Pilu Crescenzi Alberto Marchetti Spaccamela

LINGUAGGI, AUTOMI, GRAMMATICHE

Dispense di
Fondamenti di Informatica II

A.A. 2020-2021

Sommario

Prefazione

L'INFORMATICA UTILIZZA LINGUAGGI ARTIFICIALI che sono progettati per facilitare la comunicazione e la programmazione degli elaboratori elettronici. Sappiamo che prima di essere esegutio un programma scritto in un linguaggio come Python deve essere tradotto in un linguaggio comprensibile dall'elaboratore. Una delle cose più intriganti per chi iniza lo studio dell'informatica è capire che il processo di traduzione è effettuata dal *compilatore* un altro programma scritto in un linguaggio ad alto livello che viene eseguito dallo stesso elaboratore che poi eseguirà il programma tradotto.

La realizzazione di questo processo di traduzione richiede lo sviluppo di adeguati strumenti formali. Lo studio dei compilatori va oltre lo scopo di queste dispense. Il nostro scopo è introdurre lo studio della teoria dei linguaggi da un punto di vista formale, argomento noto come *la teoria dei linguaggi formali*.

Il nostro primo obiettivo Pertanto cercheremo di definire che cos'è un linguaggio da e poi vedremo cosa significa definire un linguaggio artificiale, come ad esempio Python o uno gli altri linguaggi usati per programmare.

Nel corso della dispensa cerchermo di dare risposta alle seguenti domande fondamentali

> COME POSSIAMO FORMALIZZARE LE DEFINIZIONI DI UNA LINGUA ARTIFICIALE UTILE IN AMBITO NFORMATICO? IN ALTRE PAROLE, POSSIAMO TROVARE RAPPRESENTAZIONI CHE POSSIAMO INTERPRETARE MATEMATICAMENTE IN MODO DA PERMETTERCI DI RAGIONARE FORMALMENTE SU DI LORO?

Il nostro interesse per i linguaggi formali è anche motivato dalla necessità di scrivere compilatori e interpreti che hanno il compito di tradurre i programmi scritti in un linguaggio di programmazione ad alto livello in un linguaggio direttamente eseguibile da un calcolatore. In questo scenario è necessario avere algoritmi che siano in grado di tradurre in modo automatico ed efficiente.

Questo ulteriore obiettivo pone altre questioni rilevanti.

- Quale formalismo per definire linguaggi è abbastanza espressivo per i nostri scopi ma risulta abbastanza semplice per realizzare un programma di traduzione automatico?
- La traduzione automatica di un programma da parte di elaboratore richiede che la definizioned el linguaggio sia non ambigua. Come possiamo essere sicuri che il formalismo che utlizziamo non dia mai ambiguità?

In queste dispense affronteremo da un punto di vista matematico lo studio dei linguaggi. Vedremo che ci sono diversi modi per frlo ognuno con i suoi vantaggi e svantaggi. Per questo non potremo limitarci a considerarne uno solo. In sintesi i punti principali di queste dispense possono essere così riassunti.

> È POSSIBILE DARE UNA DEFINIRE FORMALE DI TIPO MATEMATICO UN LINGUAGGIO USANDO LA TEORIA DEGLI INSIEMI.

In particolare, dato un alfabeto C, vedremo come definire un linguaggio come un possibile sottoinsieme di tutte le possibili sequenze dei caratteri dell'alfabeto che possiamo scrivere. Ad esempio, se consideriamo l'alfabeto italiano, il carattere spazio e i simboli di punteggiatura, e iniziamo a scrivere tutte

le possibili sequenze di questi ventidue caratteri possiamo affermare che solo una (piccola) parte delle possibili sequenze sono frasi corrette in italiano.

Ad esempio `qustadispanza è palllsisssima` non è una frase corretta in italiano mentre `questa dispensa non è semplice ma interessante` è corretta (anche se gli autori di questa dispensa non sono sicuri che quanto scritto sia condiviso dai lettori ...).

La definizione precedente di linguaggio come sottoinsieme delle possibili sequenze è matematicamente rigorosa ma non è molto utile in pratica. Innanzitutto l'insieme delle possibili sequenze di caratteri dell'alfabeto è infinita e la precedente definizione richiede di enumerare esplicitamente di enumerare tutte le sequenze ammissibili; ovviamente questo è possibile solo per quei linguaggi che hanno un numero finito di elementi. Questi linguaggi, avendo solo un numero finito di possibili sequenze , hanno uno scarso interesse pratico. In altre parole, la definizione di tipo insiemistico dei linguaggi non ci permette di ragionare su di essi ma solo di elencare le possibili frasi: abbiamo bisogno di metodi e algoritmi che ci permettano di decidere se una sequenza di caratteri appartiene o meno ad un dato linguaggio.

Pertanto dobbiamo sviluppare nuovi strumenti che costituiscono lo scopo principale di queste dispense. In particolare considereremo altri tre diversi approcci.

> È POSSIBILE DEFINIRE UN LINGUAGGIO UTILIZZANDO UN APPROCCIO DI TIPO ALGEBRICO CHE UTILIZZA OPPORTUNE OPERAZIONI MATEMATICHE PER DEFINIRE UN LINGUAGGIO.

Definiremo con questo approcco una particolare classe di linguaggi, le espressioni regolari. In particolare, dato un alfabeto Σ defineremo un linguaggio utilizzando un'espressione di tipo algebrico che utilizza i caratteri dell'alfabeto Σ e opportune operazioni di tipo algebrico e le parentesi. Queste operazioni invece di operare su numeri come nell'usuale algebra operano su caratteri e sequenze di caratteri.

Questo approccio ha il vantaggio di definire un linguaggio in modo sintetico anche se il linguaggio contiene infinite sequenze. In questo modo potemo scrivere anche rpogrammi Python che ricercano e manipolano testi.

> DATO UN LINGUAGGIO SIAMO INTERESSATI A SVILUPPARE ALGORITMI E PROGRAMMI IN GRADO DI RICONOSCERE SE UNA SEQUENZA DI CARATTERI APPARTIEME AL LINGUAGGIO O MENO.

In particolare l'approccio algebrico ci permette di fefinire un linguaggio ma non ci permette di sapere se una data sequenza di caratteri appartiene o meno ad un dato linguaggio. Per questo scopo abbiamo bisogno di algoritmi di riconoscimento che possiamo implementare in un programma. Ad esempio un compilatore per Python prende in ingresso un programma P e stabilisce se P è un programma sintatticamente corretto.

Per poter scrivere un compilatore è quindi fondamentale utilizzare metodi opportuni che riconoscano se una sequenza di caratteri appartiene ad un linguaggio.

In particolare vedremo come possiamo definire gli automi a sati finiti che sono in grado di riconoscere tutti i linguaggi che possiamo definire con le espressioni regolari.

> È POSSIBILE DEFINIRE UN LINGUAGGIO FORMALIZZANDO IN MODO OPPORTUNO IL CONCETTO DI GRAMMATICA.

Le espressioni regolari e gli automi a stati finiti permettono di definire linguaggi interessanti e di utilizzo pratico ma non comprendono tutti ilinguaggi e nemmeno i linguaggi di nostro interesse. Ad esempio, non è possibile definire con le espressioni regolari i programmi corretti che possiamo scrivere in Python. Per descrivere Python (e gli altri linguaggi di programmazione come C, PHP, SQL ecc.) useremo il concetto di grammatica.

Abbiamo imparato a scuola che lo studio di una lingua naturale come l'Italiano o l'Inglese richiede lo studio della sua grammatica. Le regole grammaticali di un linguaggio come l'italiano sono moltissime e di difficile formalizzazione in termini mtematici. Però agli inizi del secolo scorso un grande studioso dei

linguaggi Noam Chomsky formalizzò il concetto di grammatica in modo formale con lo scopo di studiare in questo modo i linguaggi naturali (come Italiano e Inglese). Successivamente i suoi studi hanno avuto una enorme rilevanza quando a metà del secolo scorso gli sviluppi dell'informatica hanno portato alla definzione di linguaggi artificiali con cui programmare.

In questo modo possiamo definire formalmente un linguaggio artificiale come ad esempio Python utilizzando un insieme relativamente piccolo di regole grammaticali.

Il vantaggio di questo approccio è la possibilità di defnire in modo sintetico linguaggi complessi; lo svantaggio è che il problema del riconoscimento per questi linguaggi è molto più complesso che ne caso delle espressioni regolari. Lo studio approfondito di queste problematiche va oltre lo scopo di queste dispense e per questa ragione ci limiteremo a a considerare alcuni aspetti.

Riferimenti bibliografici

Queste dispense non presuppongono particolari conoscenze da parte dello studente, a parte quelle relative a nozioni matematiche e logiche di base. Queste dispense sono tratte dalle dispense del corso di Informatica Teorica insegnato a Firenze da P.Crescenzi ed accessibili liberamente:

P. Crescenzi *Informatica Teorica*, dispense 2011, https://dl.dropboxusercontent.com/u/1975733/PC/IT.pdf

Esistono numerosi libri di testo dedicati agli argomenti trattati in queste dispense: in particolare, si consiglia come lettura aggiuntiva:

G. Ausiello, F. D'Amore, G. Gambosi. *Linguaggi, modelli, complessità*, Franco Angeli, 2003.

Capitolo 1

Introduzione

SOMMARIO

In questo capitolo iniziamo lo studio dei linguaggi da un punto di vista formale. Innanzitutto definiremo da un punto di vista matematico che cos'è un linguaggio da e poi introduciamo i possibili approcci allo studio formale dei linguaggi con particolare riferimento ai linguaggi di programmazione.

1.1 Sintassi e semantica di un linguaggio

IN QUESTA DISPENSA iniziamo lo studio dei linguaggi di programmazione da un punto di vista formale. In modo informale possiamo affermare che un linguaggio naturale come l'italiano o l'inglese, consiste in un alfabeto, un vocabolario di parole, e in un insieme di regole per esprimere idee, fatti, concetti. Lo studio dei linguaggi ci permette di dire se una frase o un saggio sono corretti (cioè rispettano le regole del linguaggio).

E' ben noto che lo studio di una lingua consiste essenzialmente nello studio della sintassi e della semantica.

Sintassi
La sintassi specifica le regole secondo le quali una frase è corretta o meno nella lingua: una frase è sintatticamente corretta se è ottenuta applicando da un numero dei componenti che abbiano un senso strutturale.

Ad esempio, una frase in italiano della forma: è
`<soggetto> <verbo> <complemento>`
Sapendo inoltre che `<soggetto>` e `<complemento>` possono essere formati da un articolo e un sostantivo e che *MANGIA* è un verbo possiamo affermare che la frase *IL TOPO MANGIA IL FORMAGGIO*, sia corretta (supponendo che gli articoli e il verbo siano accordati correttamente secondo il genere e il numero).

D'altra parte, la sintassi dell'italiano non prevede una frase della forma:
`<soggetto> <soggetto> <complemento>`
e pertanto possiamo affermare che la frase *TOPO GATTO FORMAGGIO* non è sintatticamente corretta in italiano.

Semantica
La sintassi non dice nulla circa il significato di una frase; questo è il compito della semantica. Possiamo facilmente trovare frasi contradditorie o frasi sintatticamente corrette che non hanno alcun significato. L'esempio classico è la seguente frase di Chomsky (studioso che ha dato un contributo fondamentale allo studio dei linguaggi):

```
Idee verdi incolori dormono furiosamente
```

Possiamo affermare quindi che la semantica permette di analizzare le frasi (e i testi) per verificare se hanno un qualsiasi senso per tutti[1].

Sintassi e semantica sono rilevanti non solo per i linguaggi naturali ma anche nel caso dei linguaggi artificiali come i linguaggi di programmazione o la matematica. In matematica, per esempio, se x e y sono variabili e / rappresenta il simbolo di divisione allora l'espressione x/y è sintatticamente corretta mentre non lo è $x//y$. Osserviamo peró che se $x = 10$ e $y = 0$ allora $10/0$ è un'espressione matematica che non ha un'interpretazione semantica significativa.

Nel caso dei linguaggi di programmazione al posto delle frasi dell'italiano scriviamo programmi; l'analisi sintattica del programma che stabilisce se il programma rispetta le regole del linguaggio è fatta automaticamente da un compilatore; se il programma non è corretto il compilatore fornisce gli errori sintattici. Gli errori semantici appaiono durante l'esecuzione del programma (a "run-time") quando il calcolatore interpreta il codice compilato.

Purtroppo non abbiamo a disposizione uno strumento analogo che sia in grado di individuare gli errori semantici di un programmma. In particolare, per verificare la correttezza semantica di un programma sono note solo soluzioni parziali che operano in casi ristretti.

In queste dispense concentreremo la nostra attenzione sulla definizione formale e non ambigua della sintassi di un linguaggio; questa definizione non è ovvia e richiede l'introduzione di concetti e formalismi nuovi.

Cominciamo elencando una serie di proprietà che sembrano evidenti nella definizione di un linguaggio sia naturale che artificiale:

- un numero di parole (o simboli o altro) che sono utilizzati come elementi di base e che di solito è indicato come l'alfabeto
  Ad esempio in matematica, possiamo avere le cifre e i simboli di operazioni, mentre in italiano avremo parole del dizionario italiano.

- le regole che stabiliscono che solo certe sequenze di simboli di base sono frasi validi nella lingua, mentre altre non lo sono. Ad esempio, in matematica l'espressione $2+2$ è corretta mentre $2+++2$ non lo è.

In conclusione possiamo dire che un linguaggio è un insieme di sequenze di simboli (le "frasi" del linguaggio che rispettano le regole del linguaggio.

La definizione data di linguaggio richiede di definire le regole per cui possiamo stabilire se una sequenza è corretta o no. Questo problema è complesso perchè le frasi di un linguaggio e, quindi, sequenze possibili sono in numero infinito. Per questa ragione non possiamo pensare di definire le sequenze ammissibili fornendone un elenco e dobbiamo individuare dei metodi adeguati.

Notiamo che l'osservazione precedente si applica anche al linguaggio naturale come l'italiano o l'inglese. Il nostro cervello è di dimensioni finite, ma allora in che modo possiamo imparare un linguaggio infinito? La soluzione è in realtá piuttosto semplice: per stabilire se una frase è corretta o meno è sufficiente osservare che la *grammatica* di un linguaggio permette di frinire le regole che stabiliscono se una fras e è corretta o meno. In particolare, una grammatica consiste di un numero finito di regole e diciamo che una frase è corretta se rispetta le regole della grammatica.

In questo modo non dobbiamo ricordare se la frase *"mangio una mela"* sia sintatticamente corretta, ma lo possiamo dedurre applicando mentalmente una serie di regole per dedurre la sua validità.

[1] Nel linguaggio naturale parlato, usiamo regolarmente frasi sintatticamente scorrette, anche se l'ascoltatore di solito riesce a 'determinare' la sintassi e la semantica sottostanti e capire cosa intende chi parla. Questo è particolarmente evidente nei discorsi di un bambino o in alcune poesie.

1.2 Alfabeti, stringhe, linguaggi

Un *alfabeto* è un insieme finito non vuoto di simboli (caratteri).

Possibili esempi di alfabeto sono

- l'alfabeto binario $\{0,1\}$,
- l'alfabeto delle cifre decimanli $\{0,1,2,3,4,5,6,7,8,9\}$
- l'alfabeto italiano: $\{a,b,c,d,e,f,g,h,i,l,m,n,o,p,q,r,s,t,u,v,z\}$.

Definizione 1.1: Stringa

- Dato un alfabeto Σ una sequenza finita di caratteri di Σ è una *stringa (o parola)*.
- Data una stringa x, $|x|$ rappresenta il numero di caratteri che la costituiscono (la lunghezza della stringa).
- L'operazione fondamentale sulle stringhe è la concatenazione che consiste nel giustapporre due stringhe; in particolare date w_1 e w_2 la loro concatenazione è w_1w_2.
- La stringa stringa vuota o nulla è la stringa di lunghezza zero (quella non costituita da alcun simbolo dell'alfabeto) ed è denotata con ϵ.
- L'insieme di tutte le stringhe definite sull'alfabeto Σ (inclusa la stringa vuota) è denotato Σ^*.

Esempio 1.1:

- 01000010 è una stringa definita sull'alfabeto binario $\{0,1\}$ di lunghezza è 8. Essa appartiene quindi a $\{0,1\}^*$.
- Se consideriamo l'alfabeto italiano: $\{a,b,c,d,e,f,g,h,i,l,m,n,o,p,q,r,s,t,u,v,z\}$ allora le parole possibili stringhe sono tutte le sequenze che possiamo scrivere. Pertanto possibili stringhe di lunghezza quattro sono *topo, rana, aaaa, zzzz*, mentre *xaaa* **non** è una stringa perché contiene la lettera x che non appartiene all'alfabeto italiano.
- Se consideriamo l'alfabeto itialiano e le due parole w_1= salva, w_2= gente, lallora la loro concatenazione è w_1w_2 = salvagente
- Se consideriamo l'alfabeto binario allora Σ^* denota tutte le possibili stringhe binarie di qualunque lunghezza.

La concatenazione è un'operazione associativa ma non commutativa. Infatti, dati w_1= abb, w_2=bba, abbiamo $((w_1w_2)w_1)$ = abbbbaabb = $(w_1(w_2w_1))$.

Osserviamo inoltre che w_1w_2 = abbbba $\neq w_2w_1$= bbaabb

Concatenando una stringa con la stringa vuota ϵ otteniamo la stringa stessa; in particolare per ogni stringa w, $w\epsilon = w = \epsilon w$

Per indicare la ripetizione di simboli o, più in generale, la concatenazione di due o più stringhe uguali si usa il simbolo di potenza. Abbiamo ad esempio: $ab^4a = abbbba$ e $x^3 = xxx$.

Inoltre, se $w = cous$, allora con w^2 otteniamo la stringa $couscous$.

Definizione 1.2: Linguaggio

Un linguaggio è un insieme di parole definite su un alfabeto.
Formalmente un linguaggio definito su un alfabeto Σ è un sottoinsieme di Σ^*.

Esempio 1.2:

Esempi di possibili linguaggi sono i seguenti:

- Dato l'alfabeto $\{a, b\}$, l'insieme $L = a^n b^n | n \geqslant 0$ è il linguaggio di tutte le stringhe costituite da una sequenza di n a ($n \geqslant 0$), seguite da altrettante b. Pertanto $aaabbb \in L$, $aaabb \notin L$, $\epsilon \in l$.
- Dato l'alfabeto $\{I, V, X, L, C, D, M\}$, l'insieme di tutti i numeri da 1 a 3000 rappresentati come numeri romani Ã¨ un linguaggio.
- Dato l'alfabeto $\{0, 1\}$ l'insieme di tutte le stringhe che contengono un numero pari di 1 è un linguaggio.

Dato un qualunque alfabeto Σ possiamo affermare che

- Σ stesso è un linguaggio le cui stringhe sono gli elementi dell'alfabeto; ad esempio se $\Sigma = \{0, 1\}$ è l'alfabeto binario allora $\{\{0\},\{1\}\}$ è un linguaggio.
- Σ^* è un linguaggio formato da tutte le possibili stringhe che si possono ottenere da Σ; quindi nel caso di alfabeto binario contiene la stringa vuota e tutte le infinite sequenze di 0 e 1 di qualunque lunghezza.
- L'insieme che non contiene nessuna stringa, denotato $\emptyset$ è un linguaggio (detto linguaggio vuoto). Osserviamo che

  $$\emptyset \neq \{\epsilon\}$$

  Infatti $\emptyset$ denota il linguaggio che non contiene nessuna stringa mentre $\{\epsilon\}$ è il linguaggio che contiene solo la stringa vuota.

1.3 Approcci formali allo studio dei linguaggi

NON TUTTI I LINGUAGGI che si possono definire su un dato alfabeto sono interessanti. In particolare siamo interessati a linguaggi le cui stringhe hanno struttura particolare ovvero obbediscono a particolari regole. Possibili esempi di linguaggi sono i seguenti:

1. il linguaggio costituito da stringhe di parentesi bilanciate del tipo: $(()(()))()$ - ma che non include stringhe di parentesei come $(()$ oppure $)(()))($;
2. il linguaggio costituito da espressioni aritmetiche contenenti identificatori di variabili, numeri e simboli delle quattro operazioni e delle parentesi;
3. il linguaggio costituito da tutti i programmi sintatticamente corretti (cioè accettati da un compilatore senza segnalazione di errore) scritti nel linguaggio Python.

Osserviamo che la precedente definizione di un linguaggio come un sottoinsieme delle stringhe è molto generale ma non permette di definire i linguaggi precedenti. Abbiamo quindi bisogno di individuare nuovi strumenti per definire linguaggi di nostro interesse.

Approcci allo studio dei linguaggi

Lo studio formale dei linguaggi utilizza tre diversi approcci:

1. un approccio *algebrico*, che mostra come costruire un lingauggio a partire da linguaggi più elementari utilizzando operazioni su linguaggi;
2. un approccio *riconoscitivo*, che definisce una 'macchina' o un algoritmo di riconoscimento che ricevendo una stringa in input dice se essa appartiene o no al linguaggio:
3. un approccio *generativo*, che definisce attraverso una grammatica le regole strutturali che devono essere soddisfatte dalle stringhe che appartengono al linguaggio.

Nei capitoli successivi considerermo questi diversi approcci. In particolare, nel capitolo 2 introduciamo le *espressioni regolari*, utilizzando l'approccio algebrico. Le espressioni regolari sono una classe di linguaggi che hanno una rilevanza pratica; ad esempio Python permette di definire espressioni regolari per effettuare ricerche complesse in file di dati. Tuttavia questi linguaggi sono limitati nel loro potere espressivo. Ad esempio, non sono in grado di definire linguaggi di programmazione come Python; inoltre le espressioni regolari non permettono nemmeno di descrivere il linguaggio che contiene tutte le espressioni aritmetiche ben formate (linguaggio in cui usiamo numeri, i simboli delle quattro operazioni aritmetiche e le parentesi).

Nel cpitolo 3 consideriamo anche l'approccio riconoscitivo presentando gli *automi a stati finiti*, semplici macchine in grado di riconoscere se una stringa appartiene ad una espressione regolare. Per introdurre gli automi a stati finiti introduciamo preliminarmente le macchien di Turing un improtante formalismo di calcolo teorico proposto negli anni '20 del secolo scorso.

Nel capitolo 4, considereremo l'approccio generativo e vedremo la classificazione delle grammatiche basata sul tipo di regole ammesse. La classe più semplice corrisponde ai linguaggi regolari che vederemo sono gli stessi linguaggi definibili con le espressioni regolari. Le altre classi di linguaggi hanno un potere espressivo maggiore di quello dei linguaggi regolari.

Il nostro interesse per i linguaggi formali è anche motivato dalla necessità di scrivere compilatori e interpreti che hanno il compito di tradurre i programmi scritti in un linguaggio di programmazione ad alto livello in un linguaggio direttamente eseguibile da un calcolatore. Una trattazione completa delle orblematiche relative alla traduzione è oltre gli scopi di questa dispensa. Nel capitolo 4 ci limitiamo a introdurre alcune probematiche discutendo in particolare il concetto di grammatica ambigua.

Capitolo 2

Le grammatiche generative

SOMMARIO

La teoria dei linguaggi formali (ovvero la teoria matematica delle grammatiche generative) è stata sviluppata originariamente per i linguaggi naturali (come l'italiano) e solo in seguito fu scoperta la sua utilità nel progetto di compilatori. In questo capitolo, facendo riferimento alla ben nota gerarchia di Chomsky, introduciamo le nozioni di base di tale teoria e mostriamo le connessioni esistenti tra due diversi tipi di grammatiche generative e due diversi tipi di macchine di Turing.

2.1 Produzioni, simboli terminali e non terminali

LE GRAMMATICHE generative furono introdotte dal linguista Noam Chomsky negli anni cinquanta del secolo scorso con lo scopo di individuare le procedure sintattiche alla base della costruzione di frasi in linguaggio naturale. Pur essendo stato ben presto chiaro che tali grammatiche non fossero sufficienti a risolvere tale problema, esse risultarono estremamente utili nello sviluppo e nell'analisi di linguaggi di programmazione.

Intuitivamente, una grammatica generativa specifica le regole attraverso le quali sia possibile, a partire da un simbolo iniziale, produrre tutte le stringhe appartenenti a un certo linguaggio.

Come primo esempio consideriamo una possibile grammatica per generare le date nel formato

GG/MM/AAAA

allora vuol dire che se siamo nati il 1 Gennaio del 1997 dobbiamo utilizzare due cifre per il giorno, due cifre per il mese e quattro per l'anno e scrivere

01/01/2019

ogni altro formato, come ad esempio 1/1/2019 oppure 01/01/19 è errato.

Per esprimere a parole le regole di formazione di una data secondo il formato precedente possiamo dire che una data è composta da tre parti che sono separate dal carattere '/' ; la prima e la seconda parte descrivono rispettivamente il giorno e il mese e sono formate da due cifre, la terza parte descrive l'anno ed è formata da quattro cifre. Un cifra infine è uno dei numeri da 0 a 9.

La seguente definizione formalizza quanto espresso.

- `Data` → `Giorno` /`Mese`/`Anno`
- `Giorno` → `Cifra` `Cifra`
- `Mese` → `Cifra` `Cifra`
- `Anno` → `Cifra` `Cifra` `Cifra` `Cifra`

- `Cifra` → 0|1|2|3|4|5|6|7|8|9

Per esprimere la data 01/01/1997 possiamo operare i seguenti passaggi
`Data` → `Giorno` /`Mese`/`Anno`
→ `Cifra Cifra`/`Mese`/`Anno`
→ `Cifra Cifra`/`Cifra Cifra`/`Anno`
→ `Cifra Cifra`/`Cifra Cifra`/`Cifra Cifra Cifra Cifra`
→ 0`Cifra`/`Cifra Cifra`/`Cifra Cifra Cifra Cifra`
→ 01/`Cifra Cifra`/`Cifra Cifra Cifra Cifra`
→ 01/0`Cifra`/`Cifra Cifra Cifra Cifra`
→ 01/01/1`Cifra Cifra Cifra`
→ 01/01/19`Cifra Cifra`
→ 01/01/191`Cifra`
→ 01/01/1919

Come ulteriore esempio consideriamo la frase *il cane morde la mucca pigra*. Utilizzando la linguistica sappiamo che la frase precedente è composta da un sintagma nominale (che include il soggetto della frase) seguita da un sintagma verbale (che include il verbo e il complemento oggetto). Il sintagma è, in linguistica strutturale, un'unitÃ (di proporzioni variabili) della struttura sintattica di un enunciato. In generale, nel contesto di una frase, si dicono sintagmi dei costituenti strutturali, composti da elementi appartenenti a diverse categorie lessicali (o parti del discorso).

Nel seguito possiamo assumere che un sintagma nominale è costituito da un articolo (detto anche determinante) che è facoltativo seguito da un nome (il soggetto) e da una lista di nessuno uno o pi'u aggettivi. Un sintagma verbale consiste di un verbo seguito da un sintagma nominale (non considerando, per semplicità, la possibilità che vi sia anche una preposizionale). Tutto ciò potrebbe essere formalizzato attraverso le seguenti regole di produzione. Ricordiamo che il simbolo ϵ rappresenta la stringa vuota. Nel seguito assumiamo per semplicità, che al più un aggettivo possa seguire un nome.

- `Frase` → `Sintagma-nominale Sintagma-verbale`
- `Sintagma-nominale` → `Determinante Nome Aggettivo`
- `Sintagma-verbale` → `Verbo Sintagma-nominale`
- `Determinante` → `il` | `un` | `la` | `una` | ϵ
- `Nome` → `cane` | `mucca`
- `Aggettivo` → `furioso` | `pigra` | ϵ
- `Verbo` → `morde` | `guarda`

Facendo uso di queste regole possiamo, ad esempio, generare la frase

`il cane morde la mucca`

operando i seguenti passaggi successivi
`Frase` → `Sintagma-nominale Sintagma-verbale`
→ `Determinante Nome Aggettivo Sintagma-verbale`
→ `il cane Sintagma-verbale`
→ `il cane Verbo Sintagma-nominale`
→ `il cane morde Sintagma-nominale`
→ `il cane morde Determinante Nome Aggettivo`
→ `il cane morde la mucca pigra`

oppure la frase

```
una mucca pigra guarda un cane furioso
```

Ovviamente, il linguaggio italiano (così come tutti i linguaggi naturali) è così complesso che le regole sopra descritte non sono certo in grado di catturare il meccanismo alla base della costruzione di frasi sintatticamente corrette. Ad esempio usando le regole precedenti possiamo anche ottenere la seguente frase sintatticamente scorretta

```
una cane pigra guarda la mucca furioso
```

Gli esempi precedenti motivano la seguente definizione.

Definizione 2.1: grammatica generativa

Una **grammatica** è una quadrupla (V, T, S, P) dove:

1. V è un insieme finito non vuoto di simboli **non terminali**, talvolta anche detti *categorie sintattiche* oppure *variabili sintattiche*;
2. T è un insieme finito non vuoto di simboli **terminali** (osserviamo che V e T devono essere insiemi disgiunti);
3. S è un simbolo **iniziale** (anche detto *simbolo di partenza* oppure *simbolo di frase*) appartenente a V;
4. P è un insieme finito di **produzioni** della forma $\alpha \to \beta$ dove α e β, dette **forme sentenziali**, sono sequenze di simboli terminali e non terminali con α contenente almeno un simbolo non terminale.

Nel seguito, parlando di grammatiche in generale, i simboli non terminali saranno rappresentati da lettere maiuscole, i terminali da lettere minuscole all'inizio dell'alfabeto, sequenze di terminali da lettere minuscole alla fine dell'alfabeto, e sequenze miste da lettere greche. Per esempio, parleremo dei non terminali A, B e C, dei terminali a, b e c, delle sequenze di terminali w, x e y e delle forme sentenziali α, β e γ.

Intuitivamente, una produzione significa che ogni occorrenza della sequenza alla sinistra della produzione (ovvero α) può essere sostituita con la sequenza alla destra (ovvero β). In particolare, diremo che una stringa ψ è **direttamente generabile** da una stringa ϕ se $\phi = \gamma\alpha\delta$, $\psi = \gamma\beta\delta$ e $\alpha \to \beta$ è una produzione della grammatica.

Le **frasi** del linguaggio associato a una grammatica sono, dunque, generate partendo da S e applicando le produzioni fino a quando non restano solo simboli terminali.

SI tratta di una definizione *ricorsiva*: esiste un caso iniziale (la sequenza composta dal solo simbolo iniziale S) e poi si applicano ripetutamente le possibili sostituzioni indicate dalle produzioni. In particolare, diremo che la stringa di simboli terminali e non temrinali α è generabile da S, se esiste una sequenza di stringhe $S = \beta_1, \beta_2, \ldots, \beta_{n-1}, \beta_n = \alpha$ tale che, per ogni i con $1 \leqslant i < n$, β_{i+1} è direttamente generabile da β_i.

L'insieme di tutte le sequenze di terminali $x \in T^*$ che sono generabili da S formano il **linguaggio generato** dalla grammatica G, il quale è denotato con $L(G)$.

Esempio 2.1: due grammatiche generative per le date

L'esempio precedente per definire una data porta alla seguente definizione di grammatica. $G = (V, T, S, P)$, dove $V = \{S, Giorno, Mese, Anno, Cifra\}$, $T = \{0,1,2,3,4,5,6,7,8,9,/\}$ e P contiene le seguenti produzioni:

$$S \to Giorno/Mese/Anno$$

$$Giorno \to Cifra/, Cifra$$

$$Mese \to Cifra/, Cifra$$

$$Anno \to Cifra/, Cifra/, Cifra/, Cifra$$

$$Cifra \to 0|1|2|3|4|5|6|7|8|9$$

Una seconda grammatica equivalente e che utilizza un minor numero di simboli non terminali è $G = (V, T, S, P)$, dove $V = \{S, Cifra\}$, $T = \{0,1,2,3,4,5,6,7,8,9,/\}$ e P contiene le seguenti produzioni:

$$S \to CifraCifra/CifraCifra/CifraCifraCifraCifra$$

$$Cifra \to 0|1|2|3|4|5|6|7|8|9$$

È facile verificare che in ambedue i casi il linguaggio $L(G)$ coincide con l'insieme delle stringhe che iniziano con due cifre seguite dal simbolo /, da altre due cifre, dal simbolo / e infine da quattro cifre. Quindi il linguaggio definito è quello delle date nel formato

$$GG/MM/AAAA$$

.

Esempio 2.2: un secondo esempio di grammatica generativa

Consideriamo la grammatica $G = (V, T, S, P)$, dove $V = \{S\}$, $T = \{a, b\}$ e P contiene le seguenti produzioni:

$$S \to aSb$$

$$S \to ab$$

È facile verificare che il linguaggio $L(G)$ coincide con l'insieme delle stringhe $x = a^n b^n$, per $n \geqslant 1$. In effetti, ogni stringa x di tale tipo può essere generata applicando $n-1$ volte la prima produzione e una volta la seconda produzione. Viceversa, possiamo mostrare, per induzione sul numero n di produzioni, che ogni sequenza di terminali x generabile da S deve produrre una sequenza di n simboli a seguita da una sequenza di n simboli b, ovvero $x = a^n b^n$. Se $n = 1$, questo è ovvio, in quanto l'unica produzione applicabile è la seconda. Supponiamo, quindi, che l'affermazione sia vera per $n > 0$ e dimostriamola per $n+1$. Poiché $n+1 > 1$, abbiamo che la prima produzione applicata nel generare x deve essere $S \to aSb$: quindi, $x = ayb$ dove y è generabile da S mediante l'applicazione di n produzioni: per ipotesi induttiva, $y = a^n b^b$, per cui $x = a^{n+1} b^{n+1}$.

2.1.1 La notazione BNF

FIno a non molti anni fa i manuali che illustravano i linguaggi di programmazione utilizzavano , per descrivere la sintassi del linguaggio, la notazione nota come BNF (Backus-Naur Form). Per completezza la ricordiamo brevemente.

La forma di Backus e Naur rappresenta in modo più compatto una grammatica così come è stata definita all'inizio del capitolo.

In partcolare mantiene i concetti di simboli non terminali, simboli terminali, assioma e produzioni. Si differenzia per una imp0licita rappresentazione che risulta pi‘ăgevole per grammatiche complesse.

In particolare adotta la convenzione di rappresentare i simboli non terminali tra parentesi angolari "<′ ' e ">". Al posto del simbolo $\to$ si utilizza il simbolo '::=’.

Si utlizzano inoltre i seguenti simboli

- '|' che viene letto come 'oppure'
- '+' che denota la ripetizione da 1 ad un numero qualunque di volte del simbolo precedente

2.2 La gerarchia di Chomsky

In base alle restrizioni che vengono messe sul tipo di produzioni che una grammatica può contenere, si ottengono classi di grammatiche diverse. In particolare, Chomsky individuò quattro tipologie di grammatiche generative, definite nel modo seguente.

Grammatiche regolari o di tipo 3 In questo caso, ciascuna produzione della grammatica sono **lineari a destra** ovvero deve essere del tipo $A \to aB$ oppure del tipo $A \to a$[1].
Un linguaggio L **è regolare o di tipo 3** se esiste una grammatica di tipo 3 che lo genera.

Grammatiche libere da contesto o di tipo 2 In questo caso, ogni produzione della grammatica deve essere del tipo $A \to \alpha$.
Un linguaggio L **è libero dal contesto o di tipo 2** se esiste una grammatica di tipo 2 che lo genera.

Grammatiche contestuali o di tipo 1 In tal caso, ciascuna produzione della grammatica deve essere del tipo $\alpha \to \beta$ con $|\beta| \geqslant |\alpha|$.
Un linguaggio L **è contestuale o di tipo 1** se esiste una grammatica di tipo 1 che lo genera.

Grammatiche non limitate o di tipo 0 In tal caso, nessun vincolo sussiste sulla tipologia delle produzioni della grammatica.

È facile verificare che le grammatiche dell'Esempio 2.1) sono ambedue grammatiche regolari mentre quello dell'esempio 2.2) è libera dal contesto.

La seguente definizione classifica i linguaggi sulla base delle caratterisitiche della grammatica che lo genera.

Definizione 2.2: Gerarchia di Chomsky dei linguaggi

Un linguaggio

- è **regolare** se è di tipo 3,
- **libero da contesto** se è di tipo 2,
- **contestuale** se è di tipo 1.

Si noti che una grammatica di tipo i, per i compreso tra 1 e 3, è anche una grammatica di tipo $i-1$; pertanto, l'insieme dei linguaggi generati da una grammatica di tipo i (anche detti **linguaggi di tipo** i) è contenuto nell'insieme dei linguaggi generati da una grammatica di tipo $i-1$.

È possibile dimostrare che queste inclusioni sono strette, nel senso che esistono linguaggi di tipo i che non sono di tipo $i+1$, per ogni $i = 0, 1, 2$. In particolare, vedremo nel Teorema **??** che il linguaggio dell'Esempio 2.2 è di tipo 2 ma non di tipo 3; in particolare vedremo che grammatiche di tipo 3 sono limitate e non permettono di rconoscere correttamente tutte le stringhe che appartengono al linguaggio.

Il linguaggio del prossimo esempio è di tipo 1 ma non di tipo 2. Nell'esempio vedremo una grammatica di tipo 1 che genera il linguaggio; rimandiamo ai riferimenti bilbiografici la prova che non esiste una grammatica di tipo 2 in grado di generare il linguaggio.

[1] Esiste una definizione alternativa - detta lineare a sinistra - in cui le produzioni sono del tipo $A \to Ba$ oppure del tipo $A \to a$.

Esempio 2.3: una grammatica contestuale

Consideriamo il linguaggio L costituito da tutte e sole le stringhe del tipo $0^n1^n2^n$, per $n > 0$. Tale linguaggio è generato dalla grammatica contestuale $G = (V, T, S, P)$, dove $V = \{S, A, B\}$, $T = \{0, 1, 2\}$ e P contiene le seguenti regole di produzione:

$$S \to 012 \quad S \to 0A12 \quad A1 \to 1A \quad A2 \to B122 \quad 1B \to B1 \quad 0B \to 00 \quad 0B \to 00A$$

Chiaramente, mediante la prima produzione possiamo generare la stringa 012. Per generare, invece, la stringa $0^n1^n2^n$ con $n > 1$ possiamo anzitutto applicare la seconda produzione (ottenendo 0A12), la terza produzione (per spostare il simbolo A dopo il simbolo 1), la quarta produzione (ottenendo 01B122), la sesta produzione (per spostare il simbolo B prima dei simboli 1) e, infine, la settima produzione (se $n = 2$) oppure l'ottava produzione (ottenendo 00A1122). Se $n > 2$, tale sequenza di produzioni può essere ripetuta altre $n-2$ volte. Ad esempio, la stringa $0^31^32^3$ può essere generata mediante la seguente sequenza di produzioni: $S \to 0A12 \to 01A2 \to 01B122 \to 0B1122 \to 00A1122 \to 001A122 \to 0011A22 \to 0011B1222 \to 001B11222 \to 00B111222 \to 000111222$. Osserviamo che questo tipo di sequenze di produzioni sono anche le uniche possibili, in quanto in ogni istante il contesto determina univocamente quale produzione può essere applicata: in effetti, ogni forma sentenziale prodotta a partire da S può contenere un solo simbolo non terminale, ovvero A o B, e il carattere che segue o precede tale simbolo non terminale determina quale produzione può essere applicata. Quindi, il linguaggio generato da G coincide con il linguaggio L.

2.3 Linguaggi di tipo 0

IL PRIMO risultato che presentiamo in questa parte delle dispense mostra come la classificazione di Chomsky rientri nell'ambito dei linguaggi semi-decidibili.

Definizione 2.3: Linguaggi decidibili e semi-decidibili

Dato un linguaggio L diciamo che

1. L **è decidibile** se esiste una Macchina di Turing T che con input x si ferma in uno stato accettante se $x \in L$; se $x \notin L$ T si ferma e rifiuta x. In questo caso diciamo che T decide L.
2. L **è semi-decidibile** se esiste una Macchina di Turing T che con input x si ferma in uno stato accettante se $x \in L$. In questo caso diciamo che T semi-decide L.

Osserviamo che nella definizione di semi-decidibilità precedente nel caso in cui $x \notin L$ non si richiede che la macchina di Turing termini la computazione.

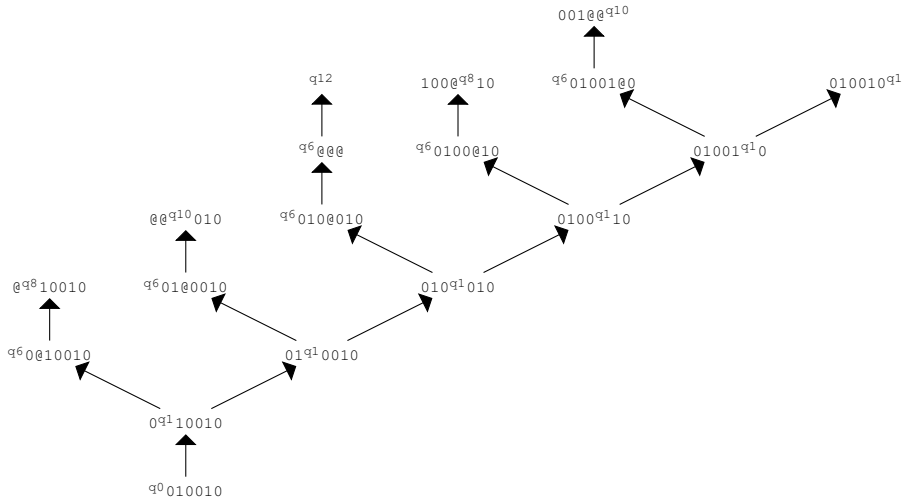
In questa sezione e nella seguente vederemo che i linguaggi di tipo 0 sono semi-decidibili mentre i linguaggi di tipo 1 sono decidibili.

Per dimostrare questi risultati introduciamo una variante delle macchine di Turing che, pur essendo equivalente al modello introdotto nel primo capitolo delle dispense, ci consentirà di dimostrare più agevolmente che la classe dei linguaggi generati da grammatiche di tipo 0 coincide con quello dei linguaggi semi-decidibili.

2.3.1 Macchine di Turing non deterministiche

Una **macchina di Turing non deterministica** è una macchina di Turing a cui non viene imposto il vincolo che, dato uno stato della macchina e un simbolo letto dalla testina, la transizione da eseguire sia univocamente determinata. Quindi, il grafo delle transizioni di una macchina di Turing non deterministica può includere un arco uscente dallo stato q la cui etichetta contiene almeno due triple (σ, τ_1, m_1) e

Figura 2.1: un albero delle computazioni di una macchina non deterministica.



(σ, τ_2, m_2) tali che $\tau_1 \neq \tau_2 \vee m_1 \neq m_2$, oppure due o più archi distinti uscenti dallo stato q le cui etichette includano ognuna almeno una tripla con lo stesso primo simbolo.

Poiché il comportamento di una macchina di Turing non deterministica, con input una stringa x, non è univocamente determinato da x stesso, dobbiamo modificare le nozioni di calcolabilità e di decidibilità fornite nel terzo capitolo. Per semplicità, nel seguito ci limiteremo a considerare macchine di Turing che decidono (oppure semi-decidono) linguaggi, anche se tutto quello che diremo si può estendere al caso di macchine di Turing che calcolano funzioni (sia pure con la specifica di diversi dettagli di carattere puramente tecnico).

Supponiamo che una macchina di Turing non deterministica si trovi nella configurazione $x^q y$ e sia σ il primo simbolo di y oppure $\square$ se $y = \lambda$ (ovvero, il simbolo σ è quello attualmente scandito dalla testina). Indichiamo con $d_{q,\sigma}$ il **grado di non determinismo dello stato** q **in corrispondenza del simbolo** σ, ovvero il numero di triple contenute in un'etichetta di un arco uscente da q, il cui primo simbolo sia σ. Allora, ognuna delle $d_{q,\sigma}$ configurazioni corrispondenti all'applicazione di tali triple può essere la configurazione prodotta da $x^q y$. Data una macchina di Turing non deterministica T, il **grado di non determinismo** r_T di T denota il massimo numero di scelte non deterministiche disponibili a partire da una qualunque configurazione. Più formalmente, $r_T = \max\{d_{q,\sigma} : q \in Q \wedge \sigma \in \Sigma\}$, dove Q è l'insieme degli stati non finali di T e Σ è il suo alfabeto di lavoro.

Chiaramente, la computazione di una macchina di Turing T non è più rappresentabile mediante una sequenza di configurazioni. Tuttavia, essa può essere vista come un albero, detto **albero delle computazioni**, i cui nodi corrispondono alle configurazioni di T e i cui archi corrispondono alla produzione di una configurazione da parte di un'altra configurazione. Per ciascun nodo dell'albero, il numero massimo dei suoi figli è deteminato dal grado di non determinismo di T.

Ciascuno dei cammini (finiti o infiniti) dell'albero che parte dalla sua radice (ovvero, dalla configurazione iniziale) è detto essere un **cammino di computazione**. Un input x è **accettato** da una macchina di Turing non deterministica T se l'albero delle computazioni corrispondente a x include almeno un **cammino di computazione accettante**, ovvero un cammino di computazione che termini in una configurazione finale. L'insieme delle stringhe accettate da T è detto essere il linguaggio **accettato** da T ed è indicato con $L(T)$.

Esempio 2.4: una macchina di Turing non deterministica

Definiamo una macchina di Turing T non deterministica per cui $L(T) = \{xx : x \in \{0,1\}^*\}$. Tale macchina opera nel modo seguente. Sposta la testina di una posizione a destra: se il simbolo letto non è un □, *non deterministicamente* sceglie di ripetere quest'operazione oppure di proseguire spostando il contenuto del nastro a partire dalla cella scandita di una posizione a destra, inserendo un simbolo `@`, e posizionando la testina sul primo simbolo a sinistra diverso dal simbolo □. Successivamente, T verifica se le due stringhe presenti sul nastro (separate da un simbolo `@`) sono uguali: in tal caso, termina nello stato finale. Supponendo che la stringa in input sia `010010`, l'albero delle computazioni corrispondente è mostrato nella Figura 2.1, in cui per brevità la maggior parte dei cammini deterministici è stata contratta in un unico arco e in cui lo stato finale di T è lo stato `q12`. In questo caso, esiste un cammino accettante (il terzo partendo da sinistra), per cui possiamo concludere che la stringa `010010` è accettata da T: in effetti, tale stringa appartiene al linguaggio L.

Equivalenza tra macchine deterministiche e non deterministiche

Una macchina deterministica può chiaramente essere interpretata come una macchina non deterministica con grado di non determinismo pari a 1. Pertanto, tutto quello che è calcolabile da una macchina di Turing deterministica lo è anche da una non deterministica. Il prossimo teorema mostra che l'uso del non determinismo non aumenta il potere computazionale del modello di calcolo delle macchine di Turing.

Teorema 2.1

Sia T una macchina di Turing non deterministica. Allora, esiste una macchina di Turing deterministica T' tale che $L(T) = L(T')$.

Dimostrazione. L'idea della dimostrazione consiste nel costruire una macchina di Turing T' deterministica che, per ogni stringa x, esegua una visita dell'albero delle computazioni di T con input x (in base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, senza perdita di generalità possiamo assumere che T' abbia a disposizione due nastri). Nel momento in cui T' incontra una configurazione finale di T, T' termina anch'essa in uno stato finale. In caso ciò non accada, se la visita dell'albero si conclude (ovvero, ogni cammino di computazione termina in una configurazione non finale), T' termina in una configurazione non finale, altrimenti T' non termina. Per poter realizzare la visita dell'albero delle computazioni in modo da essere sicuri che se un cammino accettante esiste, allora tale cammino viene, prima o poi, esplorato interamente, non possiamo fare uso della tecnica di visita in profondità, in quanto con tale tecnica la visita rischierebbe di rimanere "incastrata" in un cammino di computazione che non termina. Dobbiamo invece visitare l'albero in ampiezza o per livelli: la realizzazione di una tale visita può essere ottenuta utilizzando uno dei due nastri come se fosse una coda. In particolare, per ogni input x, T' esegue la visita in ampiezza dell'albero delle computazioni di T con input x, facendo uso dei due nastri nel modo seguente.

- Il primo nastro viene utilizzato come una coda in cui le configurazioni di T (codificate in modo opportuno) vengono inserite, man mano che sono generate, e da cui le configurazioni di T sono estratte per generarne di nuove: tale nastro viene inizializzato inserendo nella coda la configurazione iniziale di T con input x.
- Il secondo nastro viene utilizzato per memorizzare la configurazione di T appena estratta dalla testa della coda e per esaminare tale configurazione in modo da generare le (al più r_T) configurazioni da essa prodotte.

Fintanto che la coda non è vuota, T' estrae la configurazione in testa alla coda e la copia sul secondo nastro: se tale configurazione è finale, allora T' termina nel suo unico stato finale. Altrimenti, ovvero se la configurazione estratta dalla coda non è finale, T' calcola le (al più r_T) configurazioni che possono essere prodotte da tale configurazione e le inserisce in coda alla coda. Se, a un certo punto, la coda si svuota, allora tutti i cammini di computazione sono stati esplorati e nessuno di essi è terminato in una

configurazione finale: in tal caso, quindi, T' può terminare in uno stato non finale. Chiaramente, T' accetta la stringa x se e solo se esiste un cammino accettante all'interno dell'albero delle computazioni di T con input x: quindi, $L(T) = L(T')$ e il teorema risulta essere dimostrato. $\diamond$

Come vedremo nel prossimo capitolo e nella prossima parte di queste dispense, l'utilizzo del non determinismo può, invece, aumentare il potere computazionale degli automi a pila e significativamente migliorare le prestazioni temporali di una macchina di Turing, consentendo di decidere in tempo polinomiale linguaggi per i quali non è noto alcun algoritmo polinomiale deterministico.

2.3.2 Linguaggi di tipo 0 e semi-decidibilità

Il prossimo teorema mostra come un qualunque linguaggio generato da una grammatica non limitata sia semi-decidibile da una macchina di Turing non deterministica e multi-nastro e, in base a quanto detto nel primo capitolo e nel paragrafo precedente, da una macchina di Turing deterministica con un singolo nastro (osserviamo, infatti, che la dimostrazione del Teorema 2.1 può essere estesa al caso di macchine di Turing non deterministiche e multi-nastro).

Teorema 2.2

Per ogni linguaggio L di tipo 0, esiste una macchina di Turing non deterministica a due nastri che semi-decide L.

Dimostrazione. Sia $G = (V, T, S, P)$ una grammatica non limitata che genera L. Una macchina di Turing T non deterministica a due nastri che accetta tutte e sole le stringhe di L può operare nel modo seguente.

1. Inizializza il secondo nastro con il simbolo S.
2. Sia ϕ il contenuto del secondo nastro. Per ogni produzione $\alpha \to \beta$ in P, non deterministicamente applica (tante volte quanto è possibile) tale produzione a ϕ, ottenendo la stringa ψ direttamente generabile da ϕ.
3. Se il contenuto del secondo nastro è uguale a x (che si trova sul primo nastro), termina nell'unico stato finale. Altrimenti torna al secondo passo.

Evidentemente, T accetta tutte e sole le stringhe x che possono essere generate a partire da S: il teorema risulta dunque essere dimostrato. $\diamond$

Il prossimo risultato mostra, invece, che ogni linguaggio semi-deciso da una macchina di Turing può essere generato da una grammatica non limitata. L'idea della dimostrazione consiste nel definire la grammatica $G = (V, T, S, P)$ in modo tale che le produzioni della grammatica siano in grado di "simulare" la computazione di T con input una qualunque stringa x. La prova formale del teorema è omessa e ci limitiamo ad osservare che la prova necessita di regole di produzione la cui parte destra è più corta di quella sinistra: in altre parole, la grammatica definita in tale dimostrazione non è di tipo 1.

Teorema 2.3

Se T è una macchina di Turing che semi-decide un linguaggio L, allora L è di tipo 0.

2.4 Linguaggi di tipo 1

LA DIMOSTRAZIONE del Teorema 2.3 utilizza regole di produzione la cui parte destra è più corta di quella sinistra: in altre parole, la grammatica definita in tale dimostrazione non è di tipo 1.

Vediamo nel seguito che grammatiche di tipo 1 sono decidibili. A tale scopo analizziamo la dimostrazione del Teorema 2.2 supponendo che la grammatica di partenza sia una grammatica di tipo 1. Il fatto che in tale grammatica ogni produzione $\alpha \to \beta$ soddisfi il vincolo $|\beta| \geqslant |\alpha|$, ci consente di modificare

la macchina di Turing non deterministica definita nella dimostrazione in modo che essa termini per ogni input. Questo è quanto afferma il prossimo risultato.

Teorema 2.4

Se un linguaggio L è di tipo 1 allora esiste una macchina di Turing non deterministica T a tre nastri tale che $L(T) = L$ e, per ogni input x, ogni cammino di computazione di T con input x termina.

Dimostrazione. Sia $G = (V, T, S, P)$ una grammatica dipendente dal contesto che genera L. Una macchina di Turing T non deterministica a tre nastri che termina per ogni input e che accetta tutte e sole le stringhe di L opera in modo simile alla macchina non deterministica definita nella dimostrazione del Teorema 2.2, ma ogni qualvolta cerca di applicare (in modo non deterministico) una produzione di G, verifica se il risultato dell'applicazione sia una stringa di lunghezza minore oppure uguale alla lunghezza della stringa x di input. Se così non è, allora T può terminare in uno stato non finale, in quanto siamo sicuri (in base alla proprietà delle grammatiche di tipo 1) che non sarà possibile da una stringa di lunghezza superiore a $|x|$ generare x. Rimane anche da considerare la possibilità che la forma sentenziale generata da T rimanga sempre della stessa lunghezza a causa dell'applicazione ciclica della stessa sequenza di produzioni: anche in questo caso, dobbiamo fare in modo che T termini in uno stato non finale. A tale scopo, osserviamo che il numero di possibili forme sentenziali distinte di lunghezza k, con $1 \leqslant k \leqslant |x|$, è pari a $|V \cup T|^k$. La macchina T può evitare di entrare in un ciclo senza termine, mantenendo sul terzo nastro un contatore che viene inizializzato ogni qualvolta una forma sentenziale di lunghezza k viene generata sul secondo nastro per la prima volta. Per ogni produzione che viene applicata, se la lunghezza della forma sentenziale generata non aumenta, allora il contatore viene incrementato di 1: se il suo valore supera $|V \cup T|^k$, allora siamo sicuri che la computazione non avrà termine e, quindi, possiamo far terminare T in uno stato non finale. In conclusione, ogni cammino di computazione della macchina T termina e tutte e sole le stringhe generate da G sono accettate da almeno un cammino di computazione di T. Quindi, $L(T) = L$ e il teorema risulta essere dimostrato. ◇

2.5 Conclusioni

Quanto esposto in questo capitolo ci consente di riassumere nella seguente tabella, la classificazione dei linguaggi in base alla loro tipologia, al tipo di grammatiche che li generano e in base al modello di calcolo corrispondente.

Tipo di linguaggio	**Tipo di produzioni**	**Modello di calcolo**
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	

I prossimi due capitoli ci consentiranno di completare tale tabella, determinando i modelli di calcolo corrispondenti ai linguaggi regolari e a quelli liberi da contesto.

Esercizi

Esercizio 2.1. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti un numero pari di simboli `1`.

Esercizio 2.2. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno due simboli `0` e almeno un simbolo `1` è regolare.
Esercizio 2.3. Dimostrare che il linguaggio costituito da tutte e sole le stringhe binarie contenenti un numero pari di simboli `0` oppure esattamente due simboli `1` è regolare.
Esercizio 2.4. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno tre simboli `1`.
Esercizio 2.5. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie del tipo 0^n12^n (n 0 seguiti da un 1 seguito da n 2).
Esercizio 2.6. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti lo stesso numero di `1` e di `0`.
Esercizio 2.7. Completare la dimostrazione del Teorema **??**.

Capitolo 3

Linguaggi regolari

SOMMARIO

Una delle principali componenti di un compilatore è l'analizzatore lessicale, il cui compito è quello di analizzare un programma per identificarne al suo interno le unità significanti, anche dette token. La risorsa principale nello sviluppo di un analizzatore lessicale sono le espressioni regolari, che, come vedremo in questo capitolo, corrispondono in modo biunivoco agli automi a stati finiti e alle grammatiche regolari. Al termine del capitolo, presenteremo anche il principale strumento per dimostrare che un linguaggio non è regolare, ovvero il pumping lemma per linguaggi regolari.

3.1 Introduzione

UN **COMPILATORE** è un programma che traduce un programma scritto in un linguaggio ad alto livello (come JAVA) in uno scritto in linguaggio macchina. Il linguaggio macchina è il linguaggio originale del calcolatore sul quale il programma deve essere eseguito ed è letteralmente l'unico linguaggio per cui è appropriato affermare che il calcolatore lo "comprende". Agli albori del calcolo automatico, le persone programmavano in binario e scrivevano sequenze di bit, ma ben presto furono sviluppati primitivi traduttori, detti "assembler", che permettevano al programmatore di scrivere istruzioni in uno specifico codice mnemonico anzichè mediante sequenze binarie. Generalmente, un'istruzione assembler corrisponde a una singola istruzione in linguaggio macchina. Linguaggi come JAVA, invece, sono noti come linguaggi ad alto livello ed hanno la proprietà che una loro singola istruzione corrisponde a più di un'istruzione in linguaggio macchina. Il vantaggio principale dei linguaggi ad alto livello è la produttività. È stato stimato che il programmatore medio può produrre 10 linee di codice senza errori in una giornata di lavoro (il punto chiave è "senza errori": possiamo tutti scrivere enormi quantità di codice in minor tempo, ma il tempo addizionale richiesto per verificare e correggere riduce tale quadro drasticamente). Si è anche osservato che questo dato è essenzialmente indipendente dal linguaggio di programmazione utilizzato. Poichè una tipica istruzione in linguaggio ad alto livello può corrispondere a 10 istruzioni in linguaggio assemblatore, ne segue che approssimativamente possiamo essere 10 volte più produttivi se programmiamo, ad esempio, in JAVA invece che in linguaggio assembler.

Nel seguito, il linguaggio ad alto livello che il compilatore riceve in ingresso è chiamato *linguaggio sorgente*, ed il programma in linguaggio sorgente che deve essere compilato è detto *codice sorgente*. Il particolare linguaggio macchina che viene generato è il *linguaggio oggetto*, e l'uscita del compilatore è il *codice oggetto*. Abbiamo detto che il linguaggio oggetto è un linguaggio macchina ma non è sempre detto che tale macchina obiettivo sia reale. Ad esempio, il linguaggio JAVA viene normalmente compilato nel linguaggio di una macchina virtuale, detto "byte code". In tal caso, il codice oggetto va ulteriormente compilato o interpretato prima di ottenere il linguaggio macchina. Questo può sembrare un passo seccante ma il linguaggio byte code è relativamente facile da interpretare e l'utilizzo di una macchina virtuale consente di costruire compilatori per JAVA che siano indipendenti dalla piattaforma finale su cui il programma deve essere eseguito. Non a caso, JAVA si è imposto negli ultimi anni come linguaggio di programmazione per sviluppare applicazioni da diffondere attraverso reti di calcolatori.

Un compilatore è un programma molto complesso e, come molti programmi complessi, è realizzato mediante un numero separato di parti che lavorano insieme: queste parti sono note come *fasi* e sono cinque.

- Analisi lessicale: in questa fase, il compilatore scompone il codice sorgente in unità significanti dette **token**. Questo compito è relativamente semplice per la maggior parte dei moderni linguaggi di programmazione, poichè il programmatore deve separare molte parti dell'istruzione con spazi o caratteri di tabulazione; questi spazi rendono più facile per il compilatore determinare dove un token finisce ed il prossimo ha inizio. Di questa fase ci occuperemo in questo capitolo e gli strumenti di cui faremo uso sono le espressioni regolari e gli automi a stati finiti.

- Analisi sintattica: in questa fase, il compilatore determina la struttura del programma e delle singole istruzioni. Di questa fase ci occuperemo nel prossimo capitolo, in cui vedremo che la costruzione di analizzatori sintattici poggia, in generale, sulle tecniche e i concetti sviluppati nella teoria dei linguaggi formali e, in particolare, sulle grammatiche generative libere da contesto.

- Generazione del codice intermedio: in questa fase, il compilatore crea una rappresentazione interna del programma che riflette l'informazione non coperta dall'analizzatore sintattico.

- Ottimizzazione: in questa fase, il compilatore identifica e rimuove operazioni ridondanti dal codice intermedio.

- Generazione del codice oggetto: in questa fase, infine, il compilatore traduce il codice intermedio ottimizzato nel linguaggio della macchina obiettivo.

Le ultime tre fasi vanno ben al di là degli obiettivi di queste dispense: rimandiamo il lettore ai tanti libri disponibili sulla progettazione e la realizzazione di compilatori (uno per tutti il famoso *dragone rosso* di Aho, Sethi e Ullman).

3.2 Analisi lessicale di linguaggi di programmazione

IL PRINCIPALE compito dell'analizzatore lessicale consiste nello scandire la sequenza del codice sorgente e scomporla in parti significanti, ovvero nei token di cui abbiamo parlato in precedenza. Per esempio, data l'istruzione JAVA

```
if (x == y*(b - a)) x = 0;
```

l'analizzatore lessicale deve essere in grado di isolare la parole chiave `if`, gli identificatori `x`, `y`, `b` ed `a`, gli operatori ==, $\star$, $-$ e =, le parentesi, il letterale `0` ed il punto e virgola finale. In questo capitolo svilupperemo un modo sistematico per estrarre tali token dalla sequenza sorgente (in realtà l'analizzatore lessicale può prendersi cura anche di altre cose, come, ad esempio, la rimozione dei commenti, la conversione dei caratteri, la rimozione degli spazi bianchi e l'interpretazione delle direttive di compilazione).

Abbiamo già osservato come nell'esempio precedente siano presenti quattro identificatori. Dal punto di vista sintattico, tuttavia, gli identificatori giocano tutti lo stesso ruolo ed è sufficiente dire in qualche modo che il prossimo oggetto nel codice sorgente è un identificatore (d'altro canto, sarà chiaramente importante, in seguito, essere in grado di distinguere i vari identificatori). Analogamente, dal punto di vista sintattico, un letterale intero è equivalente ad un altro letterale intero: in effetti, la struttura grammaticale dell'istruzione nel nostro esempio non cambierebbe se `0` fosse sostituito con `1` oppure con `1000`. Così tutto quello che in qualche modo dobbiamo dire è di aver trovato un letterale intero (di nuovo, in seguito, dovremo distinguere tra i vari letterali interi, poiché essi sono funzionalmente differenti anche se sintatticamente equivalenti).

Trattiamo questa distinzione nel modo seguente: il tipo generico, passato all'analizzatore sintattico, è detto token e le specifiche istanze del tipo generico sono dette **lessemi**. Possiamo dire che un token è il nome di un insieme di lessemi che hanno lo stesso significato grammaticale per l'analizzatore sintattico. Così nel nostro esempio abbiamo quattro istanze (ovvero i lessemi `x`, `y`, `b` ed `a`) del token "identificatore", abbreviato come `id`, e un'istanza (ovvero il lessema `0`) del token "letterale intero", abbreviato come `int`. In molti casi, un token può avere una sola valida istanziazione: per esempio, le parole chiave non possono essere raggruppate in un solo token, in quanto hanno diversi significati per l'analizzatore sintattico. In questi casi, dunque, il token coincide con il lessema (lo stesso accade, nell'esempio precedente, con i token `(`, `)`, `==`, `*`, `-` e `;`). In conclusione, nel nostro esempio, la sequenza di token trasmessa dall'analizzatore lessicale a quello sintattico è la seguente:

```
if (id == id * (id - id)) id = int;
```

L'analizzatore lessicale deve quindi fare due cose: deve isolare i token ed anche prendere nota dei particolari lessemi. In realtà, l'analizzatore lessicale ha anche un compito aggiuntivo: quando un identificatore viene trovato, deve dialogare con il gestore della **tabella dei simboli**. Se l'identificatore viene dichiarato, un nuovo elemento verrà creato, in corrispondenza del lessema, nella tabella stessa. Se l'identificatore viene invece usato, l'analizzatore deve chiedere al gestore della tabella dei simboli di verificare che esista un elemento per il lessema nella tabella.

3.3 Automi a stati finiti

CHI SVILUPPA l'analizzatore lessicale definisce i token del linguaggio mediante espressioni regolari. Queste costituiscono una notazione compatta che indica quali caratteri possono far parte dei lessemi appartenenti a un particolare token e come devono essere messi in sequenza. Ad esempio, l'espressione regolare:

$$(0+\{1,2,3,4,5,6,7,8,9\}\{0,1,2,3,4,5,6,7,8,9\}^*)(\lambda+1+L)$$

specifica il token letterale intero decimale in JAVA. L'analizzatore lessicale, tuttavia, viene meglio realizzato come un **automa a stati finiti**. Abbiamo già introdotto tali automi nella prima parte delle dispense: si tratta, infatti, di macchine di Turing che possono solo leggere e spostarsi a destra e che al momento in cui incontrano il primo simbolo $\square$ devono terminare accettando o rigettando la stringa di input.

Esempio 3.1: un automa a stati finiti per gli identificatori

Un identificatore consiste di una lettera o di un segno di sottolineatura seguito da una combinazione di lettere, segno di sottolineatura e cifre. In Figura 3.1 mostriamo un automa a stati finiti che identifica un token di questo tipo (per semplicità, abbiamo indicato con `L` e `C` l'insieme delle lettere e delle cifre, rispettivamente).

Osserviamo che, come mostrato nell'esempio precedente, quando si descrive un automa a stati finiti non è necessario specificare, nelle etichette degli archi, il simbolo scritto e il movimento, in quanto in un tale automa la scrittura non è consentita e l'unico movimento possibile è quello a destra. Osserviamo inoltre che, essendo il criterio di terminazione applicabile solo nel momento in cui l'intera stringa in input sia stata letta, uno stato finale può avere transizioni in uscita (come nel caso dello stato `q1` dell'esempio precedente). Infine, nel seguito indicheremo con $L(T)$ il linguaggio accettato dall'automa a stati finiti T, ovvero l'insieme delle stringhe x per cui esiste un cammino nel grafo, che parte dallo stato iniziale e finisce in uno stato finale, le etichette dei cui archi formano x. Due automi a stati finiti T_1 e T_2 sono detti essere **equivalenti** se $L(T_1) = L(T_2)$.

Figura 3.1: un automa per il token identificatore.

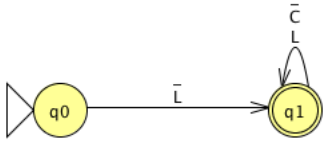
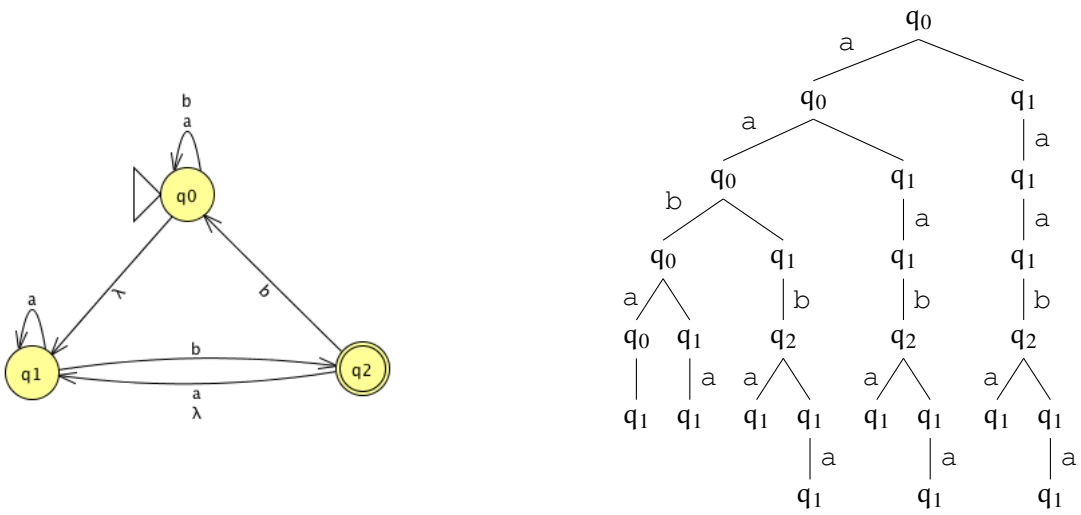


Figura 3.2: un automa a stati finiti non deterministico e un suo albero delle computazioni.



3.3.1 Automi a stati finiti non deterministici

Analogamente a quanto abbiamo fatto con le macchine di Turing nel precedente capitolo, possiamo modificare la definizione di automa a stati finiti consentendo un grado di non prevedibilità delle sue transizioni. Più precisamente, la transizione di un automa a stati finiti **non deterministico** può portare l'automa stesso in più stati diversi tra di loro. Come già osservato con le macchine di Turing non deterministiche, la computazione di un automa a stati finiti non deterministico corrisponde a un albero di possibili cammini di computazione: una stringa x è accettata dall'automa se tale albero include almeno un cammino che, dallo stato iniziale, conduce a uno stato finale e le etichette dei cui archi formino x. Nel caso degli automi a stati finiti, tuttavia, il non determinismo è generalmente esteso includendo la possibilità di λ-**transizioni**, ovvero transizioni che avvengono senza leggere alcun simbolo di input.

Esempio 3.2: un automa a stati finiti con transizioni nulle

Nella parte destra della Figura 3.2 è mostrato un automa non deterministico con λ-transizioni. L'albero delle computazioni corrispondente alla stringa `aaba` è mostrato nella parte destra della figura: come si può vedere, la stringa non viene accettata.

3.3.2 Automi deterministici e non deterministici

Nel capitolo precedente abbiamo dimostrato che una macchina di Turing non deterministica può essere simulata da una deterministica, facendo uso della tecnica di visita in ampiezza dell'albero delle computazioni. Chiaramente, tale tecnica non è realizzabile mediante un automa a stati finiti. Si potrebbe quindi

pensare che gli automi a stati finiti non deterministici siano computazionalmente più potenti di quelli deterministici. Il prossimo risultato mostra come ciò non sia vero: l'idea alla base della dimostrazione consiste nello sfruttare il fatto che, per quanto le transizioni tra i singoli stati siano imprevedibili, le transizioni tra sottoinsiemi di stati sono al contrario univocamente determinate.

Teorema 3.1

Sia T un automa a stati finiti non deterministico senza λ-transizioni. Allora, esiste un automa a stati finiti T' equivalente a T.

Dimostrazione. La dimostrazione procede definendo uno dopo l'altro gli stati di T' sulla base degli stati già definiti e delle transizioni di T (l'alfabeto di lavoro di T' sarà uguale a quello di T): in particolare, ogni stato di T' denoterà un sottoinsieme degli stati di T. Lo stato iniziale di T' è lo stato $\{q_0\}$ dove q_0 è lo stato iniziale di T. La costruzione delle transizioni e degli altri stati di T' procede nel modo seguente. Sia $Q = \{s_1, \ldots, s_k\}$ uno stato di T' per cui non è ancora stata definita la transizione corrispondente a un simbolo σ dell'alfabeto di lavoro di T e, per ogni i con $i = 1, \ldots, k$, sia $N(s_i, \sigma)$ l'insieme degli stati raggiungibili da s_i leggendo il simbolo σ (osserviamo che S potrebbe anche essere l'insieme vuoto). Definiamo allora $S = \bigcup_{i=1}^{k} N(s_i, \sigma)$ e introduciamo la transizione di T' dallo stato Q allo stato S leggendo il simbolo σ. Inoltre, aggiungiamo S all'insieme degli stati di T', nel caso non ne facesse già parte. Al termine di questo procedimento, identifichiamo gli stati finali di T' come quegli stati corrispondenti a sottoinsiemi contenenti almeno uno stato finale di T. È facile dimostrare che una stringa x è accettata da T se e solo se è accettata anche da T', ovvero $L(T) = L(T')$. ◇

Esempio 3.3: trasformazione da automa non deterministico ad automa deterministico

Consideriamo l'automa non deterministico T definito dalla seguente tabella delle transizioni, in cui, per ogni riga, specifichiamo l'insieme degli stati in cui l'automa può andare leggendo uno specifico simbolo a partire da un determinato stato (assumiamo che `q0` sia lo stato iniziale e `q1` quello finale, come mostrato nella parte sinistra della Figura 3.3).

stato	simbolo	insieme di stati
q0	0	{q0, q1}
q0	1	{q1}
q1	1	{q0, q1}

L'automa deterministico T' equivalente a T include lo stato $\{q0\}$. Relativamente a esso abbiamo che $N(q0, 0) = \{q0, q1\}$ e che $N(q0, 1) = \{q1\}$. Questi due stati non sono ancora stati generati: li aggiungiamo all'insieme degli stati di T' e definiamo una transizione verso di essi a partire dallo stato $\{q0\}$ leggendo il simbolo `0` e il simbolo `1`, rispettivamente. Dobbiamo ora definire la transizione a partire da $\{q0, q1\}$ leggendo il simbolo `0`. In questo caso, $N(q0, 0) = \{q0, q1\}$ e $N(q1, 0) = \{\}$: l'unione di questi due insiemi è uguale a $\{q0, q1\}$, che già esiste nell'insieme degli stati di T'. È sufficiente quindi definire la transizione da $\{q0, q1\}$ a se stesso leggendo `0`. Procedendo, abbiamo che $N(q0, 1) = \{q1\}$ e che $N(q1, 1) = \{q0, q1\}$: non dobbiamo aggiungere nessuno stato ma solo la transizione da $\{q0, q1\}$ a se stesso leggendo `1`. In modo analogo possiamo calcolare le transizioni a partire dallo stato $\{q1\}$: la tabella finale delle transizioni di T' è la seguente in cui lo stato iniziale `Q0` corrisponde all'insieme $\{q0\}$, lo stato finale `Q1` all'insieme $\{q0,q1\}$ e lo stato finale `Q2` all'insieme $\{q1\}$ (si veda la parte destra della Figura 3.3).

stato	simbolo	stato
Q0	0	Q1
Q0	1	Q2
Q1	0	Q1
Q1	1	Q1
Q2	1	Q1

La dimostrazione del Teorema 3.1 può essere opportunamente modificata in modo da estendere il risultato al caso in cui l'automa a stati finiti non deterministico contenga λ-transizioni. In effetti, abbiamo

Figura 3.3: trasformazione di un automa a stati finiti non deterministico in uno deterministico.

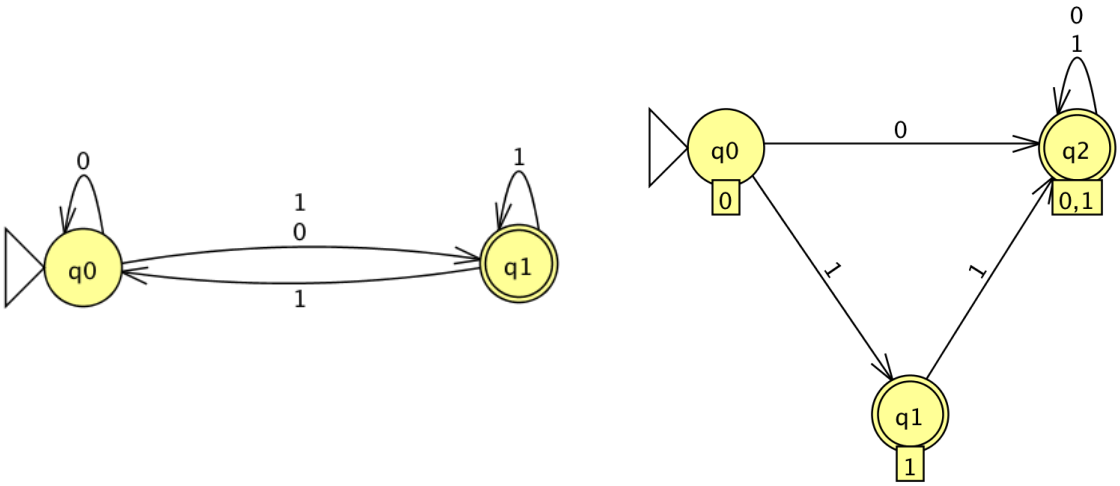
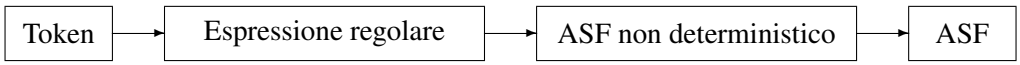


Figura 3.4: dal token all'automa a stati finiti equivalente



bisogno di una sola modifica che consiste, per lo stato iniziale di T' e per ogni nuovo stato, nell'includere tutti gli stati che possono essere raggiunti da tale stato mediante una o più λ-transizioni. Tale operazione è anche detta λ-**chiusura** di uno stato. Formalmente, la λ-*estensione* di un insieme di stati A di un automa non deterministico è definita come l'insieme degli stati che sono collegati direttamente a uno stato di A mediante una transizione con etichetta λ. La λ-chiusura di A si ottiene allora calcolando ripetutamente la λ-estensione di A fino a quando non vengono aggiunti nuovi stati. L'algoritmo di costruzione mediante sottoinsiemi dell'automa T' descritto nella dimostrazione del Teorema 3.1 può dunque essere modificato nel modo seguente: lo stato iniziale di T' corrisponde alla λ-chiusura di $\{q_0\}$ e lo stato S è definito come la λ-chiusura di $\bigcup_{i=1}^{k} N(s_i, \sigma)$.

3.4 Espressioni regolari

NEL PARAGRAFO precedente abbiamo visto come gli automi a stati finiti non deterministici non sono computazionalmente più potenti di quelli deterministici. In tal caso, a che cosa servono gli automi a stati finiti non deterministici? La risposta è che il nostro scopo consiste nel trovare un modo di costruire le tabelle degli stati di automi a stati finiti a partire dalle definizioni dei token. In questo paragrafo vedremo come le espressioni regolari risultino un modo molto conveniente ed economico di specificare i token. Vedremo anche che è relativamente semplice, anche se laborioso, trovare un automa a stati finiti non deterministico equivalente a una data espressione regolare: da quest'automa possiamo costruirne uno deterministico equivalente che può quindi essere usato dall'analizzatore lessicale. Il processo di generazione della tabella degli stati finale consiste quindi dei passi mostrati in Figura 3.4. Noi siamo partiti dalla fine della catena mostrata in figura e stiamo ora per tornare all'inizio. L'intero processo è laborioso ma non richiede molto esercizio mentale e può quindi essere delegato a un calcolatore.

Definizione 3.1: espressioni regolari

L'insieme delle espressioni regolari su di un alfabeto Σ è definito induttivamente come segue.

- Ogni carattere in Σ è un'espressione regolare.
- λ è un'espressione regolare.
- Se R ed S sono due espressioni regolari, allora
  - La *concatenazione* $R \cdot S$ (o, semplicemente, RS) è un'espressione regolare.
  - La *selezione* $R + S$ è un'espressione regolare.
  - La *chiusura di Kleene* R^* è un'espressione regolare.
- Solo le espressioni formate da queste regole sono regolari.

Nel valutare un'espressione regolare supporremo nel seguito che la chiusura di Kleene abbia priorità maggiore mentre la selezione abbia priorità minore: le parentesi verrano anche usate per annullare le priorità nel modo usuale.

Un'espressione regolare R genera un linguaggio $L(R)$ definito anch'esso in modo induttivo nel modo seguente.

- Se $R = a \in \Sigma$, allora $L(R) = \{a\}$.
- Se $R = \lambda$, allora $L(R) = \{\lambda\}$.
- Se $R = S_1 \cdot S_2$, allora $L(R) = \{xy : x \in L(S_1) \wedge y \in L(S_2)\}$.
- Se $R = S_1 + S_2$, allora $L(R) = L(S_1) \cup L(S_2)$.
- Se $R = S^*$, allora $L(R) = \{x_1 x_2 \dots x_n : n \geqslant 0 \wedge x_i \in L(S)\}$.

Osserviamo che la chiusura di Kleene consente zero concatenazioni, per cui il linguaggio generato contiene la stringa vuota.

Due espressioni regolari R e S sono equivalenti se $L(R) = L(S)$. Alcune equivalenze, la cui dimostrazione è lasciata come esercizio (si veda l'Esercizio 3.3) sono utili talvolta per analizzare espressioni regolari.

Teorema 3.2

Se R, S e T sono tre espressioni regolari, allora le seguenti affermazioni sono vere.

Associatività $R(ST)$ è equivalente a $(RS)T$.

Associatività $R + (S + T)$ è equivalente a $(R + S) + T$.

Commutatività $R + S$ è equivalente a $S + R$.

Distributività $R(S + T)$ è equivalente a $RS + RT$.

Identità $R\lambda$ ed λR sono equivalenti a R.

Osserviamo che la concatenazione non è commutativa in quanto, in generale, $L(RS) \neq L(SR)$ (si veda l'Esercizio 3.4).

Le espressioni regolari sono in grado di generare solo un insieme limitato di linguaggi ma sono abbastanza potenti da poter essere usate per definire i token. Abbiamo già visto, infatti, che un token può essere visto come un linguaggio che include i suoi lessemi: rappresentando il token mediante un

espressione regolare, definiamo esattamente in che modo i lessemi sono riconosciuti come elementi del linguaggio. Abbiamo già osservato, ad esempio, come la seguente espressione regolare

$$(0+\{1,2,3,4,5,6,7,8,9\}\{0,1,2,3,4,5,6,7,8,9\}^*)(\lambda+\mathtt{l}+\mathtt{L})$$

specifichi il token letterale intero decimale in JAVA. Infatti, tale token è definito come un numerale decimale eventualmente seguito dal suffisso `l` oppure `L` allo scopo di indicare se la rappresentazione deve essere a 64 bit. Un numerale decimale, a sua volta, può essere uno `0` oppure una cifra da `1` a `9` eventualmente seguita da una o più cifre da `0` a `9`.

3.4.1 Espressioni regolari ed automi a stati finiti

Siamo ora in grado di descrivere come può essere realizzato il secondo passo della catena che porta dalla definizione del token al corrispondente automa a stati finiti non deterministico.

Teorema 3.3

Per ogni espressione regolare R esiste un automa a stati finiti non deterministico T tale che $L(R) = L(T)$.

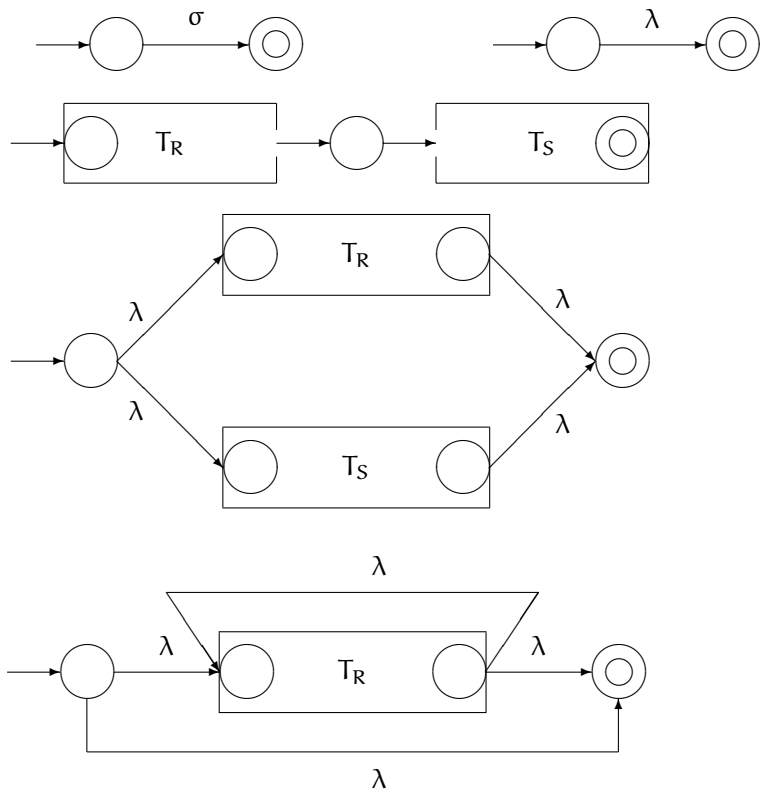
Dimostrazione. La costruzione si basa sulla definizione induttiva delle espressioni regolari fornendo una macchina oppure un'interconnessione di macchine corrispondente a ogni passo della definizione. La costruzione, che tra l'altro assicura che l'automa ottenuto avrà un solo stato finale diverso dallo stato iniziale e senza transizioni in uscita, è la seguente.

- La macchina mostrata in alto a sinistra della Figura 3.5 accetta il carattere $\sigma \in \Sigma$ mentre quella mostrata in alto a destra accetta λ.
- Date due espressioni regolari R e S, la macchina mostrata nella seconda riga della Figura 3.5 accetta $L(RS)$ dove T_R e T_S denotano due macchine che decidono, rispettivamente, $L(R)$ ed $L(S)$ e lo stato finale di T_R è stato fuso con lo stato iniziale di T_S in un unico stato.
- Date due espressioni regolari R e S, la macchina mostrata nella terza riga della Figura 3.5 accetta $L(R+S)$ dove T_R e T_S denotano due macchine che riconoscono, rispettivamente, $L(R)$ ed $L(S)$, un nuovo stato iniziale è stato creato, due λ-transizioni da questo nuovo stato agli stati iniziali di T_R ed T_S sono state aggiunte, un nuovo stato finale è stato creato e due λ-transizioni dagli stati finali di T_R ed T_S a questo nuovo stato sono state aggiunte.
- Data un'espressione regolare R, la macchina mostrata nella quarta riga della Figura 3.5 accetta $L(R^*)$ dove T_R denota una macchina che riconosce $L(R)$, un nuovo stato iniziale e un nuovo stato finale sono stati creati, due λ-transizioni dal nuovo stato iniziale al nuovo stato finale e allo stato iniziale di T_R sono state aggiunte e due λ-transizioni dallo stato finale di T_R allo stato iniziale di T_R e al nuovo stato finale sono state aggiunte.

È facile verificare che la costruzione sopra descritta permette di definire un automa a stati finiti non deterministico equivalente a una specifica espressione regolare. ◇

Dall'automa a stati finiti non deterministico ottenuto dalla costruzione precedente possiamo ottenere un automa a stati finiti equivalente all'espressione regolare di partenza che può quindi essere incorporato nell'analizzatore lessicale. Ciò completa dunque la nostra catena: sappiamo ora come ottenere, partendo da un'espressione regolare, l'equivalente automa a stati finiti facendo uso dell'automa a stati finiti non deterministico nel passo intermedio. Il ben noto programma `Lex` genera una tabella degli stati per un analizzatore lessicale essenzialmente in questo modo: l'utente specifica i token mediante espressioni regolari, `Lex` calcola gli equivalenti automi a stati finiti non deterministici e da quelli genera la tabella degli stati per gli automi a stati finiti. In realtà, quando costruiamo un automa a stati finiti equivalente a

Figura 3.5: da espressioni regolari ad automi non deterministici



uno non deterministico (a sua volta ottenuto da un'espressione regolare) si crea generalmente un'esplosione esponenziale di stati, per la maggior parte ridondanti: sebbene ciò sia al di là degli scopi di queste dispense, è bene sapere che è possibile identificare ed eliminare questi stati non necessari attraverso un processo di minimizzazione degli stati, di cui lo stesso programma `Lex` fa uso.

Osserviamo inoltre che la catena mostrata nella Figura 3.4 può in realtà essere trasformata in un ciclo dimostrando che a ogni automa a stati finiti corrisponde un'espressione regolare a esso equivalente: la dimostrazione di tale risultato va comunque al di là degli obiettivi di queste dispense.

Un esempio

Abbiamo già osservato che il token letterale intero decimale in JAVA può essere definito mediante la seguente espressione regolare:

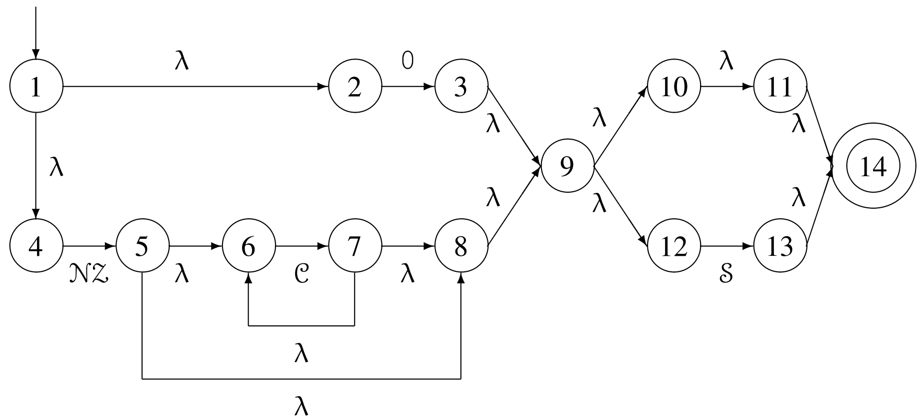
$$(0+\{1,2,3,4,5,6,7,8,9\}\{0,1,2,3,4,5,6,7,8,9\}^*)(\lambda+\mathtt{l}+\mathtt{L}).$$

Nel seguito indicheremo con $\mathcal{C}$ l'insieme delle cifre da `0` a `9`, con $\mathcal{NZ}$ l'insieme delle cifre da `1` a `9` e con $\mathcal{S}$ l'insieme $\{\mathtt{l}, \mathtt{L}\}$. Quindi l'espressione regolare associata al token letterale intero decimale diviene la seguente:

$$(0|\mathcal{NZ}(\mathcal{C})^*)(\epsilon|\mathcal{S}).$$

Da quest'espressione regolare intendiamo ora ottenere l'equivalente automa a stati finiti mediante le tecniche viste nei paragrafi precedenti. Anzitutto, costruiamo l'automa a stati finiti non deterministico

Figura 3.6: automa non deterministico per il token letterale intero decimale



T equivalente all'espressione regolare facendo uso della dimostrazione del Teorema 3.3: tale automa è mostrato in Figura 3.6.

Applicando poi la trasformazione di un automa non deterministico in uno deterministico mediante la tecnica utilizzata nella dimostrazione del Teorema 3.1 (estesa in modo da gestire anche le λ-transizioni), otteniamo l'automa a stati finiti T′ definito dalla seguente tabella delle transizioni:

stato	**simbolo**	**stato**
Q0	0	Q1
Q0	$\mathcal{NZ}$	Q2
Q1	$\mathcal{S}$	Q3
Q2	0	Q4
Q2	$\mathcal{NZ}$	Q4
Q2	$\mathcal{S}$	Q3
Q4	0	Q4
Q4	$\mathcal{NZ}$	Q4
Q4	$\mathcal{S}$	Q3

In tale tabella, lo stato iniziale Q0 corrisponde all'insieme $\{1,2,4\}$ degli stati di T, mentre gli stati Q1, Q2, Q3 e Q4 sono tutti finali e corrispondono rispettivamente agli insiemi $\{3,9,10,11,12,14\}$, $\{5,6,8,9,10,11,12,14\}$, $\{13,14\}$ e $\{6,7,8,9,10,11,12,14\}$ degli stati di T. Osserviamo come gli stati Q2 e Q4 siano equivalenti: eliminando lo stato Q4 otteniamo l'automa a stati finiti finale la cui tabella degli stati è la seguente:

stato	**simbolo**	**stato**
Q0	0	Q1
Q0	$\mathcal{NZ}$	Q2
Q1	$\mathcal{S}$	Q3
Q2	0	Q2
Q2	$\mathcal{NZ}$	Q2
Q2	$\mathcal{S}$	Q3

Tale automa è mostrato anche in Figura 3.7.

Figura 3.7: automa deterministico per il token letterale intero decimale

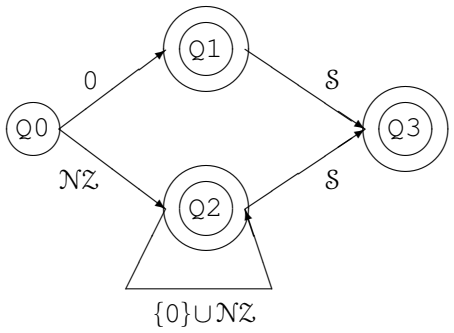
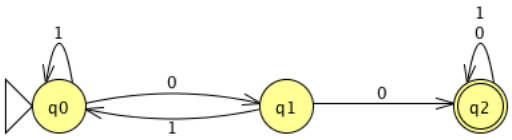


Figura 3.8: un automa a stati finiti e la corrispondente grammatica regolare.



$$Q_0 \to 1Q_0 \quad Q_0 \to 0Q_1$$
$$Q_1 \to 1Q_0 \quad Q_1 \to 0Q_2 \quad Q_1 \to 0$$
$$Q_2 \to 0Q_2 \quad Q_2 \to 0 \quad Q_2 \to 1Q_2 \quad Q_2 \to 1$$

3.5 Automi a stati finiti e grammatiche regolari

COME ABBIAMO visto nel capitolo precedente, le grammatiche regolari o di tipo 3 sono grammatiche le cui regole di produzione sono **lineari a destra**, ovvero del tipo $X \to a$ oppure del tipo $X \to aY$. Il prossimo risultato mostra come tali grammatiche siano equivalenti agli automi a stati finiti e, quindi, all'espressioni regolari.

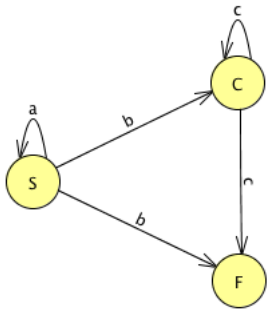
Teorema 3.4

Un linguaggio L è regolare se e solo se esiste un automa a stati finiti che decide L.

Dimostrazione. Sia T un automa a stati finiti e sia $L = L(T)$ il linguaggio deciso da T. Costruiamo ora una grammatica regolare G che genera tutte e sole le stringhe di L. Tale grammatica avrà un simbolo non terminale per ogni stato di T: il simbolo iniziale sarà quello corrispondente allo stato iniziale. Per ogni transizione che dallo stato q fa passare l'automa nello stato p leggendo il simbolo a, la grammatica include la regola di produzione $Q \to aP$, dove Q e P sono i due simboli non terminali corrispondenti agli stati q e p, rispettivamente. Inoltre, se p è uno stato finale, allora la grammatica include anche la transizione $Q \to a$. È facile verificare che $L(T) = L(G)$ (si veda ad esempio, la Figura 3.8). Viceversa, supponiamo che G sia una grammatica regolare e che $L = L(G)$ sia il linguaggio generato da G. Definiamo un automa a stati finito non deterministico T tale che $L(T) = L$: in base a quanto visto nei paragrafi precedenti, questo dimostra che esiste un automa a stati finiti equivalente a G. L'automa T ha uno stato per ogni simbolo non terminale di G: lo stato iniziale sarà quello corrispondente al simbolo iniziale di G. Per ogni produzione del tipo $X \to aY$ di G, T avrà una transizione dallo stato corrispondente a X allo stato corrispondente a Y leggendo il simbolo a (in questo modo, possiamo avere che T sia non deterministico). Inoltre, per ogni produzione del tipo $X \to a$ di G, T avrà una transizione dallo stato corrispondente a X all'unico stato finale F leggendo il simbolo a. Di nuovo, è facile verificare che $L(T) = L(G)$ (si veda ad esempio, la Figura 3.9). ◇

Figura 3.9: una grammatica regolare e il corrispondente automa a stati finiti.

$S \to aS \quad S \to bC \quad S \to b$
$C \to cC \quad C \to c$



3.5.1 Linguaggi non regolari

Il teorema precedente, oltre ad essere di per sé interessante, fornisce un'immediata intuizione su come cercare di dimostrare che esistono linguaggi non regolari. In effetti, se un linguaggio L è deciso da un automa a stati finiti T, quest'ultimo deve necessariamente avere un numero finito n di stati. Ciò significa che tutte le stringhe in L di lunghezza superiore a n devono corrispondere a una computazione di T che visita lo stesso stato più di una volta. Quindi, T deve contenere un ciclo, ovvero un cammino da uno stato q ad altri stati che ritorna poi in q: tale ciclo può essere "pompato" a piacere tante volte quante lo desideriamo, producendo comunque sempre stringhe in L. Formalizzando tale ragionamento, possiamo dimostrare il seguente risultato.

Lemma 3.1

Se L è un linguaggio infinito regolare, allora esiste un numero intero $n_L > 0$, tale che, per ogni stringa $x \in L$ con $|x| > n_L$, x può essere decomposta nella concatenazione di tre stringhe y, v e z per cui valgono le seguenti affermazioni.

1. $|v| > 0$.
2. $|yv| < n_L$.
3. Per ogni $i \geqslant 0$, $yv^i z \in L$.

Dimostrazione. Sia T un automa a stati finiti tale che $L = L(T)$: poniamo n_L uguale al numero di stati di T. Sia x una stringa in L di lunghezza n maggiore di n_L: consideriamo la computazione di T con input x. Sia $q_0, q_1, q_2, \ldots, q_n$ la sequenza di stati attraverso cui passa tale computazione (q_0 è lo stato iniziale, mentre q_n è uno degli stati finali): poiché $n > n_L$, tale sequenza deve contenere uno stato ripetuto nei primi $n_L + 1$ elementi. Sia q_j il primo stato ripetuto e supponiamo che si ripeta dopo $m > 0$ passi (ovvero, $q_j = q_{j+m}$ con $j + m \leqslant n_L$): definiamo allora y come la sequenza delle etichette delle transizioni eseguite nel passare da q_0 a q_j, v come la sequenza delle etichette delle transizioni eseguite nel passare da q_j a q_{j+m} e con z il resto della stringa x. Abbiamo che $|v| = m > 0$ e che $|yv| = j + m - 1 < n_L$. Inoltre, y fa passare T da q_0 a q_j, v fa passare T da q_j a q_j e z fa passare T da q_j a q_n (che è uno stato finale): pertanto, per ogni $i \geqslant 0$, la stringa $yv^i z$ fa passare T da q_0 a q_n e, quindi, appartiene a L. ◇

Il risultato precedente fornisce una condizione necessaria affinché un linguaggio sia regolare: per questo motivo, esso viene principalmente utilizzato per dimostrare che un linguaggio *non* è regolare.

Esempio 3.4: un linguaggio non regolare

Consideriamo il linguaggio L dell'Esempio **??** costituito da tutte e sole le stringhe del tipo $a^n b^n$, per $n > 0$. Chiaramente questo linguaggio è infinito. Se L fosse regolare, allora esisterebbe il numero $n_L > 0$ definito nel Lemma 3.1: consideriamo la stringa $a^{n_L} b^{n_L}$. Dal lemma, segue che tale stringa può essere decomposta nella concatenazione di tre stringhe y, v e z tale che $|v| > 0$ e $yv^2z \in L$. D'altra parte, poiché $|yv| < n_L$, abbiamo che v deve essere costituita dal solo simbolo a, per cui, assumendo che $yvz \in L$, yv^2z conterrebbe un numero di simboli a maggiore del numero di simboli b, contraddicendo il fatto che $yv^2z \in L$: pertanto, L non può essere un linguaggio regolare.

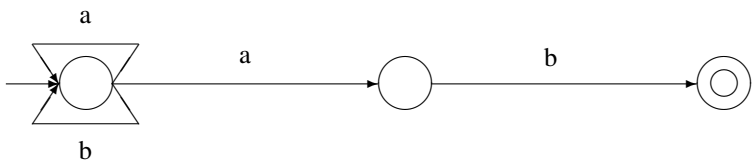
In conclusione, quanto esposto in questo capitolo ci consente di aggiornare la tabella di classificazione dei linguaggi in base alla loro tipologia, al tipo di grammatiche che li generano e in base al modello di calcolo corrispondente: la nuova tabella è la seguente.

Tipo di linguaggio	**Tipo di produzioni**	**Modello di calcolo**
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	Automa a stati finiti

Esercizi

Esercizio 3.1. Trasformare l'automa a stati finiti non deterministico della Figura 3.2 in un automa deterministico equivalente, facendo uso della tecnica di costruzione mediante sottoinsiemi.

Esercizio 3.2. Facendo uso della tecnica di costruzione mediante sottoinsiemi, trasformare il seguente automa a stati finiti non deterministico in un automa deterministico equivalente.



Esercizio 3.3. Dimostrare il Teorema 3.2.

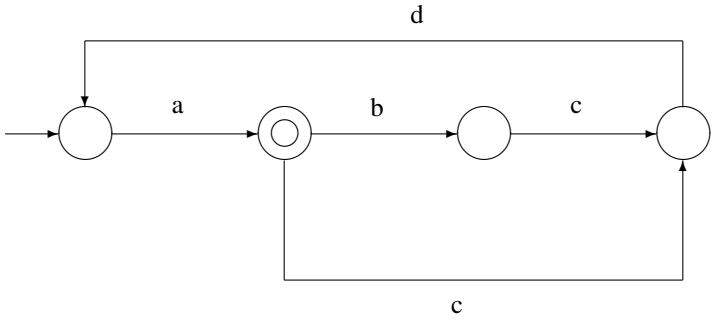
Esercizio 3.4. Dimostrare che la concatenazione non è commutativa.

Esercizio 3.5. Derivare un'espressione regolare che generi l'insieme di tutte le sequenze di 0 ed 1 che contengono un numero di 0 divisibile per 3.

Esercizio 3.6. Sia rev un operatore che inverte le sequenze di simboli: ad esempio, $rev(abc) = cba$. Sia L un linguaggio generato da un'espressione regolare e sia $L^r = \{x : rev(x) \in L\}$. Dimostrare che esiste un'espressione regolare che genera L^r.

Esercizio 3.7. Definire un automa a stai finiti non deterministico che accetta il linguaggio generato dalla seguente espressione regolare: $(((1 \cdot 1)^* 0)^* + 0 \cdot 0)^*$. Quindi derivare l'equivalente automa a stati finiti.

Esercizio 3.8. Definire un'espressione regolare che generi il linguaggio accettato dal seguente automa a stati finiti.



Esercizio 3.9. Dimostrare che il linguaggio complementare di quello considerato nell'Esempio 3.4 non è regolare.
Esercizio 3.10. Dire, giustificando la risposta, quali delle seguenti affermazioni sono vere.

1. Se L è un linguaggio regolare e L' è un linguaggio non regolare, allora $L \cap L'$ è un linguaggio regolare se e solo se $L \cup L'$ è un linguaggio regolare.
2. Se $L_1 \subseteq L_2$ e L_2 è un linguaggio regolare, allora L_1 è un linguaggio regolare.
3. Se L_1 e L_2 sono due linguaggi non regolari, allora $L_1 \cup L_2$ non può essere un linguaggio regolare.
4. Se L_1 è un linguaggio non regolare, allora il complemento di L_1 non può essere un linguaggio regolare.

Esercizio 3.11. Utilizzando il Lemma 3.1 dimostrare che il linguaggio costituito da tutte e sole le stringhe binarie del tipo ww, con $w \in \{0,1\}^*$, non è regolare.
Esercizio 3.12. Utilizzando il Lemma 3.1 dimostrare che il linguaggio costituito da tutte e sole le stringhe del tipo 0^{2^n}, con $n \geqslant 0$, non è regolare.
Esercizio 3.13. Facendo uso del pumping lemma per i linguaggi regolari, dimostrare che il seguente linguaggio non è regolare.

$$L = \{0^i 1^j : i > j\}$$

Esercizio 3.14. Facendo uso del pumping lemma per i linguaggi regolari, dimostrare che il seguente linguaggio non è regolare.

$$L = \{x2y : x, y \in \{0,1\}^* \text{ e il numero di } 0 \text{ in } x \text{ è uguale al numero di } 1 \text{ in } y\}$$

Esercizio 3.15. Facendo uso del pumping lemma, dimostrare che il seguente linguaggio non è regolare.

$$L = \{0^p : p \text{ è un numero primo}\}$$

Esercizio 3.16. Dimostrare che il linguaggio

$$L = \{0^i 1^j 2^k : i = 1 \Rightarrow j = k\}$$

soddisfa le condizioni del Lemma 3.1, pur non essendo regolare.
Esercizio 3.17. Si consideri la seguente dimostrazione.

> Sia L il linguaggio costituito da tutte e sole le stringhe binarie di lunghezza pari a 1000. Supponiamo che L sia regolare. Sia $x = 0^{1000}$ una stringa in L: in base al pumping lemma per i linguaggi regolari, x può essere decomposta nella concatenazione di tre stringhe y, v e z tali che $|v| > 0$ e, per ogni $i > 0$, $yv^i z \in L$. Ma questo contraddice il fatto che L contiene solo stringhe di lunghezza pari a 1000. Quindi, L non è regolare.

Tale dimostrazione è certamente sbagliata. Perché? Dire, inoltre, qual è l'errore contenuto nella dimostrazione.

Capitolo 4

Analisi sintattica top-down

SOMMARIO

Una delle principali componenti di un compilatore è il parser, il cui compito è quello di analizzare la struttura di un programma e delle sue istruzioni componenti e di verificare l'esistenza di errori. La risorsa principale nello sviluppo di un parser sono le grammatiche libere da contesto. In questo capitolo, analizzeremo tali grammatiche e mostreremo una tecnica per utilizzarle per effettuare l'analisi sintattica di un programma, basata sulla costruzione top-down degli alberi di derivazione.

4.1 Introduzione

L'analizzatore sintattico, anche detto *parser*, è il cuore della prime tre fasi di un compilatore. Il suo compito principale è quello di analizzare la struttura del programma e delle sue istruzioni componenti e di verificare l'esistenza di errori. A tale scopo interagisce con l'analizzatore lessicale, che fornisce i token in risposta alle richieste del parser: allo stesso tempo, il parser può supervisionare le operazioni del generatore di codice intermedio.

La risorsa principale nello sviluppo del parser sono le grammatiche libere da contesto, le quali sono molto spesso utilizzate per definire uno specifico linguaggio di programmazione. A essere precisi, molti linguaggi di programmazione reali non possono essere descritti completamente da questo tipo di grammatiche: in tali linguaggi, infatti, vi sono spesso delle restrizioni che non possono essere imposte da grammatiche libere da contesto. Per esempio, i linguaggi fortemente tipati richiedono che ogni variabile sia dichiarata prima di essere usata: le grammatiche libere da contesto non sono in grado di imporre tale vincolo (d'altra parte, le grammatiche che possono farlo sono troppo complesse per essere usate nella costruzione di compilatori). In ogni caso, eccezioni come queste sono rare e possono essere gestite semplicemente con altri mezzi: pertanto le grammatiche libere da contesto, essendo così comode da usare e fornendo così facilmente metodi efficienti per la costruzione di analizzatori sintattici, sono generalmente utilizzate per il progetto dei compilatori.

4.2 Alberi di derivazione

Consideriamo le espressioni aritmetiche in un linguaggio di programmazione come JAVA formate facendo uso delle sole operazioni di somma, sottrazione, moltiplicazione e divisione. Una semplice grammatica libera da contesto per tali espressioni è la seguente:

$$E \to E+E \quad E \to E-E \quad E \to E*E$$
$$E \to E/E \quad E \to (E) \quad E \to \textbf{id}$$

Facendo uso di tale grammatica, analizziamo l'espressione $(a+b)/(a-b)$. L'analizzatore lessicale passerà quest'espressione in forma di token, sostituendo i lessemi a e b con il token **id**. Quindi la sequenza

Figura 4.1: lo sviluppo di un albero di derivazione.

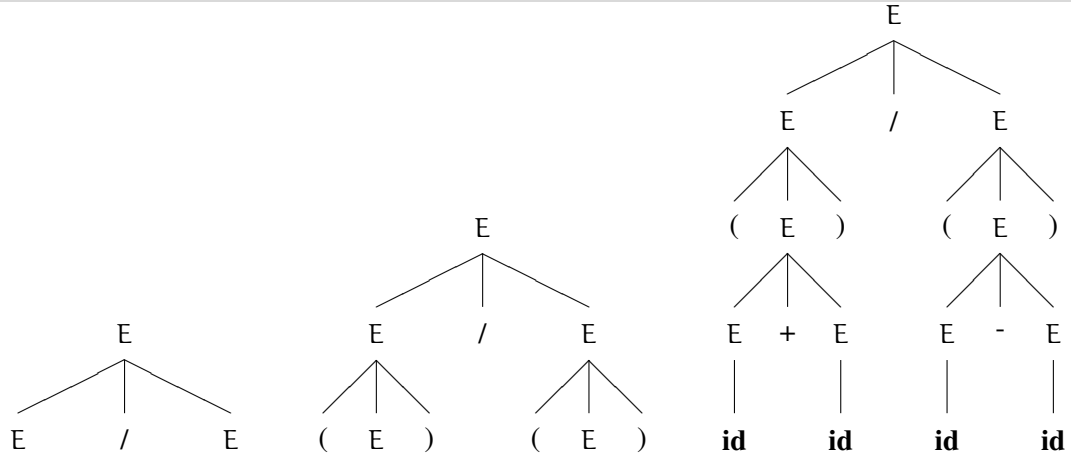
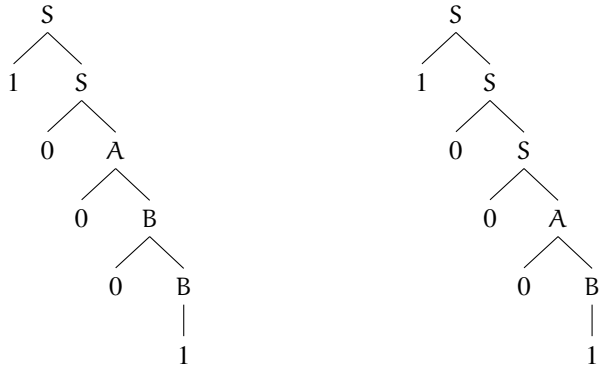


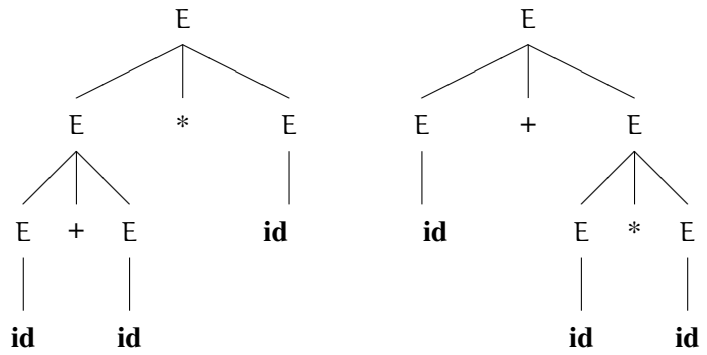
Figura 4.2: alberi di derivazione.



di simboli terminali da dover generare è $(\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})$. Quest'espressione è una frazione il cui numeratore è $(\mathbf{id}+\mathbf{id})$ e il cui denominatore è $(\mathbf{id}-\mathbf{id})$: pertanto, la prima produzione che usiamo è $\mathtt{E} \to \mathtt{E}/\mathtt{E}$ come mostrato nella parte sinistra della Figura 4.1. Osserviamo che la produzione è rappresentata mediante un albero in cui la parte a sinistra della produzione è associata al nodo padre e i simboli della parte destra ai nodi figli. Il numeratore e il denominatore della frazione sono entrambi espressioni parentesizzate: quindi, li sostituiamo entrambi facendo uso della regola $\mathtt{E} \to (\mathtt{E})$ come mostrato nella parte centrale della Figura 4.1. Il contenuto della parentesi al numeratore è una somma: quindi, sostituiamo la E all'interno della parentesi facendo uso della produzione $\mathtt{E} \to \mathtt{E}+\mathtt{E}$. Il denominatore invece è una differenza: quindi, sostituiamo la E all'interno della parentesi con la produzione $\mathtt{E} \to \mathtt{E}-\mathtt{E}$. A questo punto, le E rimanenti devono produrre i token identificatori terminali: quindi, facciamo uso della regola $\mathtt{E} \to \mathbf{id}$ come mostrato nella parte destra della Figura 4.1.

Data una grammatica $G = (V, N, S, P)$, un **albero di derivazione** è un albero la cui radice è etichettata con S (ovvero il simbolo iniziale della grammatica), le cui foglie sono etichettate con simboli in V (ovvero simboli terminali) e tale che, per ogni nodo con etichetta un simbolo non terminale A, i figli di tale nodo siano etichettati con i simboli della parte destra di una produzione in P la cui parte sinistra sia A. Un albero di derivazione è detto essere un albero di derivazione della stringa x se i simboli che etichettano le foglie dell'albero, letti da sinistra verso destra, formano la stringa x.

Figura 4.3: due alberi di derivazione in una grammatica ambigua.



Esempio 4.1: alberi di derivazione

Consideriamo la grammatica regolare contenente le seguenti produzioni:

$$S \to 1S \quad S \to 0S \quad S \to 0A \quad A \to 0B \quad A \to 0 \quad B \to 0B \quad B \to 1B \quad B \to 0 \quad B \to 1.$$

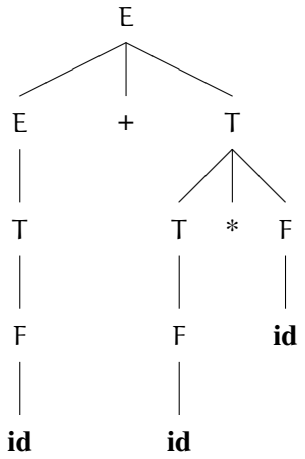
Un esempio di albero di derivazione della stringa `10001` è mostrato nella parte sinistra della Figura 4.2, mentre un altro esempio di albero di derivazione della stessa stringa è mostrato nella parte destra della figura.

4.2.1 Grammatiche ambigue

L'esempio precedente mostra un'altra importante caratteristica delle grammatiche generative, ovvero il fatto che tali grammatiche possano essere ambigue in quanto la stessa stringa può essere prodotta in diversi modi. Nell'esempio precedente ciò non sembra comportare particolari problemi, ma, come ulteriore esempio di tale fenomeno e facendo riferimento alla grammatica delle espressioni aritmetiche, consideriamo l'espressione $\mathbf{id} + \mathbf{id} * \mathbf{id}$. Possiamo generarla usando la produzione $E \to E * E$, quindi applicando la produzione $E \to E + E$ e, infine, applicando ripetutamente la produzione $E \to \mathbf{id}$ per ottenere l'albero mostrato nella parte sinistra della Figura 4.2. Ma avremmo anche potuto iniziare applicando $E \to E + E$, quindi $E \to E * E$ e, infine, $E \to \mathbf{id}$ per ottenere l'albero mostrato nella parte destra della figura. Questi due alberi sono chiaramente diversi: qual'è quello giusto?

Non possiamo rispondere a questa domanda se conosciamo solo la grammatica, in quanto entrambi gli alberi sono stati costruiti usando le sue produzioni e non vi è ragione perchè entrambi non possano essere accettabili. Una grammatica nella quale è possibile analizzare anche una sola sequenza in due o più modi diversi è detta **ambigua**: quest'ambiguità se non viene risolta in qualche modo è chiaramente inaccettabile per un compilatore. Sebbene non esistano metodi automatici per eliminare le ambiguità di una grammatica, esistono comunque due principali tecniche per trattare questo problema. Facendo riferimento alla prima, quando guardiamo dal di fuori la grammatica delle espressioni e consideriamo le regole di precedenza degli operatori in JAVA e in molti altri linguaggi ad alto livello, vediamo che uno solo dei due alberi della Figura 4.2 può essere quello giusto. L'albero di sinistra, infatti, dice che l'espressione è un prodotto e che uno dei due fattori è una somma, mentre l'albero di destra dice che l'espressione è una somma e che uno dei due addendi è un prodotto: quest'ultima è la normale interpre-

Figura 4.4: albero di derivazione in una grammatica non ambigua.



tazione dell'espressione $\mathbf{id} + \mathbf{id} * \mathbf{id}$. Se possiamo in qualche modo incorporare all'interno del parser il fatto che la moltiplicazione ha una maggiore priorità rispetto all'addizione, questo risolverà l'ambiguità.

La seconda alternativa è quella di riscrivere la grammatica in modo da eliminare le ambiguità. Per esempio, se distinguiamo tra *espressioni*, *termini* e *fattori*, possiamo definire la seguente grammatica alternativa:

$$\begin{array}{lll} E \to E+T & E \to E-T & E \to T \\ T \to T*F & T \to T/F & T \to F \\ F \to (E) & F \to \mathbf{id} & \end{array}$$

In altre parole, questa grammatica esplicita il fatto che, se vogliamo usare la somma o sottrazione di due addendi come fattore di una moltiplicazione o di una divisione, allora dobbiamo prima racchiudere tale somma o sottrazione tra parentesi. Facendo uso di questa grammatica, il nostro esempio può essere analizzato in un solo modo come mostrato nella Figura 4.4 (considereremo di nuovo tale grammatica nel seguito).

4.2.2 Derivazioni destre e sinistre

Consideriamo ancora la grammatica delle espressioni aritmetiche introdotta all'inizio di questo paragrafo e supponiamo di voler generare la sequenza $(\mathbf{id} + \mathbf{id})/(\mathbf{id} - \mathbf{id})$. Come abbiamo già osservato, esistono diversi modi per poterlo fare. Uno di questi è la seguente derivazione:

$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E}/\mathtt{E} \\
&\rightarrow \mathtt{E}/(\mathtt{E}) \\
&\rightarrow \mathtt{E}/(\mathtt{E}-\mathtt{E}) \\
&\rightarrow \mathtt{E}/(\mathtt{E}-\mathbf{id}) \\
&\rightarrow \mathtt{E}/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E}+\mathtt{E})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E}+\mathbf{id})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}
$$

Osserviamo che, a ogni passo della derivazione precedente, abbiamo scelto il non terminale più a destra come quello da sostituire. Ad esempio, nella forma sentenziale $\mathtt{E}/(\mathtt{E}-\mathtt{E})$ avevamo tre scelte di $\mathtt{E}$ da poter sostituire e abbiamo scelto quella più a destra. Una tale derivazione è detta **derivazione destra**. Alternativamente, avremmo potuto selezionare il non terminale più a sinistra a ogni passo e avremmo ottenuto una **derivazione sinistra**, come quella seguente:

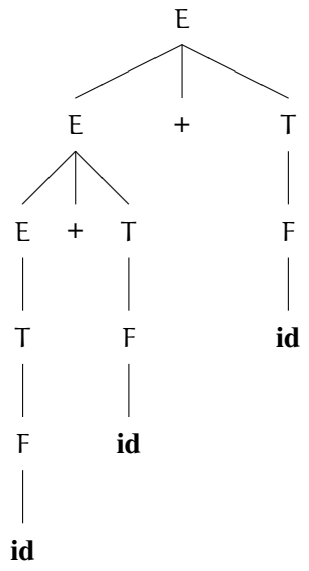
$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E}/\mathtt{E} \\
&\rightarrow (\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathtt{E}+\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathbf{id})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathtt{E}-\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}
$$

Notiamo che mentre è possibile costruire diverse derivazioni corrispondenti allo stesso albero di derivazione, le derivazioni sinistre e destre sono uniche. Ogni forma sentenziale che occorre in una derivazione sinistra è detta *forma sentenziale sinistra* e ogni forma sentenziale che occorre in una derivazione destra è detta *forma sentenziale destra*. La distinzione tra derivazioni sinistre e destre non è puramente accademica: esistono, infatti, due tipi di analizzatori sintattici, di cui un tipo genera derivazioni sinistre e un altro derivazioni destre, e le differenze tra questi due tipi incide direttamente sui dettagli della costruzione del parser e sulle sue operazioni. In queste dispense ci limiteremo ad analizzare nel prossimo paragrafo gli analizzatori del primo tipo, anche detti parser top-down.

4.3 Parser top-down

Un parser **top-down** parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero che porta alla data sequenza di simboli terminali: nel fare ciò, ricostruisce una derivazione sinistra. Il parser top-down deve iniziare dalla radice dell'albero e determinare in base alla sequenza di simboli terminali come far crescere l'albero di derivazione: inoltre, deve fare questo in base solo alla conoscenza delle produzioni nella grammatica e dei simboli terminali in arrivo da sinistra verso destra. Quest'approccio fa sorgere diversi problemi che analizzeremo e risolveremo, in questo paragrafo, man mano che si presenteranno fino a sviluppare gradualmente il metodo generale per la costruzione di un parser top-down.

Figura 4.5: l'albero di derivazione di $\mathbf{id} + \mathbf{id} + \mathbf{id}$.



4.3.1 Ricorsione sinistra

La scansione della sequenza di simboli terminali da sinistra a destra ci crea subito dei problemi. Supponiamo che la nostra grammatica sia la seguente:

$$\begin{array}{ll} E \to E + T & E \to T \\ T \to T * F & T \to F \\ F \to (E) & F \to \mathbf{id} \end{array}$$

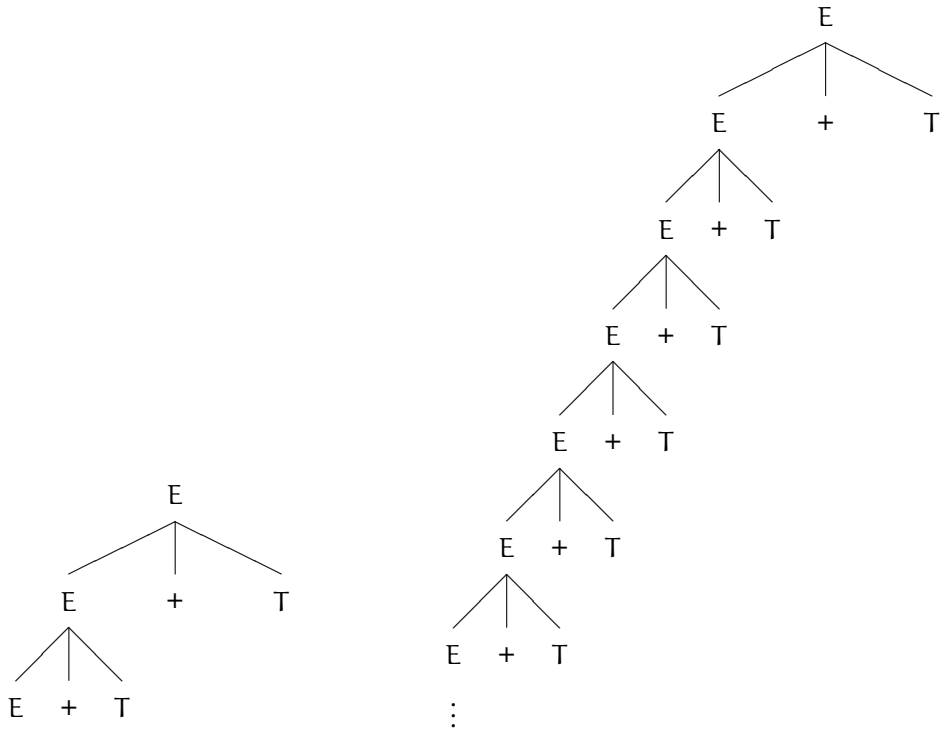
(si tratta di una versione semplificata della grammatica precedentemente introdotta per risolvere il problema delle ambiguità). Supponiamo di stare analizzando l'espressione $\mathbf{id} + \mathbf{id} + \mathbf{id}$, il cui albero di derivazione è mostrato nella Figura 4.5. Un parser top-down per questa grammatica partirà tentando di espandere E con la produzione $E \to E + T$. Quindi tenterà di espandere il nuovo E nello stesso modo: questo ci darà l'albero mostrato nella parte sinistra della Figura 4.6 che finora è corretto. Ora sappiamo che la E più a sinistra dovrebbe essere sostituita mediante la regola $E \to T$ invece di usare di nuovo $E \to E + T$. Ma come può l'analizzatore sintattico saperlo? Il parser non ha nulla per andare avanti se non la grammatica e la sequenza di simboli terminali in input e nulla nell'input è cambiato fino a ora. Per questo motivo, non vi è modo di evitare che il parser faccia crescere l'albero di derivazione indefinitamente come mostrato nella parte destra della figura.

In effetti, non esiste soluzione a questo problema finchè la grammatica è nella sua forma corrente. Produzioni della forma

$$A \to A\alpha$$

sono produzioni **ricorsive a sinistra** e nessun parser top-down è in grado di gestirle. Infatti, osserviamo che il parser procede "consumando" i simboli terminali: ognuno di tali simboli guida il parser nella sua scelta di azioni. Quando il simbolo terminale è usato, un nuovo simbolo terminale diviene disponibile e ciò porta il parser a fare una diversa mossa. I simboli terminali vengono consumati quando si accordano con i terminali nelle produzioni: nel caso di una produzione ricorsiva a sinistra, l'uso ripetuto di tale produzione non usa simboli terminali per cui, a ogni mossa nuova, il parser ha di fronte lo stesso simbolo terminale e quindi farà la stessa mossa.

Figura 4.6: il problema della ricorsione sinistra.



Questo problema affligge tutti i parser top-down comunque essi siano implementati. La soluzione consiste nel riscrivere la grammatica in modo che siano eliminate le ricorsioni sinistre. A tale scopo distinguiamo tra due tipi di tali ricorsioni: le ricorsioni sinistre **immediate** generate da produzioni del tipo

$$A \to A\alpha$$

e quelle **non immediate** generate da produzioni del tipo

$$A \to B\alpha \qquad B \to A\beta$$

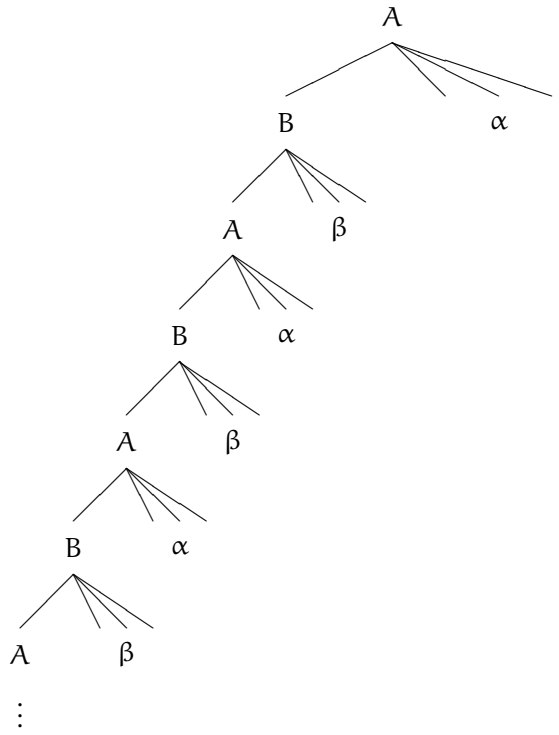
In quest'ultimo caso, A produrrà $B\alpha$, B produrrà $A\beta$ e così via: il parser costruirà quindi l'albero mostrato nella Figura 4.7. Nel seguito, discuteremo anzitutto l'eliminazione delle ricorsioni dirette, poichè questa è in un certo senso l'operazione di base, facendo uso della quale potremo poi descrivere la tecnica generale che rimuove tutti i tipi di ricorsione sinistra.

Eliminazione di ricorsioni sinistre immediate

Per rimuovere le ricorsioni sinistre immediate, procediamo nel modo seguente (nel seguito assumiamo che la grammatica originale non contenga λ-produzioni, ovvero produzioni la cui parte destra sia uguale a λ). Per ogni non terminale A nella grammatica che presenta almeno una produzione ricorsiva sinistra immediata, eseguiamo le seguenti operazioni.

1. Separiamo le produzioni ricorsive a sinistra immediate dalle altre. Supponiamo che le produzioni siano le seguenti:

Figura 4.7: ricorsione sinistra non immediata.



$$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots$$

e

$$A \to \delta_1 \mid \delta_2 \mid \delta_3 \mid \cdots$$

2. Introduciamo un nuovo non terminale A'.

3. Sostituiamo ogni produzione non ricorsiva $A \to \delta_i$ con la produzione $A \to \delta_i A'$.

4. Sostituiamo ogni produzione ricorsiva immediata $A \to A\alpha_i$ con la produzione $A' \to \alpha_i A'$.

5. Aggiungiamo la produzione $A' \to \lambda$.

Osserviamo che la rimozione di ricorsioni immediate introduce produzioni del tipo $A \to \lambda$. L'idea dell'algoritmo appena descritto è quella di anticipare l'uso delle produzioni non ricorsive che in ogni caso dovranno essere prima o poi utilizzate: così facendo, le ricorsioni sinistre immediate vengono trasformate, mediante l'uso di un nuovo simbolo non terminale, in ricorsioni destre immediate, le quali non creano problemi a un parser di tipo top-down.

Esempio 4.2: eliminazione di ricorsioni sinistre immediate

Consideriamo la seguente grammatica:

$$S \to Sa \qquad S \to b$$

Tutte le derivazioni in questa grammatica hanno la forma

$$S \to Sa \to Saa \to Saaa \to \cdots \to ba^n$$

(il processo ha termine quando viene scelta la produzione $S \to b$). La grammatica trasformata è la seguente:

$$S \to bS' \qquad S' \to aS' \qquad S' \to \lambda$$

Tutte le derivazioni in questa grammatica hanno la forma

$$S \to bS' \to baS' \to baaS' \to baaaS' \to \cdots \to ba^n$$

(in questo caso il processo ha termine quando scegliamo $S' \to \lambda$). Come detto in precedenza, la procedura di eliminazione delle ricorsioni immediate ha anticipato l'uso della produzione non ricorsiva che in ogni caso doveva essere prima o poi utilizzata.

Osserviamo che l'eliminazione di ricorsioni immediate sinistre mediante l'introduzione di ricorsioni immediate destre può causare qualche problema quando si ha a che fare con operatori associativi a sinistra (come nel caso della sottrazione): questi problemi devono essere tenuti in considerazione al momento della produzione del codice intermedio.

Eliminazione di ricorsioni sinistre non immediate

Per rimuovere tutte le ricorsioni sinistre in una grammatica procediamo invece nel modo seguente.

1. Creiamo una lista ordinata $A_1, \ldots, A_m$ di tutti i simboli non terminali.
2. Per $j = 1, \ldots, m$, eseguiamo le seguenti operazioni:
   (a) per $h = 1, \ldots, j-1$, sostituiamo ogni produzione del tipo $A_j \to A_h\beta$ con l'insieme delle produzioni $A_j \to \gamma\beta$ per ogni produzione del tipo $A_h \to \gamma$ (facendo riferimento alle produzioni di A_h già modificate);
   (b) facendo uso della tecnica descritta in precedenza, rimuoviamo le eventuali ricorsioni sinistre immediate a partire da A_j (i nuovi non terminali che eventualmente vengono introdotti in questo passo non sono inseriti nella lista ordinata).

Esempio 4.3: eliminazione di ricorsioni sinistre non immediate

Consideriamo la seguente grammatica:

$$S \to aA \qquad S \to b \qquad S \to cS \qquad A \to Sd \qquad A \to e$$

Supponendo che S preceda A, abbiamo che le produzioni per S non richiedono alcuna modifica. Per il simbolo non terminale A, abbiamo una parte destra che inizia con S. Quindi sostituiamo tale produzione con le seguenti produzioni:

$$A \to aAd \qquad A \to bd \qquad A \to cSd$$

Quello che è successo è di avere inserito i passi iniziali di tutte le possibili derivazioni sinistre a partire da A tramite S nelle nuove parti destre. Infatti, a partire da A possiamo ottenere solo le seguenti forme sentenziali:

$$A \to Sd \to \begin{cases} aAd \to \cdots \\ bd \\ cSd \to \cdots \end{cases}$$

Nel riscrivere le produzioni di A, abbiamo semplicemente saltato il primo passo.

L'algoritmo appena descritto garantisce l'eliminazione di tutte le ricorsioni sinistre se si assume non solo che la grammatica non contenga λ-produzioni ma che non contenga nemmeno cicli, ovvero non sia possibile derivare il simbolo non terminale A a partire da A stesso. Il motivo per cui l'algoritmo funziona è che, all'iterazione j dell'algoritmo, ogni produzione del tipo $A_h \to A_i\eta$ con $h < j$ deve avere $i > h$ (in quanto A_h è già stato analizzato): pertanto, la produzione $A_j \to A_i\eta\beta$ che viene eventualmente aggiunta tra quelle in sostituzione di $A_j \to A_h\beta$ verrà successivamente elaborata durante la stessa iterazione e sostituita a sua volta con nuove produzioni. Prima o poi l'indice del simbolo non terminale che appare in prima posizione di una produzione la cui parte sinistra è A_j dovrà essere non inferiore a j: l'eliminazione delle ricorsioni sinistre immediate garantisce che, al termine dell'iterazione, quest'indice sarà strettamente maggiore di j. Il prossimo esempio mostra che l'algoritmo può funzionare anche nel caso la grammatica contenga λ-produzioni.

Esempio 4.4: eliminazione di ricorsioni sinistre non immediate con produzioni nulle

Consideriamo la seguente grammatica: $S \to Aa$, $S \to b$, $A \to Ac$, $A \to Sd$, $A \to \lambda$. Supponiamo che S preceda A. Analizzando il simbolo non terminale A, otteniamo le seguenti produzioni:

$$A \to Ac \qquad A \to Aad \qquad A \to bd \qquad A \to \lambda$$

Eliminando le ricorsioni sinistre immediate, otteniamo la nuova grammatica che non include alcuna ricorsione sinistra: $S \to Aa$, $S \to b$, $A \to bdA'$, $A \to A'$, $A' \to cA'$, $A' \to adA'$ e $A' \to \lambda$.

4.3.2 Backtracking

Un modo per sviluppare un parser top-down consiste semplicemente nel fare in modo che il parser tenti esaustivamente tutte le produzioni applicabili fino a trovare l'albero di derivazione corretto. Questo è talvolta detto il metodo della "forza bruta". Tale metodo può far sorgere tuttavia alcuni problemi. Per esempio, consideriamo la grammatica seguente:

$$S \to ee \qquad S \to bAc \qquad S \to bAe \qquad A \to d \qquad A \to cA$$

e la sequenza di simboli terminali $bcde$. Se il parser tenta tutte le produzioni esaustivamente, incomincerà considerando $S \to bAc$ poichè il b all'inizio dell'input impedisce di usare la produzione $S \to ee$. Il prossimo simbolo di input è c: questo elimina $A \to d$ e quindi viene tentata $A \to cA$. Proseguendo in questo modo, il parser genera l'albero mostrato nella Figura 4.8. Ma quest'albero è sbagliato! Esso genera la stringa $bcdc$ invece di $bcde$. Chiaramente ciò è dovuto al fatto che il parser è partito con la parte destra sbagliata: se fosse stato capace di guardare in avanti al simbolo di input finale, non avrebbe fatto quest'errore. Purtroppo, i parser top-down scandiscono i simboli terminali da sinistra verso destra. Per rimediare al danno, dobbiamo tornare indietro (in inglese *backtrack*) fino a trovare una produzione alternativa: in questo caso, il parser deve ritornare alla radice e tentare la terza produzione $S \to bAe$.

Il backtrack è sostanzialmente una visita in profondità di un grafo. La ricerca procede in avanti da nodo a nodo finchè viene trovata la soluzione oppure viene raggiunto un vicolo cieco. Se questo è il caso, deve tornare indietro fino a trovare una biforcazione lungo la strada e tentare quella possibilità. Similmente, il metodo della forza bruta lavora in avanti di produzione in produzione fino al successo oppure fino a un vicolo cieco. Se questo è il caso, deve tornare indietro fino a trovare una biforcazione ovvero una produzione con una parte destra non ancora tentata.

Ma con il procedere dell'analisi, l'input viene consumato. Ad esempio, nel generare l'albero precedente quando viene scelta $S \to bAc$, il parser supera la b nella sequenza di input e quando sceglie $A \to cA$ supera la c. Quindi, una volta constatato di essere giunto a un vicolo cieco, non solo deve demolire l'albero sbagliato ma deve anche tornare indietro nell'input al simbolo terminale che stava esaminando quando ha sbagliato strada. In alcuni casi, questo è relativamente facile, in altri può essere proibitivo. Se l'analisi lessicale viene eseguita come un passo separato e l'intero programma è stato ridotto in forma di token,

Figura 4.8: un esempio di derivazione sbagliata.

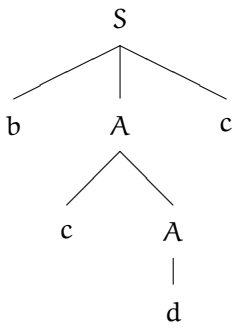
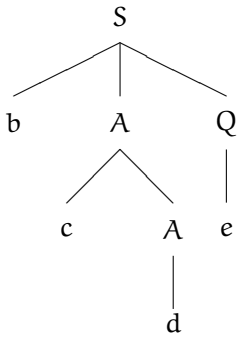


Figura 4.9: albero di derivazione per la grammatica fattorizzata.



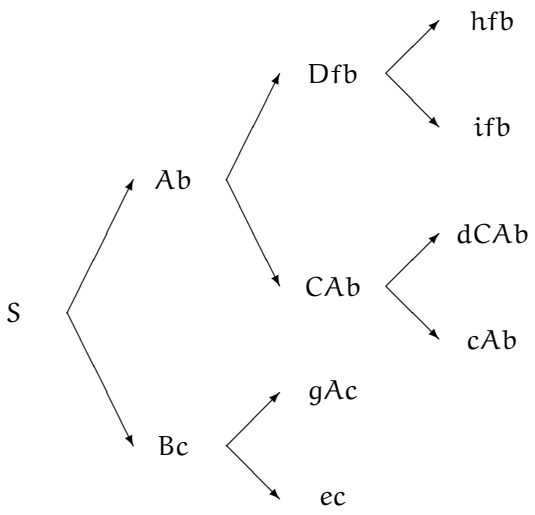
allora può essere semplice tornare indietro lungo la lista di token. Se invece l'analizzatore lessicale è sotto il controllo del parser e crea i token man mano che si procede, tornare indietro può essere difficile perchè il lettore deve tornare indietro al punto corrispondente della sequenza in input. Questi problemi possono tutti essere risolti ma rallentano le operazioni del compilatore. L'operazione è particolarmente lenta se il codice sorgente contiene un errore poiché il compilatore deve tornare indietro ripetutamente per tentare tutte le possibilità e vedere che tutte falliscono, prima di concludere che l'input è sbagliato. Per questi motivi, ogni metodo basato sul backtrack non è molto attraente.

Il problema è in parte nello sviluppo del parser e in parte nello sviluppo del linguaggio. L'esempio che abbiamo usato aveva una produzione che sembrava promettente ma che aveva una trappola alla fine. Osserviamo quanti meno problemi avremmo con la seguente grammatica:

$$S \to ee \qquad S \to \mathrm{b}A\mathrm{Q} \qquad \mathrm{Q} \to \mathrm{c} \qquad \mathrm{Q} \to e \qquad \mathrm{A} \to \mathrm{d} \qquad \mathrm{A} \to \mathrm{c}A$$

In questo caso abbiamo fattorizzato il prefisso comune bA e usato un nuovo non terminale per permettere la scelta finale tra c ed e. Questa grammatica genera lo stesso linguaggio di quella precedente ma il parser può ora generare l'albero di derivazione giusto senza backtrack come mostrato nella Figura 4.9. La trasformazione appena mostrata è nota come fattorizzazione sinistra ed è la prima di tante tecniche che rendono i parser top-down praticabili: un altro importante modo di evitare il backtrack è quello di cui parleremo nel prossimo paragrafo.

Figura 4.10: terminali derivabili dal simbolo iniziale.



4.3.3 Parser predittivi

Se nella grammatica al più una produzione inizia con un simbolo non terminale, allora la strategia del parser potrebbe essere facile: tenta le parti destre che iniziano con un terminale e se fallisce tenta quella che inizia con il non terminale. Ma supponiamo di avere la grammatica seguente:

$$S \to Ab \qquad S \to Bc \qquad A \to Df \qquad A \to CA$$

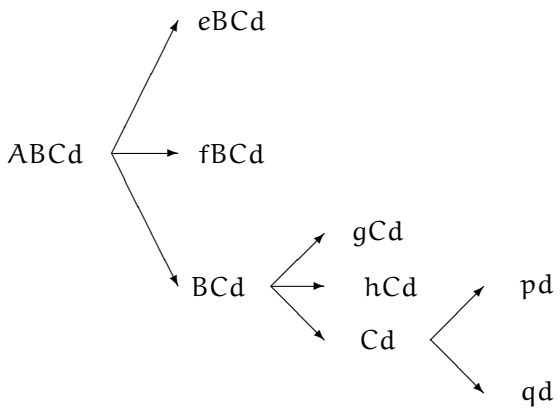
$$B \to gA \qquad B \to e \qquad C \to dC \qquad C \to c \qquad D \to h \qquad D \to i$$

In questo caso le parti destre delle produzioni da S e A non cominciano con simboli terminali. Di conseguenza il parser non ha una guida immediata dalla sequenza di input. Ad esempio, se la sequenza di input è gchfc, un parser deve sperimentare e fare molto backtrack prima di trovare la seguente derivazione: $S \to Bc \to gAc \to gCAc \to gcAc \to gcDfc \to gchfc$. Questo esempio non è irrealistico: frequentemente le grammatiche hanno diverse produzioni le cui parti destre iniziano con simboli non terminali.

Il backtrack potrebbe essere evitato se il parser avesse la capacità di guardare avanti nella grammatica in modo da anticipare quali simboli terminali sono derivabili (mediante derivazioni sinistre) da ciascuno dei vari simboli non terminali nelle parti destre delle produzioni. Per esempio, se seguiamo le parti destre di S, considerando tutte le possibili derivazioni sinistre, troviamo le possibilità mostrate nella Figura 4.10. Da questa figura vediamo che se la sequenza di input inizia con c, d, h oppure i dobbiamo scegliere $S \to Ab$ mentre se inizia con e oppure g dobbiamo scegliere $S \to Bc$. Se inizia con qualcosa di diverso, dobbiamo dichiarare un errore. La figura ci dice infatti quali simboli terminali possono iniziare forme sentenziali derivabili da Ab e quali terminali possono iniziare sequenze derivabili da Bc: questi insiemi sono noti come insiemi FIRST e sono generalemente indicati come $\text{FIRST}(Ab)$ (che è uguale a $\{c, d, h, i\}$) e $\text{FIRST}(Bc)$ (che è uguale a $\{e, g\}$). Data questa informazione, il parser tenterà la produzione $S \to Ab$ se il simbolo terminale in input appartiene a $\text{FIRST}(Ab)$ e la produzione $S \to Bc$ se appartiene a $\text{FIRST}(Bc)$. Se tale simbolo terminale non appartiene a $\text{FIRST}(S) = \text{FIRST}(Ab) \cup \text{FIRST}(Bc)$, allora la sequenza è grammaticalmente scorretta e può essere rifiutata.

Data una grammatica $G = (V, N, S, P)$, possiamo in generale definire formalmente la funzione $\text{FIRST} : (V \cup N)^+ \to 2^V$ come segue: se α è una sequenza di simboli terminali e non terminali e se X è l'insieme di tutte le forme sentenziali derivabili da α mediante derivazioni sinistre, allora, per ogni $\beta \in X$ che inizia

Figura 4.11: simboli terminali annullabili e funzione FIRST.



con un terminale x, x appartiene a $\text{FIRST}(\alpha)$. Per convenzione, inoltre, assumiamo che se la stringa λ è generabile a partire da α, allora $\lambda \in \text{FIRST}(\alpha)$.

In grammatiche di dimensione moderata, gli insiemi FIRST possono essere trovati a mano in modo simile a quanto fatto nella Figura 4.10. In generale, tuttavia, dovremo definire un metodo di calcolo di tali insiemi. A tale scopo, osserviamo che dalla definizione di FIRST seguono le seguenti proprietà:

1. se α inizia con un terminale x, allora $\text{FIRST}(\alpha) = x$;
2. $\text{FIRST}(\lambda) = \{\lambda\}$;
3. se α inizia con un non terminale A, allora $\text{FIRST}(\alpha)$ include $\text{FIRST}(A) - \{\lambda\}$.

La terza proprietà contiene una trappola nascosta. Supponiamo che α sia $AB\delta$ e che sia possibile generare λ a partire da A. Allora, per calcolare $\text{FIRST}(\alpha)$, dobbiamo anche seguire le possibilità a partire da B. Inoltre, se è possibile generare λ anche a partire da B, allora dobbiamo anche seguire le possibilità a partire da δ.

Esempio 4.5: simboli non terminali annullabili

Supponiamo che la grammatica G includa le seguenti produzioni:

$$S \to ABCd \qquad A \to e \qquad A \to f \qquad A \to \lambda$$

$$B \to g \qquad B \to h \qquad B \to \lambda \qquad C \to p \qquad C \to q$$

e che vogliamo trovare $\text{FIRST}(S) = \text{FIRST}(ABCd)$. Esplorando questa forma sentenziale, troviamo le possibilità mostrate nella Figura 4.11. Quindi $\text{FIRST}(ABCd) = \{e, f, g, h, p, q\}$.

Se è possibile generare λ a partire da un non terminale A, diciamo che A è **annullabile** (nell'esempio precedente sia A che B sono annullabili). Osserviamo che sebbene $\text{FIRST}(A)$ e $\text{FIRST}(B)$ includano λ, questo non è vero per $\text{FIRST}(ABCd)$: in effetti, $\text{FIRST}(\alpha)$ include λ solo se è possibile generare λ a partire da α e ciò può accadere solo se ogni elemento di α è annullabile.

Questo problema non sorge molto spesso, ma se vogliamo programmare la procedura di calcolo degli insiemi FIRST dobbiamo risolverlo. A tale scopo, supponiamo che α è del tipo $\beta X \delta$ dove β è una sequenza di zero o più simboli non terminali annullabili, X è un terminale oppure un non terminale non

annullabile, e δ è tutto quello che segue. Allora $\mathrm{FIRST}(\alpha) = (\mathrm{FIRST}(\beta) - \{\lambda\}) \cup \mathrm{FIRST}(X)$ (se tutto in α è annullabile, ovvero $\alpha = \beta$, allora $\mathrm{FIRST}(\alpha) = \mathrm{FIRST}(\beta)$).

Possiamo ora definire un algoritmo per il calcolo della funzione FIRST distinguendo i seguenti due casi.

1. α è un singolo carattere oppure $\alpha = \lambda$.
   (a) Se α è un terminale y allora $\mathrm{FIRST}(\alpha) = y$.
   (b) Se $\alpha = \lambda$ allora $\mathrm{FIRST}(\alpha) = \{\lambda\}$.
   (c) Se α è un non terminale A e $A \to \beta_i$ sono le produzioni a partire da A, per $i = 1, \ldots, k$, allora
   $$\mathrm{FIRST}(\alpha) = \bigcup_k \mathrm{FIRST}(\beta_k).$$
2. $\alpha = X_1 X_2 \cdots X_n$ con $n > 1$.
   (a) Poniamo $\mathrm{FIRST}(\alpha) = \emptyset$ e $j = 1$.
   (b) Poniamo $\mathrm{FIRST}(\alpha) = \mathrm{FIRST}(\alpha) \cup \mathrm{FIRST}(X_j) - \{\lambda\}$.
   (c) Se X_j è annullabile e $j < n$, allora poniamo $j = j + 1$ e ripetiamo il passo precedente.
   (d) Se $j = n$ e X_n è annullabile, allora includiamo λ in $\mathrm{FIRST}(\alpha)$.

Osserviamo come i due casi sopra descritti siano mutuamente ricorsivi.

Esempio 4.6: calcolo della funzione FIRST

Considerando l'esempio precedente, si ha che nel caso di $\alpha = ABCd$, C è il primo non terminale non annullabile. Quindi abbiamo che

$$\begin{aligned} \mathrm{FIRST}(ABCd) &= \{e, f\} \quad (\text{ovvero } \mathrm{FIRST}(A) - \{\lambda\}) \\ &\cup \{g, h\} \quad (\text{ovvero } \mathrm{FIRST}(B) - \{\lambda\}) \\ &\cup \{p, q\} \quad (\text{ovvero } \mathrm{FIRST}(C)) \\ &= \{e, f, g, h, p, q\} \end{aligned}$$

in accordo con quanto avevamo trovato in precedenza esplorando le produzioni a mano.

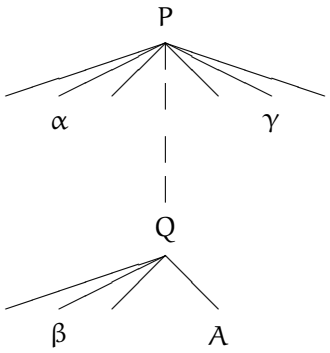
I parser che usano gli insiemi FIRST sono detti parser **predittivi**, in quanto hanno la capacità di vedere in avanti e prevedere che il primo passo di una derivazione ci darà prima o poi un certo simbolo terminale.

Grammatiche LL(1)

La tecnica basata sul calcolo della funzione FIRST non sempre può essere utilizzata. Talvolta la struttura della grammatica è tale che il simbolo terminale successivo non ci dice quale parte destra utilizzare (ad esempio, nel caso in cui $\mathrm{FIRST}(\alpha)$ e $\mathrm{FIRST}(\beta)$ non siano disgiunti). Inoltre, quando le grammatiche acquisiscono λ-produzioni come risultato della rimozione della ricorsione sinistra, gli insiemi FIRST non ci dicono quando scegliere $A \to \lambda$. Per gestire questi casi, abbiamo bisogno di una seconda funzione, FOLLOW. Nel definire tale funzione, assumeremo che una sequenza di simboli terminali, prima di essere passata al parser, abbia un segno di demarcazione appeso, che indicheremo con \$.

Data una grammatica $G = (V, N, S, P)$, definiamo la funzione $\mathrm{FOLLOW}: N \to 2^V$ come segue: se A è un simbolo non terminale, allora, per ogni $x \in V$ che può seguire A in una forma sentenziale, x appartiene a $\mathrm{FOLLOW}(A)$. Per convenzione, inoltre, assumiamo che se A può apparire come ultimo simbolo di una forma sentenziale, allora $\$ \in \mathrm{FOLLOW}(A)$. Per ogni non terminale A, possiamo calcolare $\mathrm{FOLLOW}(A)$ nel modo seguente.

Figura 4.12: calcolo della funzione FOLLOW.



1. Se A è il simbolo iniziale, allora includiamo \$ in FOLLOW(A).

2. Cerchiamo attraverso la grammatica le occorrenze di A nelle parti destre delle produzioni. Sia $Q \to xAy$ una di queste produzioni. Distinguiamo i seguenti tre casi:

   (a) se y inizia con un terminale q, allora includiamo q in FOLLOW(A);

   (b) se y non inizia con un terminale, allora includiamo FIRST(y) $-\{\lambda\}$ in FOLLOW(A);

   (c) se $y = \lambda$ (ovvero A è in fondo) oppure se y è annullabile, allora includiamo FOLLOW(Q) in FOLLOW(A).

Osserviamo che se la prima regola si applica, non possiamo fermarci ma dobbiamo procedere e applicare la seconda regola. Inoltre notiamo che escludiamo la stringa vuota λ perchè essa non non apparirà mai esplicitamente.

L'ultima regola richiede qualche spiegazione. Anzitutto, osserviamo che se A è alla fine della parte destra, questo significa che A può venire alla fine di una forma sentenziale derivabile da Q. Inoltre, se $Q \to \beta AB$ e B è annullabile, anche in questo caso A può venire alla fine di una forma sentenziale derivabile da Q. In questi casi, cosa può venire dopo A? Per rispondere consideriamo da dove viene Q stesso e per semplicità supponiamo che $B = \lambda$, come mostrato nella Figura 4.12. Se guardiamo in alto l'albero, vediamo che a partire da S abbiamo prima prodotto la forma sentenziale $\alpha Q\gamma$ e quindi la forma sentenziale $\alpha\beta A\gamma$. Quindi quello che viene dopo A è γ, che è esattamente ciò che viene dopo Q: per cui γ deve avere contribuito a FOLLOW(Q). Questo è il motivo per cui tutto ciò che è in FOLLOW(Q) deve anche stare in FOLLOW(A). Nel caso particolare in cui la parte sinistra sia anche A, ovvero $A \to xA$, ignoriamo questa regola poichè non aggiungerebbe nulla di nuovo.

Esempio 4.7: la grammatica delle espressioni

Consideriamo la grammatica delle espressioni che qui riformuliamo dopo avere eliminato le ricorsioni sinistre.

$$\begin{array}{lllll} E \to TQ & Q \to +TQ & Q \to -TQ & Q \to \lambda & T \to FR \\ R \to *FR & R \to /FR & R \to \lambda & F \to (E) & F \to \mathbf{id} \end{array}$$

In questo caso , è facile verificare che gli insiemi FIRST e FOLLOW sono i seguenti:

$$\mathrm{FIRST}(E) = \mathrm{FIRST}(F) = \mathrm{FIRST}(T) = \mathrm{FIRST}(TQ) = \mathrm{FIRST}(FR) = \{(, \mathbf{id}\}$$

$$\mathrm{FIRST}(Q) = \{+, -, \epsilon\} \qquad \mathrm{FIRST}(R) = \{*, /, \epsilon\} \qquad \mathrm{FIRST}(+TQ) = \{+\}$$

$$\mathrm{FIRST}(-TQQ) = \{-\} \qquad \mathrm{FIRST}(*RF) = \{*\} \qquad \mathrm{FIRST}(/RF) = \{/\}$$

$$\mathrm{FOLLOW}(E) = \mathrm{FOLLOW}(Q) = \{\$,)\} \qquad \mathrm{FOLLOW}(F) = \{+, -, *, /, \$,)\}$$

$$\mathrm{FOLLOW}(T) = \mathrm{FOLLOW}(R) = \{+, -, \$,)\}$$

In un parser predittivo, l'insieme FOLLOW ci dice quando usare le λ-produzioni. Supponiamo di dover espandere un non terminale A. Inizialmente vediamo se il simbolo terminale in arrivo appartiene all'insieme FIRST di una parte destra di una produzione la cui parte sinistra è A. Se così non è questo normalmente vuol dire che si è verificato un errore. Ma se una delle produzioni da A è $A \to \lambda$, allora dobbiamo vedere se il simbolo terminale appartiene a $\mathrm{FOLLOW}(A)$. Se così è, allora può non trattarsi di un errore e $A \to \lambda$ è la produzione scelta.

Grammatiche per cui questa tecnica può essere usata sono note come **grammatiche LL(1)** e i parser che usano questa tecnica sono detti *parser LL(1)*. In questa notazione, la prima L sta per "left" per indicare che la scansione è da sinistra a destra, la seconda L sta per "left" per indicare che la derivazione è sinistra, e '(1)' indica che si guarda in avanti di un carattere. Le grammatiche LL(1) assicurano che guardando un carattere in avanti il token in arrivo determina univocamente quale parte destra scegliere (nelle grammatiche LL(2) bisognerebbe guardare a coppie di simboli terminali). Perchè una grammatica sia LL(1) richiediamo che per ogni coppia di produzioni $A \to \alpha$ e $A \to \beta$, valgano le seguenti due proprietà:

1. $\mathrm{FIRST}(\alpha) - \{\lambda\}$ e $\mathrm{FIRST}(\beta) - \{\lambda\}$ siano disgiunti;
2. se α è annullabile, allora $\mathrm{FIRST}(\beta)$ e $\mathrm{FOLLOW}(A)$ devono essere disgiunti.

Se la prima regola è violata, allora ogni simbolo terminale in $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta)$ non riuscirà a dirci quale parte destra scegliere. Se la regola 2 è violata, allora ogni token in $\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A)$ non riuscirà a dirci se scegliere tra β oppure λ (ottenibile a partire da α).

4.4 Costruzione di un parser predittivo

Se possiamo evitare il backtrack, allora vi sono diversi modi di implementare il parser. Uno è quello di racchiudere ogni insieme di produzioni a partire da un simbolo non terminale in una funzione Booleana. Vi è una tale funzione per ogni non terminale della grammatica. La funzione tenterà ciascuna parte destra fino a che una corrispondenza non viene trovata. In tal caso ritorna il valore `true`, altrimenti ritorna il valore `false`. Nel caso di un parser LL(1) ciascuna di queste funzioni dovrà scegliere la parte destra in base agli insiemi FIRST e FOLLOW. La rimozione delle ricorsioni sinistre e l'utilizzo degli insiemi FIRST e FOLLOW rendono un tale parser praticabile. Il problema è che dobbiamo scrivere una funzione per ogni produzione. Se sorge qualche problema che rende necessario un cambiamento della grammatica, dobbiamo riprogrammare una o più di queste funzioni.

Una forma più conveniente di parser predittivo consiste di una semplice procedura di controllo che utilizza una tabella. Parte dell'attrattiva di questo approccio è che la procedura di controllo è generale: se

la grammatica va cambiata, solo la tabella deve essere riscritta creando meno problemi che la riprogrammazione. La tabella può essere costruita a mano per piccole grammatiche o mediante il calcolatore per grammatiche grandi. La tabella della versione non ricorsiva dice quale parte destra scegliere e i simboli terminali sono usati nel modo naturale.

Il nostro esempio sarà un parser per l'espressioni aritmetiche, usando la grammatica vista nell'esempio precedente, ovvero la grammatica delle espressioni aritmetiche ottenuta dopo aver avere eliminato le ricorsioni sinistre:

$$
\begin{array}{lll}
E \to TQ & & \\
Q \to +TQ & Q \to -TQ & Q \to \lambda \\
T \to FR & & \\
R \to *FR & R \to /FR & R \to \lambda \\
F \to (E) & F \to \textbf{id} &
\end{array}
$$

È più facile vedere prima la tabella e come viene utilizzata, per poi mostrare come costruirla. In particolare, la tabella per questa grammatica è la seguente (gli spazi bianchi indicano condizioni di errore):

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	*FR	/FR		ϵ	ϵ
F	**id**					(E)		

Intuitivamente, ogni elemento non vuoto della tabella indica quale produzione debba essere scelta dal parser trovandosi a dover espandere il simbolo non terminale che etichetta la riga della tabella e leggendo in input il simbolo terminale che etichetta la colonna della tabella.

Pertanto, nel caso dei parser guidati da una tabella siffatta, è utile mantenere una pila che viene usata nel modo seguente.

- Inseriamo $ nella pila e alla fine della sequenza e inseriamo il simbolo iniziale nella pila.
- Fintantoché la pila non è vuota
  - Sia x l'elemento in cima alla pila e a il simbolo terminale in input.
  - Se $x \in V$ allora:
    - se $x = a$ allora estraiamo x dalla pila e avanziamo di un simbolo terminale, altrimenti segnaliamo un errore.
  - Se $x \notin V$, allora:
    - se $\texttt{table}[x, a]$ non è vuoto, allora estraiamo x dalla pila e inseriamo $\texttt{table}[x, a]$ nella pila in *ordine inverso*, altrimenti segnaliamo un errore.

Notiamo che, nell'ultimo caso, inseriamo la parte destra di una produzione in ordine inverso in modo che il simbolo più a sinistra sarà in cima alla pila pronto per essere eventualmente espanso o cancellato. Ciò è dovuto al fatto che il parser sta cercando di generare una derivazione sinistra, in cui il simbolo non terminale più a sinistra è quello da espandere e al fatto che la pila è una struttura dati che consente l'inserimento e l'estrazione di un elemento dallo stesso punto di accesso.

Esempio 4.8: analisi di un'espressione aritmetica

Supponiamo che la sequenza in input sia **(id+id)*id**. Lo stato iniziale sarà il seguente:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$		

La sequenza ha il simbolo $ appeso in fondo e la pila ha tale simbolo e il simbolo iniziale inseriti (scriviamo il contenuto della pila in modo che cresca da sinistra verso destra). A questo punto l'analizzatore entra nel ciclo principale. Il simbolo in cima alla pila è E e $\texttt{table}[E, (] = TQ$: quindi la nostra produzione è $E \to TQ$ e abbiamo

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$		

In questo caso, E è stato estratto dalla pila e la parte destra TQ inserita nella pila in ordine inverso. Ora abbiamo un non terminale T in cima alla pila e il simbolo terminale in input è ancora (. Quindi abbiamo $\texttt{table}[T, (] = FR$ e la produzione è $T \to FR$:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$		

Proseguendo abbiamo $\texttt{table}[F, (] = (E)$ e la produzione è $F \to (E)$:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$	$F \to (E)$	$\to (E)RQ$
$QR)E(	**(id+id)*id**$		

Ora abbiamo un terminale in cima alla pila. Lo confrontiamo con il simbolo in input e, poichè coincidono, estraiamo il simbolo terminale dalla pila e ci spostiamo al prossimo simbolo della sequenza in input:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$	$F \to (E)$	$\to (E)RQ$
$QR)E(	**(id+id)*id**$		
$QR)E	**id+id)*id**$		

Così andando avanti si arriva a consumare l'intero input con la pila vuota e possiamo annunciare il successo dell'analisi. Possiamo costruire l'albero a partire dalle produzioni lette nell'ordine in cui appaiono come mostrato nella Figura 4.13.

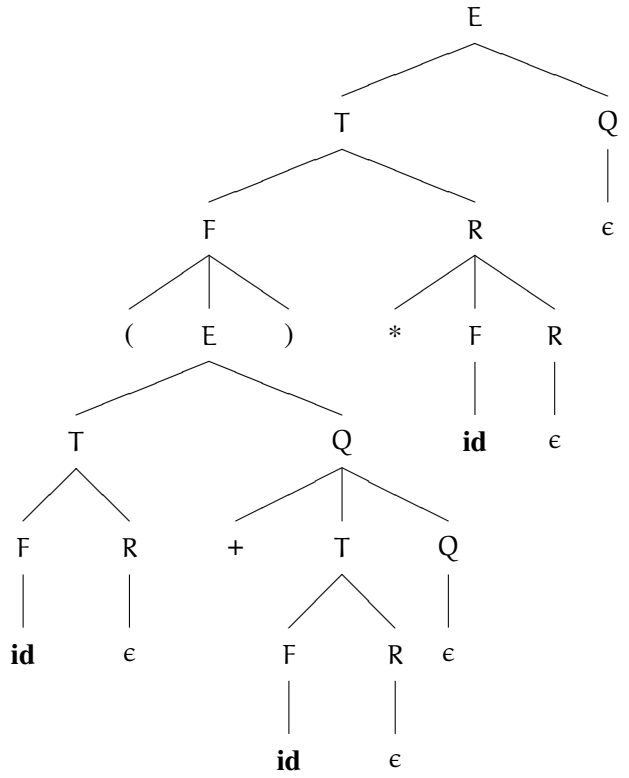
Costruzione della tabella del parser predittivo

Ricordiamo che in un parser top-down, se non vi deve essere backtrack, allora il simbolo terminale in arrivo deve sempre dirci cosa fare. La stessa cosa si applica alla costruzione della tabella. Supponiamo che abbiamo X in cima alla pila e che a sia il simbolo terminale in arrivo. Vogliamo selezionare una parte destra che incominci con a oppure che possa portare a una forma sentenziale che inizi con a.

Per esempio, all'inizio del nostro esempio, avevamo E nella pila e (come input. Avevamo bisogno di una produzione della forma $E \to (\cdots$ Ma una tale produzione non esiste nella grammatica. Poichè non era disponibile, avremmo dovuto tracciare un cammino di derivazione che ci conducesse a una forma sentenziale che iniziasse con (. L'unico tale cammino è

$$E \to TQ \to FRQ \to (E)RQ$$

Figura 4.13: albero di derivazione generato dal parser predittivo.



e se guardiamo alla tabella vediamo che essa contiene esattamente la parte destra che determina il primo passo del cammino.

Ciò ci sta conducendo verso un terreno familiare: vogliamo selezionare una parte destra α se il token appartiene a $\mathrm{FIRST}(\alpha)$; quindi per una riga A e una produzione $A \to \alpha$, la tabella deve avere la parte destra α in ogni colonna etichettata con un terminale in $\mathrm{FIRST}(\alpha)$. Ciò funzionerà in tutti i casi eccetto quello in cui $\mathrm{FIRST}(\alpha)$ include λ poichè la tabella non ha una colonna etichettata λ. Per questi casi, seguiamo gli insiemi FOLLOW.

La regola per costruire la tabella è dunque la seguente.

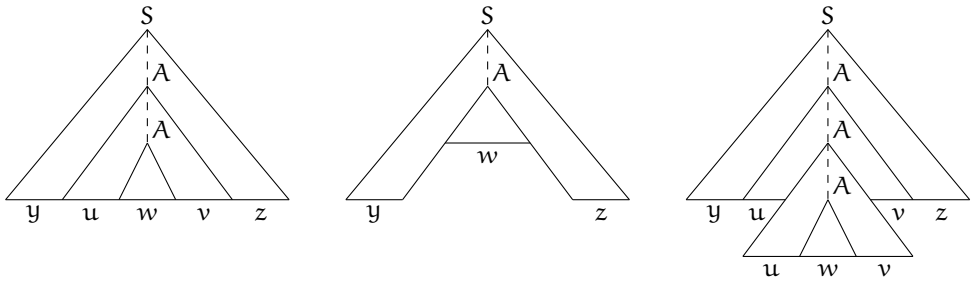
- Visita tutte le produzioni. Sia $X \to \beta$ una di esse.
  - Per tutti i terminali a in $\mathrm{FIRST}(\beta)$, poniamo $\mathtt{table}[X, a] = \beta$.
  - Se $\mathrm{FIRST}(\beta)$ include λ, allora, per ogni $a \in \mathrm{FOLLOW}(X)$, $\mathtt{table}[X, a] = \lambda$.

Nel caso della grammatica delle espressioni matematiche, usando gli insiemi FIRST e FOLLOW precedentemente calcolati e le regole sopra descritte, otteniamo la tabella mostrata in precedenza. Infine, se la grammatica è di tipo LL(1) siamo sicuri che la tabella non conterrà elementi multipli.

4.5 Linguaggi di tipo 2

Come nel caso dei linguaggi generati da grammatiche regolari, è naturale chiedersi se esistano linguaggi che non sono di tipo 2 ma che siano di tipo 1. La risposta a tale domanda è affermativa

Figura 4.14: il pumping lemma per linguaggi liberi da contesto.



Anche in questo caso come in quello dei linguaggi regolari, il risultato precedente viene comunemente utilizzato per dimostrare che un determinato linguaggio *non* è libero da contesto.

Esempio 4.9: un linguaggio non libero da contesto

Consideriamo il linguaggio L costituito da tutte e sole le stringhe del tipo $0^n1^n2^n$, per $n > 0$. Chiaramente, L è infinito. Quindi, se L fosse libero da contesto, esisterebbe il numero intero $n_L > 0$ del Lemma **??**. Consideriamo la stringa $x = 0^{n_L}1^{n_L}2^{n_L} \in L$: in base al lemma, abbiamo che $x = yuwvz$, con $|uv| > 0$, e che $yu^2wv^2z \in L$. Distinguiamo i seguenti due casi.

- Sia u che v contengono un solo tipo di simbolo. In tal caso, deve esistere un simbolo in $\{0, 1, 2\}$ che non appare in uv: pertanto, yu^2wv^2z non può contenere lo stesso numero di simboli 0, 1 e 2 e, quindi, non può appartenere a L.
- u oppure v contiene almeno due tipi di simboli (supponiamo che ciò sia vero per u). Poiché $x \in L$, i simboli 0, 1 e 2 devono apparire in u nell'ordine giusto, ovvero gli eventuali simboli 0 devono precedere gli eventuali simboli 1 i quali, a loro volta, devono precedere gli eventuali simboli 2: pertanto, la stringa yu^2wv^2z può contenere lo stesso numero di simboli 0, 1 e 2 ma certamente non li contiene nell'ordine giusto e, quindi, non può appartenere a L.

In entrambi i casi, abbiamo generato un assurdo e, quindi, il linguaggio L non può essere libero da contesto.

È facile verificare che il linguaggio L dell'esempio precedente è generato dalla grammatica contestuale contenente le seguenti regole di produzione: $A \to 0ABC$, $A \to 0BC$, $CB \to BC$, $1B \to 11$, $1C \to 12$, $2C \to 22$ e $0B \to 01$. Alternativamente, è altrettanto facile costruire una macchina di Turing lineare che decida L. Pertanto, possiamo concludere che la classe dei linguaggi liberi da contesto è strettamente inclusa in quella dei linguaggi contestuali.

L'Esempio 3.4 oltre a mostrare (combinato con l'Esempio **??**) che la classe dei linguaggi di tipo 3 è strettamente contenuta nella classe dei linguaggi di tipo 2, suggerisce anche quale potrebbe essere il modello di calcolo corrispondente a quest'ultima classe. In effetti, per riconoscere il linguaggio costituito da tutte e sole le stringhe del tipo a^nb^n, per $n > 0$, sembra necessario avere a disposizione una memoria aggiuntiva potenzialmente infinita, che consenta all'automa di ricordare il numero di simboli a letti. In realtà, è sufficiente un ben noto tipo di memoria, ovvero una *pila*, a cui è possibile accedere da un solo estremo, detto *cima* della pila, in cui un nuovo elemento viene inserito e da cui un elemento viene estratto. In effetti, il linguaggio costituito da tutte e sole le stringhe del tipo a^nb^n, per $n > 0$, può essere deciso da un automa a stati finiti dotato di una pila aggiuntiva, il quale inizialmente legge i simboli a e li inserisce nella pila: quindi, una volta incontrato il primo simbolo b, l'automa inizia a rimuovere i simboli a dalla pila, uno per ogni simbolo b che incontra nella stringa di input. Quest'ultima sarà accettata se alla fine dell'input la pila risulterà essere vuota.

Per poter caratterizzare i linguaggi liberi da contesto, tuttavia, abbiamo bisogno anche della possibilità di definire transizioni non deterministiche: intuitivamente, il non determinismo consentirà all'automa

a pila di provare in modo, appunto, non deterministico le diverse regole di produzione applicabili a una determinata forma sentenziale.

Definizione 4.1: automi a pila non deterministici

Un **automa a pila non deterministico** è una macchina di Turing non deterministica con due nastri semi-infiniti, con *alfabeto di input* Σ e con *alfabeto di pila* Γ (con $\square \notin \Sigma \cup \Gamma$), tale che ogni etichetta di un arco del grafo delle transizioni contiene triple del tipo (x, y, z), dove $x \in \Sigma^*$ è la sequenza di simboli letti sul primo nastro, $y \in \Gamma^*$ è la sequenza di simboli letti sul secondo nastro e $z \in \Gamma^*$ è la sequenza di simboli da scrivere sul secondo nastro al posto di y (eventualmente spostando a destra oppure a sinistra il contenuto alla destra di y). A ogni transizione, la testina del primo nastro si sposta di $|x|$ posizioni a destra e la testina del secondo nastro rimane sulla prima cella.

La computazione di un automa a pila non deterministico T ha inizio con la stringa di input x presente sul primo nastro e con il secondo nastro completamente vuoto: un cammino di computazione è accettante se termina in uno stato finale con la testina del primo nastro posizionata sul primo simbolo $\square$ alla destra di x. Quindi, se un cammino di computazione di T con input una stringa $x \in \Sigma^*$ termina in uno stato finale con la testina del primo nastro posizionata su un simbolo di x, allora tale cammino di computazione non è accettante. Osserviamo, inoltre, che un cammino di computazione accettante non necessariamente deve terminare con la pila vuota: non è difficile, comunque, modificare T in modo che ogni cammino di computazione accettante termini dopo aver svuotato completamente la pila.

Esempio 4.10: un automa a pila non deterministico

Consideriamo il linguaggio L costituito da tutte e sole le stringhe binarie del tipo $w_1 \cdots w_n w_n \cdots w_1$ con $w_i \in \{0, 1\}$ e $n > 0$. Un automa a pila non deterministico T, che accetti tutte e sole le stringhe in L, può non deterministicamente scegliere un punto in cui spezzare la stringa x di input e successivamente, facendo uso della pila, verificare che le due parti di x così ottenute siano una l'inversa dell'altra (si veda la Figura 4.15). In particolare, l'automa T, dopo aver inserito nella pila un simbolo speciale (come, ad esempio, il simbolo #), non deterministicamente prosegue l'esecuzione inserendo nella pila il successivo simbolo di x oppure (se il simbolo letto e quello in cima alla pila coincidono) iniziando il confronto tra i simboli di x e quelli contenuti nella pila. In quest'ultimo caso, T termina il confronto ed entra nel suo unico stato finale nel momento in cui il simbolo in cima alla pila è il simbolo #: se la stringa x è stata interamente letta, allora T accetta x. Osserviamo che L è chiaramente un linguaggio libero da contesto generato dalla grammatica avente le seguenti regole di produzione: $A \to 0A0$, $A \to 1A1$, $A \to 00$ e $A \to 11$.

L'esempio precedente può essere generalizzato all'intera classe dei linguaggi generati da grammatiche libere da contesto, come mostrato dal seguente risultato.

Teorema 4.1

Un linguaggio L è di tipo 2 se e solo se esiste un automa a pila non deterministico T tale che $L = L(T)$.

Esercizi

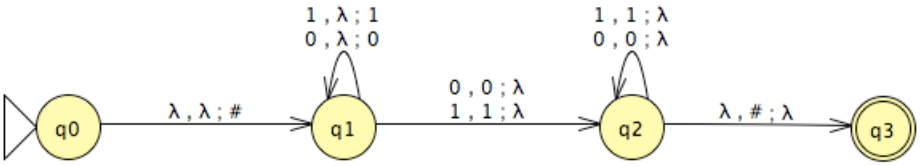
Esercizio 4.1. Definire una grammatica libera da contesto che generi il seguente linguaggio

$$L = \{a^n b^m : n > m + 2\}.$$

Esercizio 4.2. Definire una grammatica libera da contesto che generi il seguente linguaggio:

$$L = \{a^n b^n c^m : n > 0, m > 0\}.$$

Figura 4.15: l'automa a pila non deterministico dell'Esempio 4.10.



Esercizio 4.3. Definire una grammatica libera da contesto che generi il linguaggio formato da sequenze palindrome, ovvero

$$L = \{x a x^r : x \in (0|1)^* \wedge a \in (0|1|\epsilon)\}$$

dove x^r indica la sequenza ottenuta invertendo la sequenza x (ad esempio, $abc^r = cba$).
Esercizio 4.4. Dato un linguaggio L sull'alfabeto $\{0,1\}$, definiamo

$$\mathtt{Init}(L) = \{z \in \{0,1\}^* : (\exists w \in \{0,1\}^*)[zw \in L]\}.$$

Dimostrare che se L è libero da contesto allora anche $\mathtt{Init}(L)$ è libero da contesto.
Esercizio 4.5. Si consideri la grammatica

$$S \to \mathbf{a}S\mathbf{b}S \qquad S \to \mathbf{b}S\mathbf{a}S \qquad S \to \lambda$$

Quanti differenti alberi di derivazione esistono per la sequenza **abab**? Mostrare le derivazioni sinistre e destre.
Esercizio 4.6. Quali delle due seguenti grammatiche sono LL(1)? Giustificare la risposta.

1. $S \to ABBA \quad A \to \mathbf{a} \quad A \to \lambda \quad B \to \mathbf{b} \quad B \to \lambda$
2. $S \to \mathbf{a}S\mathbf{e} \quad S \to B \quad B \to \mathbf{b}B\mathbf{e} \quad B \to C \quad C \to \mathbf{c}C\mathbf{e} \quad B \to \mathbf{d}$

Esercizio 4.7. Dimostrare che ogni linguaggio riconosciuto da un automa a stati finiti può essere generato da una grammatica LL(1).
Esercizio 4.8. Dimostrare che l'eliminazione delle ricorsioni sinistre e la fattorizzazione non garantiscono che la grammatica ottenuta sia LL(1).
Esercizio 4.9. Progettare una tabella di parsing LL(1) per la grammatica

$$E \to -E \quad E \to (E) \quad E \to VT \quad T \to -E \quad T \to \lambda \quad V \to \mathbf{i}U \quad U \to (E) \quad U \to \lambda$$

Tracciare quindi la sequenza di parsing con input **i–i((i))** e con input **i-(()i)**.
Esercizio 4.10. Dimostrare che la seguente grammatica

$$S \to \mathbf{a}B \quad S \to \mathbf{a}C \quad S \to C \quad B \to \mathbf{b}B \quad B \to \mathbf{d} \quad C \to C\mathbf{c}B \quad C \to B\mathbf{b}B \quad C \to B$$

non è LL(1). Costruire una grammatica LL(1) equivalente alla precedente. Progettare quindi una tabella di parsing LL(1) per tale nuova grammatica e tracciare la sequenza di parsing con input **abdbd** e con input **abcbd**.