y \to X\ d \mid Y\ d$$
$$X \to d \mid d\ Y \qquad X \to p \mid W\ p$$
$$Y \to d \mid d\ Y \qquad W \to d \mid W\ d$$

3.1.2 Riconoscitori top down

Derivazione top-down

Si parte dallo scopo della grammatica e si tenta di coprire la frase data tramite produzioni successive.

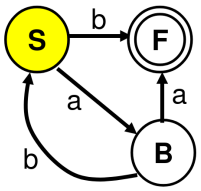
Data una grammatica regolare lineare a destra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato iniziale** lo scopo S
- per ogni regola del tipo $X \to x\ Y$, l'automa con ingresso x passa dallo stato X allo stato Y
- per ogni regola del tipo $X \to x$, l'automa con ingresso x passa dallo stato X allo stato finale F

Esempio

Sia G una grammatica lineare a destra caratterizzata dalle seguenti produzioni:

- $S \to a\ B \mid b$
- $B \to b\ S \mid a$



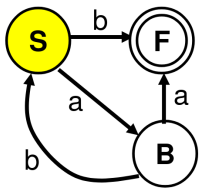
3.1.3 Riconoscitori bottom up

Data una grammatica regolare lineare a sinistra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato finale** lo scopo S
- per ogni regola del tipo $X \rightarrow Y\ x$, l'automa con ingresso x passa dallo stato Y allo stato X
- per ogni regola del tipo $X \rightarrow x$, l'automa con ingresso x passa dallo stato iniziale I allo stato X

Esempio
Sia G una grammatica lineare a sinistra caratterizzata dalle seguenti produzioni:

- $S \rightarrow B\ a \mid b$
- $B \rightarrow S\ b \mid a$

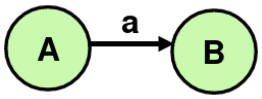


3.1.4 Dall'automa alle grammatiche

Dato un automa riconoscitore, se ne possono trarre sia una grammatica regolare a destra (interpretazione top-down) che una grammatica regolare a sinistra (interpretazione bottom-up).

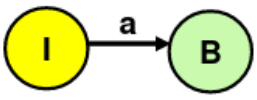
Caso generico

- top-down: $A \rightarrow a\ B$
- bottom-up: $A\ a \leftarrow B$



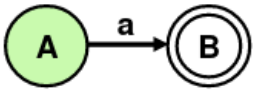
Caso iniziale

- top-down: $S \rightarrow a\ B$
- bottom-up: $I\ a \leftarrow B$

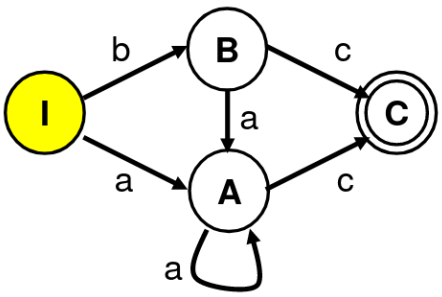


Caso finale

- top-down: $A \rightarrow a\ F$
- bottom-up: $A\ a \leftarrow S$



Esempio



Analisi top-down (C finale omesso):

- $I \rightarrow b\ B$
- $I \rightarrow a\ A$
- $B \rightarrow a\ A$
- $A \rightarrow a\ A$
- $B \rightarrow c$
- $A \rightarrow c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow S$
- $A\ c \leftarrow S$

Grammatica G1 regolare a destra:

- $I \rightarrow b\ B \mid a\ A$
- $B \rightarrow a\ A \mid c$
- $A \rightarrow a\ A \mid c$

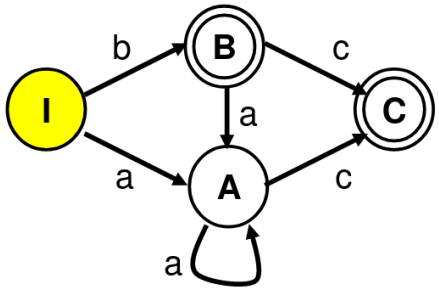
Grammatica G2 regolare a sinistra:

- $S \rightarrow B\ c \mid A\ c$
- $A \rightarrow B\ a \mid A\ a \mid a$
- $B \rightarrow b$

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

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

Esempio multipli stati finali



Analisi top-down (regola in più sulla I):

- $I \to b\ B \mid b$
- $I \to a\ A$
- $B \to a\ A$
- $A \to a\ A$
- $B \to c$
- $A \to c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow C$
- $A\ c \leftarrow C$

L'analisi bottom-up deve scegliere quale scopo adottare, se B o C. Per risolvere questo problema si creano due grammatiche in cui si assumono a turno $C \equiv S$ e $B \equiv S$.

*Grammatica G2' (assume **C ≡ S**)*:
S → B c | A c
A → B a | A a | a
B → b
L(*G2'*) = (**b** + **a**) **a*** **c**

*Grammatica G2" (assume **B ≡ S**)*:
b ← S
le altre regole sono inutili, essendo i loro metasimboli irraggiungibili !
L(*G2"*) = **b**

Caso generale

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

- si assume come (sotto)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come unione dei vari sotto-linguaggi, definendo lo scopo globale come $S \to S_1 \mid S_2 \mid \cdots \mid S_n$

4 — Riconoscitori con PDA

Un RSF non può riconoscere un linguaggio di tipo 2, ha un limite intrinseco alla capacità di memorizzazione: non riesce a riconoscere frasi che richiedano di memorizzare una parte di lunghezza non nota a priori.

Esempio: bilanciamento delle parentesi $L = ({}^n c)^n, n \geq 0$, $G = S \to (S) \mid c$

In questo linguaggio il prefisso $({}^n$ non ha lunghezza limitabile a priori.

4.1 Push-Down Automaton (PDA)

Un PDA è un RSF con aggiunto uno stack, con stack non ci si riferisce a una struttura dati fisica ma a un suo modello astratto, ovvero una sequenza di simboli, definito in modo tale che si possa operare soltanto su quello in "cima".

Il PDA legge un simbolo d'ingresso e transita in un nuovo stato, in più a ogni passo altera lo stack, producendo una nuova configurazione.

Un PDA può prevedere ε-mosse, ovvero transizioni spontanee che manipolano lo stack senza consumare simboli in ingresso.

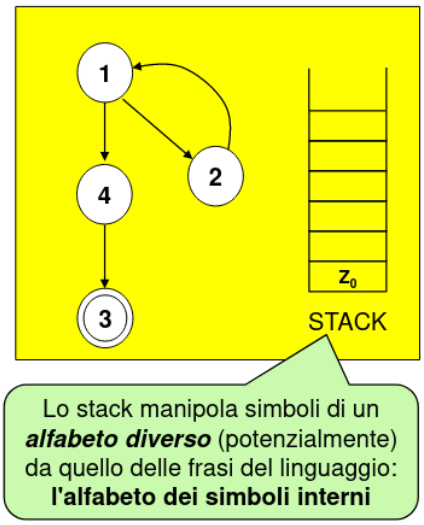
Un PDA è una sestupla:

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

dove:

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

Figura 4.1: Struttura PDA



Il linguaggio accettatto da un PDA è definibile in 2 modi equivalenti:

- **Criterio dello stato finale**: il linguaggio accettato è l'insieme di tutte le stringhe di ingtesso che portano il PDA in uno degli stati finali.
- **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*.

La funzione sfn, dati:

- un *simbolo in ingresso* $a \in A$
- lo *stato attuale* $s \in S$
- il simbolo interno attualmente al *top dello stack* $z \in Z$

opera come segue:

- **consuma** il simbolo di ingresso a
- effettua il **POP** dello dallo stack, prelevando il simbolo z
- porta l'automa in stato futuro $s' = sfn(a, s, z)_{\prod S}$
- effettua una **PUSH** sullo stack di zero o più simboli interni $z' \in Z$
  $z' = sfn(a, s, z)_{\prod S}$

Si consideri il linguaggio generato da:

- $A = \{0, 1, c\}$

- $P = \{S \to 0\ S\ 0 \mid 1\ S\ 1 \mid c\}$
- $L = \{\texttt{word}\ c\ \texttt{word}^R\}$

Definiamo il PDA come segue:

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

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

4.1.1 PDA non deterministici

Anche un PDA può essere non deterministico: in tal caso, la funzione sfn produce insiemi di elementi W (W sottoinsieme finito di $S \times Z^*$).

Ad esempio, il PDA tale che $sfn(Q_0, a, Z) = \{(Q_1, Z_1)(\cdots)(Q_k, Z_k)\}$ è **non deterministico** in quanto l'automa, nello stato Q_0, con simbolo interno in cima allo stack Z e ingresso a ha più di uno stato futuro possibile in base alla evoluzione cambia anche il se di simboli da porre nello stack.

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

É possibile rinunciare al determinismo?
In generale no:

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

ma in molti casi di interesse pratico esistono linguaggi context-free riconoscibili da PDA deterministici: linguaggi context-free deterministici.

4.1.2 PDA deterministici

Cosa serve per ottenerlo?

Viste le condizioni precedenti, non deve succedere che l'automa, in dato stato Q_0, con simbolo in cima allo stack Z e ingresso x possa:

- portarsi in più stati futuri $sfn(Q_i, x, Z) = \{(Q_1, Z_1), (\cdots), (Q_k, Z_k)\}$
- optare se leggere o non leggere il simbolo di ingresso x a causa della presenza di entrambe le mosse $sfn(Q_i, x, Z)$ e $sfn(Q_i, \varepsilon, Z)$

Unendo, intersecando o concatenando linguaggi deterministici, non necessariamente si ottiene un linguaggio deterministico.

I complemento di un linguaggio è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il linguaggio quoziente L/R (insieme di stringhe di L private di suffisso regolare) è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il concatenamento $L.R$ (insieme di stringhe di L cun suffisso regolare) è deterministico.

Sottoclassi particolari

Per un PDA **deterministico**:

- il criterio dello stack vuoto risulta meno potente del criterio stati finali
- una limitazione sul numero di stati intenro o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- l'assenza di ε-mosse riduce l'insieme dei linguaggi riconoscibili

4.1.3 Realizzazione di PDA deterministici

Possibilità di seguire la definzione, ma non molto pratico.

Si adotta un approccio che manipoli uno stack con la stessa logica di un PDA, dove lo stack è la vera differenza, ad esempio una macchina virtuale che abbia uno stack può essere fatta funzionare come PDA, opportunamente pilotata.

Si potrebbe controllare "a mano" lo stack, oppure in modo automatico attraverso le chiamate ricorsive di funzioni, dove sono già gestiti stack relativi alla ricorsione.

Top-Down Recursive-Descent Parsing

Con l'**analisi ricorsiva discendente** si introduce una funzione per ogni metasimbolo della grammatica, e la si chiama ogni volta che si icontra quel metasimbolo.

Ogni funzione copre le regole di quel metasimbolo, ossia riconosce il sotto-linguaggio corrispondente:

- termina normalmente (o segno di successo) se incontra simboli coerenti
- abortisce (o restituisce un segno di fallimento) se incontra simboli non coerenti

Esempio

Il solito linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$, alfabeto $A = \{0,\ 1,\ c\}$ e regole $S \to 0\ S\ 0 \mid 1\ S\ 1 \mid c$.

- Introdurre tante funzioni quanti i metasimboli, qui una sola $S()$.
- Chiamare una funzione ogni volta che si incontra il suo metasimbolo
- Ogni funzione deve coprire le regole di quel metasimbolo
  - se il simbolo d'ingresso è $0 \to$ seguire la prima regola
  - se il simbolo d'ingresso è $1 \to$ seguire la seconda regola
  - se il simbolo d'ingresso è c $\to$ seguire la terza regola

Nel caso della prima regola, consumiamo il carattere di ingresso 0, invochiamo la funzione $S()$ e consumiamo un nuovo carattere d'ingresso e verifichiamo che sia 0.

Se la verifica ha esito positivo, significa che la funzone ha incontrato simboli coerenti con le proprie regole.

Se la verifica ha esito negativo, significa che la funzione ha incontrato simboli che non corrispondono alle sue regole.

4.1.4 Separare motore e grammatica

Applicare l'analisi ricorsiva discendente è un processo meccanico, che tuttavia introduce informazioni cablate nel codice, difficili da aggiornare.

É possibile separare il **motore**, invariante rispetto alle regole, dalle regole della grammatica; si presta a questo scopo la tabella di parsing, simile alla tabella delle transizioni di un RSF, indica però la prossima *produzione* da applicare.

Il motore prenderà singole decisioni consultando questa tabella.

Parsing tables - esempi deterministici

Linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

Linguaggio $L = \{\texttt{if c then cmd (endif | else cmd) }\}$

Produzioni $S \rightarrow \texttt{if c then cmd X}$, $\texttt{X} \rightarrow \texttt{endif} \mid \texttt{else cmd}$

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

4.1.5 Analisi ricorsiva discendente - vantaggi e limiti

Vantaggi

- immediata scrittura del riconoscitore
- migliore leggibilità a modificabilità del codice
- facilitata inserzione di azioni nella fase di analisi

Svantaggi

- non sempre applicabile
- funzionale solo se non esistono ambiguità sulla regola da applicare

Si individua una sottoclasse di grammatiche context-free che garantisce il determinismo dell'analisi sintattica.

Per rendere deterministica l'analisi ricorsiva, si rende necessario avere una visione del *passato* dell'analisi (simboli consumati fino a quel punto) e una del *futuro*, generalmente un solo carattere in avanti.

4.2 Grammatiche $LL(k)$

Si definiscono *grammatica* $LL(k)$ quelle che sono analizzabili in modo deterministico:

- procedendo *left to right*
- applicando *left-most derivation*

- guardando avanti al più k simboli

Ricoprono un posto fondamentale le grammatiche LL(1), ovvero quelle in cui basta guardare un simbolo in avanti per operare in modo deterministico.

Esempio

Si consideri la grammatica

- VT = {p, q, a, b, d, x, y}
- VN = {S, X, Y}

Produzioni

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{p X} \quad | \texttt{q Y} \\ \texttt{X} &\rightarrow \texttt{a X b} \quad | \texttt{x} \\ \texttt{Y} &\rightarrow \texttt{a Y d} \quad | \texttt{y} \end{aligned}$$

Le parti **destre** delle produzioni di uno stesso meta simbolo, iniziano tutte con un simbolo terminale diverso, è sufficiente guardare avanti di un carattere per scegliere con certezza la produzione per scegliere con certezza la produzione con cui proseguire l'analisi.

Creando la parsing table, si nota che ogni cella contiene una sola produzione, quindi non si hanno ambiguità sulle prossime mosse da fare, è facile vedere che il parser è deterinistico.

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

La frase di inzio deve essere completa oppure può essere parziale? Nel caso in cui si voglia imporre che la frase deve essere finale occorre imporre una regola *top-level* che specifica che la frase deve terminare con $, carattere che rappresenta il fine stringa/linea/file del tipo Z → S $.

4.2.1 Starter Symbol Set

Spesso le parti destre delle produzioni di uno stesso metasimbolo non iniziano tutte con un simbolo terminale, non è chiaro quali siano gli input ammissibili.

Occorre ridefinire il concetto di simbolo iniziale → Starter Symbols Set.

Lo starter symbols set della riscrittura α è l'insieme

$$SS(\alpha) = \{a \in \texttt{VT} \mid \alpha \rightarrow^* a\ \beta\}, \quad \texttt{con}\ \ \alpha \in V^+\ \ \texttt{e}\ \ \beta \in V^*$$

In sostanza gli starter symbols sono le iniziali di una forma di frase α, ricavate applicando con più produzioni. L'operatore $*$ cattura il caso limite in cui $\alpha \in VT$, ossia già terminale, e non richiede di applicare derivazioni.

Per includere anche il caso $\alpha \to \varepsilon$, si introduce l'insieme

$$\texttt{FIRST}(\alpha) = \texttt{trunc}_1(\{x \in VT^* \mid \to^*\}), \ \texttt{con} \ \alpha \in V^*$$

dove $\texttt{trunc}_1$ denota il troncamento della stringa al primo elemento.

Generalizzando la regola precendete, condizione necessaria perché una grammatica sia $LL(1)$ è che gli *start symbols* relativi alle parti destre di uno stesso metasimbolo siano disgiunti.

É possibile capire in modo più rapido se una grammatica è $LL(1)$?

Due opzioni:

- agire sulla parsing table, formalizzando il concetto di **blocco annullabile** e integrando nella tabella l'informazione sulle stringhe che possono scomparire.
- ampliare la nozione di starte symbols set con i Director Symbols set (o Look-Ahed set)

4.2.2 Parsing table con blocchi annullabili

Un **blocco annullabile** è una stringa che può *degenerare* in ε.

In presenza di blocchi annullabili, un metasimbolo che non appare iniziale, può trovarsi a inizio frase, per tenere conto delle ε-rules è necessario considerare anche i simboli che possono *seguire* quellli annullabili, l'insieme **FOLLOW**.

Un blocco annullabile deve essere previsto dove la sua presenza non appare evidente, ovvero in corrispondenza dei terminali che possono *seguire* il blocco annullabile.

Esempio

$$\begin{aligned}
&\texttt{S} \to \texttt{A B} \\
&\texttt{A} \to \texttt{P Q} \mid \texttt{B C} \\
&\texttt{P} \to \texttt{p P} \mid \varepsilon \\
&\texttt{Q} \to \texttt{q Q} \mid \varepsilon \\
&\texttt{B} \to \texttt{b B} \mid \texttt{d} \\
&\texttt{C} \to \texttt{c C} \mid \texttt{f}
\end{aligned}$$

Il blocco `PQ` è annullabile, perché sia `P` che `Q` possono degenerare in ε.

La regola A → P Q va inclusa in tutte le colonne della riga A che possono seguire A perché potrebbero avere un PQ invisibile davanti.

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

4.2.3 Director Symbols Set

L'idea è quella di ampliare la nozione di Starte Symbols, definendo un nuovo insieme che integri a priori l'effetto delle ε-rules.

Si crea un nuovo insieme caratterizzante, integrato dal nuovo insiem **Follow** quando qualche produzione genera la stringa vuota (o uguale allo SSS senza stringhe vuote).

Il *Director Symbols set* della produzione della $\mathtt{A} \to \alpha$ è l'unione di due insiemi:

- lo Start Symbols set
- il **Following Symbols set**

$$\mathtt{DS}(\mathtt{A} \to \alpha) = \mathtt{SS}(\alpha) \cup \mathtt{FOLLOW}(\mathtt{A}) \text{ se } \to^* \varepsilon$$

dove $\mathtt{FOLLOW}(\mathtt{A})$ denota l'insieme dei simboli che possono seguire la frase generate da A:

$$\mathtt{FOLLOW}(\mathtt{A}) = \{\mathtt{a} \in \mathtt{VT} \mid \mathtt{S} \to^* \gamma \mathtt{A} \mathtt{a} \beta\} \text{ con } \gamma, \beta \in \mathtt{V}^*$$

In pratica

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

o anche:

$$\mathtt{DS}(\mathtt{A} \to \alpha) = \mathtt{trunc}_1(\mathtt{FIRST}(\alpha) \cdot \mathtt{FOLLOW}(\mathtt{A}))$$

ossia quello che si ottiene prendendo l'iniziale delle frasi ottenute concatenando $\mathtt{FIRST}(\alpha)$ con ciò che segue A.

Questo insieme permette di formulare la condizione $\mathtt{LL}(1)$:

> Condizione necessaria e sufficiente perché una grammatica sia LL(1) è che i Director Symbols set relativi a produzione *alternativa* siano **disgiunti**.

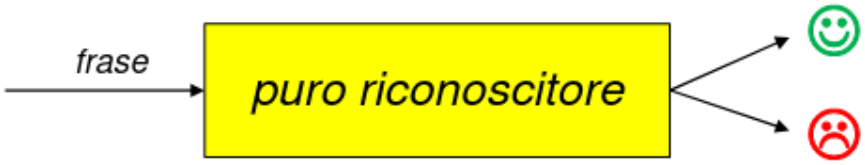
Questi set sono spesso chiamati **Look-Ahead set** perché servono per "guardare avanti" per guidare deterministicamente il parser.

5 — Dai riconoscitori agli interpreti

Dai puri riconoscitori...

Finora si sono considerati soltanto *puri riconoscitori*, che:

- accettano in ingresso una stringa di caratteri
- riconoscono se essa appartiene a un linguaggio



La risposta è sempre di tipo booleano, non ha importanza *come* si arriva a stabilire la correttezza.

...agli interpreti

Un **interprete** è più di un puro riconoscitore, riconosce una stringa ma esegue anche azioni in base al significato (*semantica*) della frase analizzata.

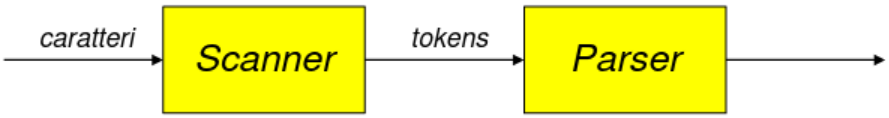
In questo caso diventa importante la sequenza di derivazione.

5.1 Interprete

5.1.1 Struttura

Un interpreste è solitamente basasto su due componenti:

- analizzatore lessicale, **scanner** o *lexer*
- analizzatore sintattico-semantico, **parser**



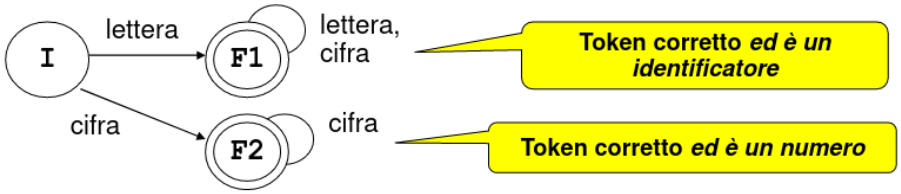
Lo scanner analizza le parti *regolari* dei linguaggi, fornendo al parser singoli token già aggregati.

Il parser riceve dallo scanner i token, considerati come elementi terminali del suo linguaggio per valutare la correttezza della loro sequenza: opera sulle parti context free del linguaggio.

5.1.2 Analisi lessicale

L'analisi lessicale è la fase in cui si individuano le singole parole (token) che compongono la frase. Questa azione viene svolta raggruppando i singoli caratteri dell'input secondo **produzioni regolari** associate alle diverse possibili **categorie** lessicali.

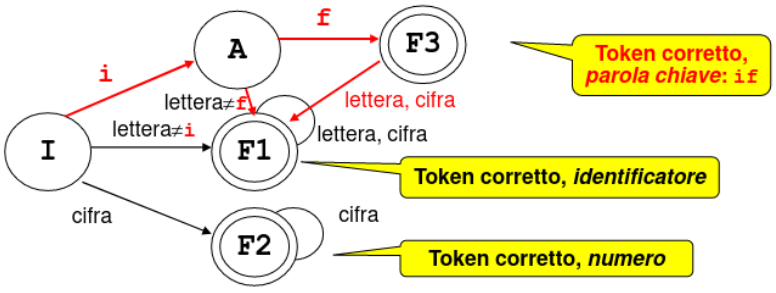
L'analizzatore lessicale categorizza i token osservando in quale stato finale del RSF si ritrova.



5.1.3 Parole chiave

Cablare ogni dettaglio nella struttura del RSF non è una buona strategia in quanto renderebbe estremamente complicato il riconoscitore.

Figura 5.1: Esempio riconoscimento identificatore if



5.1.4 Tabelle

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

- categorizzare prima le parole chiave come identificatori
- ricategorizzare poi consultando opportune tabelle che incapsulano il dettaglio del linguaggio
  - tabella *parole chiave*
  - tabella *simboli*

5.1.5 Analisi sintattica top-down

In caso di grammatiche $LL(1)$, una tecnica semplice per costruire il riconoscitore è l'analisi top-down ricorsiva discendente.

Per passare da un puro riconoscitore a un interprete occorre propagare qualcosa di più di un si o no, come ad esempio un **albero**, per una valutazione differita.

5.2 Caso di studio - espressioni aritmetiche

Si supponga di voler riconoscere espressioni aritmetiche con le quattro operazioni + * - /.

```
3+4-5     3+4*5     9-4-1     9-4/2
```

Un puro riconoscitore deve solo dire se sono corrette, ma un interprete deve anche dire:

- se il dominio sono gli *interi* il risultato può essere un valore int
- se il dominio sono i *reali* il risultato può essere un valora double
- se l'obiettivo è la *valutazione differita*, il risultato può essere un opportuno oggetto (albero)

5.2.1 Analisi del dominio

Sintassi

La notazione classica insegnata identifica i quattro operatori con i seguenti simboli: $+$, $-$, $\times$, $:$.

Sono stati sostituiti dagli informatici con $+$, $-$, $*$, $/$, spesso vengono anche usate le parentesi per esprimere una priorità associativa.

Semantica

Nel dominio aritmetico usuale:

- i *valori numerici* si assumono espressi in notazione **posizionale** su base dieci
- il significato inteso dei quattro *operatori* è quello di somma, sottrazione, moltiplicazione, divisione
- si denotano le nozioni di priorità e associatività
  - **priorità** fra operatori diversi, moltiplicativi prioritari su quelli additivi
  - **associatività** tra operatori equiprioritari, solitamente si associa a sinistra

5.2.2 Grammatica per le espressioni

Consideriamo il linguaggio $E(G)$ relativo alla seguente grammatica per espressioni aritmetiche;

$\texttt{VN} = \{\texttt{EXP}\}$
$\texttt{VT} = \{+, *, -, :, \texttt{num}\}$
$S = \texttt{EXP}$
$P = \{$
$\quad \texttt{EXP} := \texttt{EXP} + \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} - \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} * \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} : \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{num}$
$\}$

É una grammatica ambigua, la semantica è informale: se `EXP` è un numero, l'espressione denota un intero e il valore dell'espression coincide con quello del numero.

5.2.3 Una grammatica a "strati"

É possibile dare una struttura *gerarchica* alle espressioni, esprimendo così intrinsicamente priorità e associatività degli operatori.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$
$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$
$S = \texttt{EXP}$
$P = \{$
$\quad \texttt{EXP} := \texttt{TERM}$
$\quad \texttt{EXP} := \texttt{EXP} + \texttt{TERM}$
$\quad \texttt{EXP} := \texttt{EXP} - \texttt{TERM}$

TERM := FACTOR

TERM := TERM ∗ FACTOR

TERM := TERM : FACTOR

FACTOR := num

FACTOR := (EXP)

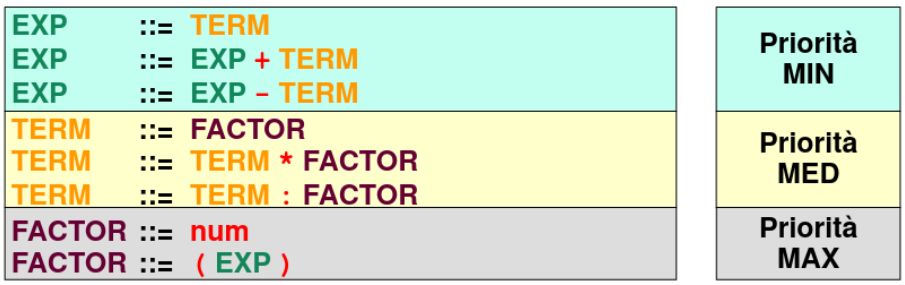
}

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

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

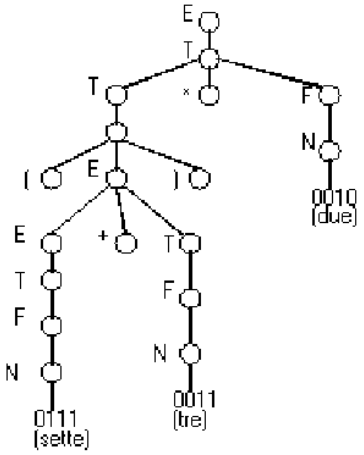
Le *somme* e *sottrazioni* aggregano i termini, le *moltiplicazioni* e *divisioni* aggregano i fattori e i *fattori* sono entità atomiche.

Figura 5.2: Priorità operatori

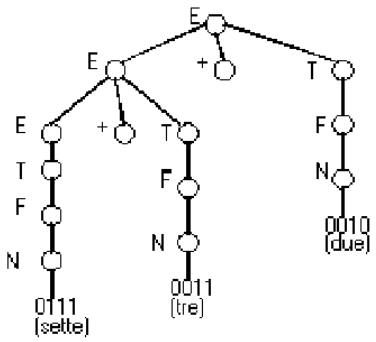


Esempi

Frase:
$(0111 + 0011) * 0010$
Albero di derivazione



Frase:
$0111 + 0011 + 0010$
Albero di derivazione



La grammatica individuata è *ricorsiva a sinistra* per esprimere la corretta associatività, tuttavia è incompatibile con l'analisi ricorsiva discendente, utile per la costruzione del parser.

5.2.4 Variante 1 - Associativa a destra

Se si riscrivono le regole in forma ricorsiva a destra, ovvero invertendo `TERM` e `EXP` e `FACTOR` e `TERM`, si otterrebbe una associatività inversa rispetto a quella usuale.

Ad esempio l'espressione $7 - 3 - 2$ verrebbe derivata come $7 - (3 - 2)$ anziché come $(7 - 3) - 2$.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$

$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$

$P = \{$

$\texttt{EXP} := \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} + \texttt{EXP}$

$\texttt{EXP} := \texttt{TERM} - \texttt{EXP}$

$\texttt{TERM} := \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} * \texttt{TERM}$

$\texttt{TERM} := \texttt{FACTOR} : \texttt{TERM}$

$\texttt{FACTOR} := \texttt{num}$

$\texttt{FACTOR} := (\texttt{EXP})$

$\}$

5.2.5 Variante 2 - Non associativa

Si potrebbe anche fare a meno dell'associatività, mantenendo però lo stesso ordine di priorità. Queste regole non usano ricorsione, ne destra ne sinistra, risolvendo il problema *obbligando a usare le parentesi*.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$

$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$

$P = \{$

$\texttt{EXP} := \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} + \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} - \texttt{TERM}$

$\texttt{TERM} := \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} * \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} : \texttt{FACTOR}$

$\texttt{FACTOR} := \texttt{num}$

$\texttt{FACTOR} := (\texttt{EXP})$

$\}$