Pilu Crescenzi Alberto Marchetti Spaccamela

LINGUAGGI, AUTOMI, GRAMMATICHE

Dispense di Fondamenti di Informatica

Firenze, Roma 2016

Sommario

Prefazione

L'INFORMATICA utilizza linguaggi artificiali che sono progettati per facilitare la comunicazione con gli elaboratori elettronici e la loro programmazione. Sappiamo che, prima di essere eseguito, un programma scritto in un linguaggio (come Python) deve essere tradotto in un linguaggio comprensibile dall'elaboratore. Una delle cose più intriganti, per chi inizia lo studio dell'informatica, è scoprire che il processo di traduzione è effettuato 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 nella loro interezza va oltre lo scopo di queste dispense, il cui obiettivo è introdurre lo studio della teoria dei linguaggi da un punto di vista formale (argomento noto come *teoria dei linguaggi formali*). Cercheremo, pertanto, di definire prima che cosa sia un linguaggio, per poi vedere cosa significhi definire un linguaggio artificiale, come, ad esempio, Python o uno dei tanti altri linguaggi usati per programmare. Nel corso della dispensa cercheremo di dare risposta alla seguente domanda fondamentale.

> COME POSSIAMO FORMALIZZARE LE DEFINIZIONI DI UNA LINGUA ARTIFICIALE UTILE IN AMBITO INFORMATICO? IN ALTRE PAROLE, POSSIAMO TROVARE RAPPRESENTAZIONI CHE SIANO INTERPRETABILI MATEMATICAMENTE IN MODO DA PERMETTERCI DI RAGIONARE FORMALMENTE SU DI ESSE E PROGETTARE PROGRAMMI TRADUTTORI DA UN LINGUAGGIO ARTIFICIALE NEL LINGUAGGIO MACCHINA DI UN CALCOLATORE ELETTRONICO?

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. Quest'ulteriore obiettivo pone altre questioni rilevanti.

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

In queste dispense affronteremo da un punto di vista matematico lo studio dei linguaggi. Vedremo che ci sono diversi modi per farlo ognuno con i suoi vantaggi e svantaggi. Per questo non potremo limitarci a considerarne uno solo ma li considereremo insieme e li confronteremo fra loro.

I contenuti principali di queste dispense possono essere così riassunti.

> 1) È POSSIBILE DARE UNA DEFINIZIONE FORMALE (OVVERO, DI TIPO MATEMATICO) DI UN LINGUAGGIO USANDO LA TEORIA DEGLI INSIEMI.

In particolare, dato un alfabeto Σ, 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, la sequenza

```
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 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 (sarebbe come dire che possiamo scrivere solo un numero finito di programmi). 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 (ad esempio, se un determinato programma Python sia stato scritto correttamente). Pertanto dobbiamo sviluppare nuovi strumenti che costituiscono lo scopo principale di queste dispense. In particolare considereremo altri tre diversi approcci.

> 2) È POSSIBILE DEFINIRE UN LINGUAGGIO UTILIZZANDO UN APPROCCIO DI TIPO ALGEBRICO CHE UTILIZZA OPPORTUNE OPERAZIONI MATEMATICHE PER SPECIFICARE LE SEQUENZE INCLUSE NEL LINGUAGGIO STESSO.

Definiremo con quest'approccio una particolare classe di linguaggi, ovvero le espressioni regolari. In particolare, dato un alfabeto Σ definiremo un linguaggio utilizzando un'espressione di tipo algebrico che utilizza i caratteri dell'alfabeto Σ, 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. Quest'approccio ha il vantaggio di definire un linguaggio in modo sintetico, anche se il linguaggio contiene infinite sequenze. In questo modo potremo scrivere anche programmi Python che ricercano e manipolano testi.

> 3) È POSSIBILE SVILUPPARE ALGORITMI E PROGRAMMI CHE, DATO UN LINGUAGGIO, SIANO IN GRADO DI RICONOSCERE SE UNA SEQUENZA DI CARATTERI APPARTIENE AL LINGUAGGIO O MENO.

In particolare l'approccio algebrico ci permette di definire un linguaggio ma non ci permette di sapere se una data sequenza di caratteri appartiene o meno al linguaggio stesso. Per questo scopo abbiamo bisogno di algoritmi di riconoscimento che possiamo implementare in un programma (ad esempio, un compilatore per Python riceve 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 stati finiti che sono in grado di riconoscere tutti i linguaggi che possiamo definire con le espressioni regolari.

> 4) 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 i linguaggi 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 e così via) useremo il concetto di grammatica.

Abbiamo imparato a scuola come lo studio di una lingua naturale, come l'italiano o l'inglese, richieda lo studio della sua grammatica. Le regole grammaticali di un linguaggio come l'italiano sono moltissime e di difficile formalizzazione in termini matematici. Tuttavia, agli inizi del secolo scorso il grande studioso dei linguaggi Noam Chomsky formalizzò il concetto di grammatica 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 definizione 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 quest'approccio è la possibilità di definire in modo sintetico linguaggi complessi, mentre lo svantaggio è che il problema del riconoscimento per questi linguaggi è molto più complesso che nel caso delle espressioni regolari. Lo studio approfondito di queste problematiche va oltre lo scopo di queste dispense e per questa ragione ci limiteremo a considerarne solo alcuni aspetti, che speriamo diano al lettore un'idea abbastanza abbastanza chiara delle problematiche che devono essere affrontate quando si vuole sviluppare un compilatore.

Riferimenti bibliografici Queste dispense non presuppongono particolari conoscenze da parte dello studente, a parte quelle relative a nozioni matematiche e logiche di base. Esistono numerosi libri di testo dedicati agli argomenti trattati in queste dispense: in particolare, i seguenti due volumi hanno ispirato in parte la stesura delle dispense stesse e sono fortemente consigliati come letture aggiuntive.

- G. Ausiello, F. D'Amore, G. Gambosi. *Linguaggi, modelli, complessità*, Franco Angeli, 2003.
- P. Crescenzi *Informatica Teorica*, `http://piluc.dsi.unifi.it`, 2011.

A chi poi volesse approfondire le nozioni relative alla teoria dei linguaggi formali e allo sviluppo di compilatori, suggeriamo la lettura dei seguenti due libri.

- J.E. Hopcroft, R. Motwani, J.D. Ullman. *Automi, linguaggi e calcolabilità*, Pearson, 2009.
- A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. *Compilatori. Principi, tecniche e strumenti*, Pearson, 2009.

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. Introdurremo, poi, i possibili approcci allo studio formale dei linguaggi con particolare riferimento ai linguaggi di programmazione.

1.1 Sintassi e semantica di un linguaggio

IN QUESTO capitolo 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, in 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). È 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 delle specifiche regole, a partire da un certo insieme di componenti che abbiano un senso strutturale. Ad esempio, una frase in italiano ha spesso la forma seguente:

```
<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 *IL TOPO IL GATTO IL 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 contraddittorie o frasi sintatticamente corrette che non hanno alcun significato. L'esempio classico è la seguente frase di Chomsky (studioso che, come abbiamo citato nella prefazione, 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 $x = 0$ e $y = 0$ allora $0/0$ è un'espressione matematica sintatticamente corretta 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 (ovvero, a "run-time") quando il calcolatore interpreta il codice compilato. Purtroppo non abbiamo a disposizione uno strumento analogo al compilatore che sia in grado di individuare gli errori semantici di un programma. 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 due componenti che sembrano essenziali per la definizione di un linguaggio, sia naturale che artificiale.

- Un insieme 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 i caratteri dell'alfabeto italiano e i segni di punteggiatura.

- 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 perché le frasi di un linguaggio e, quindi, le 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 realtà piuttosto semplice: per stabilire se una frase è corretta o meno è sufficiente osservare che la *grammatica* di un linguaggio permette di definire le regole che stabiliscono se una frase 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 decimali $\{0,1,2,3,4,5,6,7,8,9\}$,
- 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: stringhe

- Dato un alfabeto Σ, una sequenza finita di simboli di Σ è una *stringa* (o parola).
- Data una stringa x, $|x|$ rappresenta il numero di simboli 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 indicata con 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 con Σ^*.

Esempio 1.1: stringhe

- 01000010 è una stringa definita sull'alfabeto binario $\{0,1\}$ di lunghezza 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 italiano e le due parole w_1= salva e w_2= gente, allora 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. Ad esempio, se w_1= abb e w_2=bba, allora $((w_1w_2)w_1)$ = abbbbaabb = $(w_1(w_2w_1))$ e w_1w_2 = abbbba $\neq w_2w_1$= bbaabb. Osserviamo, inoltre, che, concatenando una stringa con la stringa vuota ϵ, otteniamo la stringa stessa, ovvero, per ogni stringa w, $w\epsilon = w = \epsilon w$. Per brevità, per indicare la ripetizione di simboli o, più in generale, la concatenazione di due o più stringhe uguali, si usa il simbolo di potenza. Ad esempio, $ab^4a = abbbba$, $x^3 = xxx$, e se w=cous, allora w^2= couscous.

Definizione 1.2: linguaggi

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

Dato un qualunque alfabeto Σ possiamo affermare che:

- Σ stesso è un linguaggio le cui stringhe sono gli elementi dell'alfabeto (ad esempio, se Σ è l'alfabeto binario, allora $\{0,1\}$ è un linguaggio definito su Σ);
- Σ^* è un linguaggio formato da tutte le possibili stringhe che si possono ottenere da Σ (ad esempio, nel caso dell'alfabeto binario, Σ^* contiene la stringa vuota e tutte le infinite sequenze di 0 e 1 di qualunque lunghezza);
- l'insieme che non contiene nessuna stringa, denotato con $\emptyset$ è un linguaggio, detto linguaggio vuoto (osserviamo che $\emptyset \neq \{\epsilon\}$, in quanto $\emptyset$ denota il linguaggio che non contiene nessuna stringa, mentre $\{\epsilon\}$ è il linguaggio che contiene solo la stringa vuota).

Esempio 1.2: linguaggi

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 simboli a (con $n \geqslant 0$), seguita da altrettanti simboli b. Pertanto $aaabbb \in L$, $aaabb \notin L$ e $\epsilon \in l$.
- Dato l'alfabeto $\{I,V,X,L,C,D,M\}$, l'insieme di tutti i numeri da 1 a 3000 rappresentati come numeri romani è un linguaggio definito su tale alfabeto.
- Dato l'alfabeto $\{0,1\}$, l'insieme di tutte le stringhe binarie che contengono un numero pari di 1 è un linguaggio definito su tale alfabeto.

1.3 Approcci formali allo studio dei linguaggi

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

1. Il linguaggio costituito da stringhe di parentesi bilanciate del tipo $(()(()))()$. Tale linguaggio non include stringhe di parentesi come $(()$ (in cui ci sono due parentesi aperte e una sola chiusa) oppure come $)(()))($ (in cui le parentesi non sono bilanciate).
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 i linguaggi di nostro interesse. In particolare, lo studio formale dei linguaggi utilizza tre diversi approcci:

1. un approccio *algebrico*, che mostra come costruire un linguaggio 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.

Nel prossimo capitolo considereremo questi diversi approcci. In particolare, nel capitolo 2 introduciamo le *espressioni regolari*, utilizzando l'approccio algebrico. Le espressioni regolari individuano una classe di linguaggi che hanno una grande 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) e 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 Capitolo 3 consideriamo anche l'approccio riconoscitivo presentando gli *automi a stati finiti*, semplici macchine in grado di riconoscere se una stringa appartiene al linguaggio definito da un'espressione regolare. Per introdurre gli automi a stati finiti introduciamo preliminarmente le macchine di Turing, un importante formalismo di calcolo teorico proposto negli anni venti 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 vedremo 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 problematiche relative alla traduzione è oltre gli scopi di questa dispensa. Nel Capitolo 5 ci limitiamo a introdurre alcune problematiche discutendo in particolare il concetto di grammatica ambigua.

Capitolo 2

Le espressioni regolari

SOMMARIO

Le espressioni regolari permettono di definire in modo semplice linguaggi. Questi linguaggi, anche se non hanno la capacità espressiva di un linguaggio di programmazione come Python, sono spesso utilizzati nella programmazione.

2.1 Espressioni regolari

LE ESPRESSIONI regolari sono un modo semplice per capire se una stringa ha un certo formato. Per esempio, quando compiliamo la registrazione su una pagina web dobbiamo fornire le nostre generalità. Se ci viene chiesto di indicare tra le altre cose la data di nascita, dobbiamo fornirla secondo un preciso formato. Ad esempio, se ci viene indicato di fornire la data di nascita secondo il formato

`GG/MM/AAAA`,

allora vuol dire che, se siamo nati l'1 Gennaio del 1997, dobbiamo utilizzare due cifre per il giorno (anche se si tratta del numero 1), due cifre per il mese (anche se si tratta del numero 1) e quattro cifre per l'anno. In altre parole, dobbiamo scrivere

`01/01/1997`.

Ogni altro formato, come ad esempio `1/1/1997` oppure `01/01/97`, è errato. Le espressioni regolari sono uno strumento con cui formalmente definiamo il formato delle stringhe e ci permette di riconoscere che `01/01/1997` rispetta la definizione data, mentre questo non è vero per gli altri formati.

Definizione 2.1: 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.
- ϵ (la stringa vuota) 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.
- Le espressioni regolari sono formate utilizzando queste regole ed eventualmente le parentesi tonde $(,)$ per stabilire le priorità fra le operazioni.

Il passo successivo consiste nel definire in modo preciso l'insieme di stringhe che soddisfano un'espressione regolare. Assumendo che l'insieme dei caratteri sia Σ, le stringhe che collimano una certa espressione regolare R sono denotate con $L(R)$.

Definizione 2.2: linguaggio definito da un'espressione regolare

Un'espressione regolare R genera il linguaggio $L(R)$ definito in modo induttivo come segue.

- Se $R = a \in \Sigma$, allora $L(R) = \{a\}$.
- Se $R = \epsilon$, allora $L(R) = \{\epsilon\}$.
- Se $R = S_1 S_2$, 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 $L(R) = \{x_1 x_2 \ldots x_n : n \geqslant 0 \wedge x_i \in L(S)\}$.

Esempio 2.1: esempi di espressioni regolari

Le seguenti sono espressioni regolari sull'alfabeto $\Sigma = \{a, b\}$.

- a è un'espressione regolare composta dalla sola a e definisce il linguaggio $L = \{a\}$ che contiene una sola stringa di una singola a.
- $aabb$ (abbreviazione di $a \cdot a \cdot b \cdot b$) rappresenta la concatenazione di quattro espressioni regolari, le prime due costituite ciascuna da una a e le seconde due costituite ciascuna da una b; $aabb$ definisce il lingaggio $L = \{aabb\}$ che contiene una sola stringa.
- $a|b$ rappresenta la selezione di due espressioni regolari, la prima costituita da una a e la seconda costituita da una b e e definisce il linguaggio $L = \{a, b\}$ che contiene due sole stringhe a e b;
- $a|(b^*)$ rappresenta la selezione di due espressioni regolari, la prima costituita da una a e la seconda costituita dalla chiusura di Kleene dell'espressione regolare costituita da una b; $a|b$ definisce il linguaggio $L = \{a, \epsilon, b, bb, bbb, bbbb, \ldots\}$ che contiene la stringa vuota, la stringa a e le stringhe di sole b aventi lunghezza qualunque.
- $(a|b)^*$ rappresenta la chiusura di Kleene dell'espressione regolare $a|b$, la quale a sua volta rappresenta la selezione di due espressioni regolari, la prima costituita da una a e la seconda costituita da una b; $(a|b)^*$ definisce il linguaggio che contiene la stringa vuota e le stringhe di a e b di lunghezza qualunque.
- $b^*(a|\epsilon)$ (abbreviazione di $b^* \cdot (a|\epsilon)$) rappresenta la concatenazione dell'espressione regolare b^* (ovvero, la chiusura di Kleene dell'espressione regolare costituita da una b) e dell'espressione regolare $a|\epsilon$ (ovvero la selezione dell'espressione regolare costituita da una a e l'espressione regolare costituita da ϵ, la stringa vuota); $b^*(a|\epsilon)$ definisce il linguaggio che contiene la stringa vuota, la stringa a, le stringhe di sole b di lunghezza qualunque e le stringhe che iniziano con un numero qualunque di b e terminano con a.

Due espressioni regolari R e S sono equivalenti se $L(R) = L(S)$. Osserviamo che sia l'operazione di concatenazione che quella di selezione sono associative e che, pertanto, $R(ST)$, $(RS)T$ e RST sono equivalenti, così come $R|(S|T)$, $(R|S)|T$ e $R|S|T$.

Nell'analizzare un'espressione regolare, supporremo nel seguito che la chiusura di Kleene (ovvero l'operazione *) abbia la massima priorità e che la selezione (ovvero l'operazione $|$) abbia la minima priorità (le parentesi verrano usate per annullare tali priorità nel modo usuale). Pertanto abbiamo che valgono, ad esempio, le seguenti uguaglianze e disuguaglianze.

- $L((ab)^*) \neq L(ab^*) = L(a(b^*))$. Infatti, $L((ab)^*)$ contiene le stringhe formate da zero o più copie di ab, mentre $L(ab^*)$ contiene le stringhe formate da una a seguita da zero o più copie di b.

- $L(a(b|c)) \neq L(ab|c) = L((ab)|c)$. Infatti, $L(a(b|c)) = \{ab, ac\}$, mentre $L(ab|c) = \{ab, c\}$.
- $L((a|b)^*) \neq L(a|b^*) = L(a|(b^*))$. Infatti, $L((a|b)^*)$ contiene tutte le stringhe formate da a e b (inclusa la stringa vuota ϵ), mentre $L(a|b^*)$ contiene la stringa a e le stringhe formate da zero o più copie di b.

Osserviamo anche che la chiusura di Kleene consente zero concatenazioni, per cui il linguaggio generato dall'espressione regolare R^* contiene la stringa vuota ϵ.

Esempio 2.2: linguaggi definiti da espressioni regolari

Consideriamo le seguenti espressioni regolari.

- $R = a|(b^*)$. Abbiamo già osservato che $L(R) = \{\epsilon, a, b, bb, bbb, \ldots\}$.
- $R = (a|b)^*$. Abbiamo già osservato che $L(R) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \ldots\}$.
- $R = ab^*(c|\epsilon)$. In questo caso, $L(R)$ contiene le stringhe che iniziano con a, proseguono con zero o più copie di b e terminano con una c opzionale. Ovvero, $L(R) = \{a, ac, ab, abc, abb, abbc, \ldots\}$.
- $R = (0|(1(01^*0)^*1))^*$. In questo caso, $L(R)$ contiene le stringhe binarie che rappresentano numeri non negativi multipli di 3. Ovvero,

$$L(R) = \{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, 1100, 1111, 00000, \ldots\}.$$

Esempio 2.3: un'espressione regolare per esprimere le date

Vediamo ora l'espressione regolare D che genera il linguaggio delle stringhe corrispondenti a una data con un formato del tipo `GG/MM/AAAA` dove `GG`, `MM` e `AAAA` rappresentano rispettivamente il giorno, il mese e l'anno. Innanzitutto, l'alfabeto in questo caso deve includere le cifre decimali e il simbolo `/`. Ovvero,

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, /\}.$$

Per semplicità di esposizione, procediamo per passi e definiamo anzitutto le seguenti due espressioni regolari che producono una qualunque cifra compresa tra 1 e 9 e tra 0 e 9, rispettivamente:

$$P = 1|2|3|4|5|6|7|8|9 \qquad e \qquad Z = 0|1|2|3|4|5|6|7|8|9$$

Osserviamo che, se non teniamo conto dei diversi giorni dei mesi e degli anni bisestili, il giorno deve essere un numero compreso tra 1 e 31. Possiamo quindi definire la seguente espressione regolare per generare l'insieme delle stringhe corrispondenti a un giorno di un mese:

$$G = 0(1|2|3|4|5|6|7|8|9) \mid (1|2)(0|1|2|3|4|5|6|7|8|9) \mid 3(0|1)$$

Analogamente, la seguente espressione regolare genera l'insieme delle stringhe corrispondenti a un mese (ovvero un numero compreso tra 1 e 12):

$$M = 0(1|2|3|4|5|6|7|8|9) \mid 1(0|1|2)$$

Infine, la seguente espressione regolare genera l'insieme delle stringhe corrispondenti a un anno (espresso con quattro cifre):

$$A = (0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)$$

Concludendo, l'espressione regolare per l'intera data è la seguente: $D = G/M/A$ in particolare
$D = (0(1|2|3|4|5|6|7|8|9) \mid (1|2)(0|1|2|3|4|5|6|7|8|9) \mid 3(0|1)) \mid (0(1|2|3|4|5|6|7|8|9) \mid 1(0|1|2))$
$|(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)$

2.2 Espressioni regolari in Python

Python permette di definire espressioni regolari per facilitare la ricerca e la manipolazione di archivi. Vedremo nel seguito alcuni esempi applicativi. Prima però di proseguire vediamo come le espressioni regolari in Python sono definite in modo diverso.

Prima di proseguire spieghiamo come mai i progettisti di Python hanno deciso di usare un modo diverso di scrivere le espressioni regolari rispetto a quello introdotto nel paragrafo precedente. In particolare, invece di introdurre le espressioni regolari usando le tre operazioni presentate nel paragrafo precedente, Python introduce molte più operazioni. Ci possiamo chiedere quale sia la ragione di tale apparente complicazione. In realtà, i motivi sono pratici, in quanto, introducendo altri operatori, possiamo definire le espressioni regolari in modo più compatto.

Per esempio, un codice fiscale è composto da tre lettere maiuscole, uno spaio, altre tre maiuscole, uno spazio, due cifre, e così via. In Python possiamo scrivere l'espressione regolare di un codice fiscale nel seguente modo intuitivo.

$$[A-Z]\{3\}\ [A-Z]\{3\}\ [0-9]\{2\}[A-Z][0-9]\{2\}\ [A-Z][0-9]\{3\}[A-Z]$$

Il senso di questa espressione è il seguente. Un codice fiscale è una lettera maiuscola ($[A-Z]$) ripetuta tre volte ($\{3\}$), seguita da uno spazio, da altre tre lettere maiuscole (ancora $[A-Z]\{3\}$), da uno spzio, da due cifre ($[0-9]\{2\}$), e così via. Come si vede, $[A-Z]$ indica una singola lettera maiuscola, ossia una lettera "compresa" fra A e Z. La costruzione $[A-Z]\{3\}$ indica una sequenza di tre lettere maiuscole. Nello stesso modo, $[0-9]$ è una singola cifra, mentre $[0-9]\{2\}$ è una sequenza di due cifre.

In Python, è possibile verificare se una stringa ha il formato sintattico di un codice fiscale nel seguente modo.

```
1 import re
2 e = '[A-Z]{3} [A-Z]{3} [0-9]{2}[A-Z][0-9]{2} [A-Z][0-9]{3}[A-Z]'
3 s = 'RLT FRL 32B12 H501R'
4 m = re.search(e, s)
5 if m:
6   print("codice fiscale")
7 else:
8   print("non codice fiscale")
```

L'espressione specifica il formato sintattico del codice fiscale, ma non, per esempio, che l'ultimo carattere debba essere ottenuto da un calcolo effettuato con i precedenti (non viene altresì considerato il caso di codici alterati per *omocodia*, cioè di due persone diverse ma con lo stesso codice fiscale[1]).

È possibile modificare l'espressione di sopra, specificando che il primo gruppo di tre lettere più uno spazio va ripetuto due volte, nel modo seguente:

$$([A-Z]\{3\}\ )\{2\}[0-9]\{2\}[A-Z][0-9]\{2\}\ [A-Z][0-9]\{3\}[A-Z]$$

Come per le espressioni algebriche, è possibile quindi usare le parentesi per indicare un gruppo come $([A-Z]\{3\}\ )$ su cui è possibile poi applicare gli operatori come la ripetizione $\{2\}$, ottenendo così $([A-Z]\{3\}\ )\{2\}$, che ha il significato atteso: una sequenza di tre lettere seguita da uno spazio, tale sequenza ripetuta due volte.

Altri esempi di espressioni regolari in Python sono i seguenti.

- `codice=.{10}`
  indica la stringa `codice=` seguita da dieci caratteri qualsiasi. Il punto . indica infatti un carattere

[1] Il codice fiscale è di norma definito usando le prime tre lettere (possibilmente consonanti) del nome e del cognome, la data di nascita e il luogo di nascita. Per questa ragione due persone diverse, Gaspare Rossi e Giuseppe Rossini, se sono nati lo stesso giorno nella stessa città, avrebbero lo stesso codice fiscale; quindi la regola generale in questo caso non può essere applicata perché darebbe lo stesso codice a due persone diverse.

qualsiasi, mentre {10} indica una sequenza di dieci (quindi dieci caratteri qualsiasi, anche diversi fra loro).

- `-?[0-9]+`
  indica un numero intero con un possibile segno −. Infatti, all'inizio − è opzionale (l'operatore ? indica l'opzionalità di ciò che precede) ed è seguito da una sequenza di almeno una cifra (il + indica la ripetizione un numero illimitato di volte).

- `[A-Z][a-z]+ [A-Z][a-z]+`
  indica ad esempio il nome e il cognome semplici (cioè composti da una sola parola). Infatti la prima parte `[A-Z][a-z]+` specifica un carattere maiuscolo seguito da almeno un carattere minuscolo, da uno spazio e poi da un cognome (analogo al caso precedente).

- `[0-9]{1,2} (gennaio|febbraio|marzo|aprile|maggio |giugno|luglio|agosto|settembre|ottobre|novembre |dicembre) [0-9]{4}`
  l'operatore | indica una scelta fra diverse possibilità e, quindi, l'espressione regolare indica una data. Infatti {1,2} indica una sequenza di una o due cifre, segue uno spazio, una stringa qualsiasi scelta fra `gennaio, febbraio, ...`, uno spazio e altre quattro cifre.

Nell'ultimo esempio non vengono fatti controlli sintattici complessi come il fatto che, se la prima cifra è 3, allora la successiva deve essere 0 oppure 1. Questo è possibile con espressioni regolari ma in modo più complicato, ovvero nel modo seguente: $[1-9]|[12][0-9]|3[01]$. Quest'espressione indica una scelta (espressa dal segno |) fra le tre espressioni $[1-9]$, che indica una singola cifra maggiore di 0, $[12][0-9]$, che è una cifra fra uno e due seguita da una cifra qualsiasi, e $3[01]$, ossia il tre seguito da zero o uno. Anche con questa variante, non esiste un controllo che escluda una data come `31 aprile 2001` (aprile ha trenta giorni) oppure `29 febbraio 2011` (il 2011 non era un anno bisestile). Questo tipo di controlli è semantico, ossia dipendente non dal formato della stringa ma dal significato delle sue parti. Nel caso specifico, la sequenza di lettere centrale indica un mese, e il particolare mese implica dei vincoli sul numero del giorno.

2.2.1 Operatori utilizzati in Python

Gli operatori delle espressioni regolari sono i seguenti.

1. Il punto, '.', indica un carattere qualsiasi.

2. Le parentesi quadre [ ] indicano una scelta fra caratteri singoli inclusi fra le parentesi ([129] è uno oppure due oppure nove) oppure in intervalli ($[c-m]$ è un singolo carattere fra c e m).

3. Il punto interrogativo ? indica opzionalità (ad esempio, a? indica che la a può comparire oppure no).

4. Il simbolo di addizione + indica la ripetizione di almeno una volta (ad esempio, a+ indica a oppure aa oppure aaa, e così via).

5. L'asterisco $*$ indica la ripetizione un numero qualunque di volte (anche zero) (ad esempio, $ab*c$ include abc, abbc e abbbc ma anche ac).

6. La notazione $\{n,m\}$, dove n e m sono numeri interi positivi, specifica un numero di ripetizioni compreso fra n e m. Quindi $ab\{2,4\}c$ include abbc (due ripetizioni di b), abbbc (tre ripetizioni) e abbbbc (quattro ripetizioni). Osserviamo che sia n che m possono non apparire, e allora si assumono zero e infinito, rispettivamente. Quindi $\{10,\}$ indica un numero di volte maggiore o uguale a 10, mentre $\{,5\}$ rappresenta un numero di volte da 0 a 5. Si noti, infine, che la forma $\{n\}$ è equivalente a $\{n,n\}$.

7. La barra verticale | indica la scelta fra due o più casi (ad esempio, entrata|uscita specifica che la stringa può essere o "entrata" oppure "uscita").

8. Le parentesi tonde () indicano l'operatore di gruppo che permette di raggruppare parti dell'espressione regolare. L'utilizzo di questo operatore permette di stabilire l'ordine con cui devono essere eseguite le operazioni. Infatti osserviamo che gli altri operatori di norma agiscono sull'ultimo carattere o intervallo; con le parentesi è possibile applicare gli operatori a una qualsiasi espressione regolare. Ad esempio, si consideri di voler verificare se una stringa è una sequenza di zeri e uni intervallati da spazi, come, per esempio, 0 1 1 0 1 1 1 0. La singola cifra binaria è ovviamente [01] (o anche 0|1) per cui una cifra seguita da uno spazio è "[01] ". Applicando l'operatore + si otterrebbe "[01] +", che però indica che solo lo spazio va ripetuto da uno a più volte. Quindi la stringa "0 " verrebbe erroneamente accettata mentre "0 1" no. Usando le parentesi intorno a "[01] " si ottiene ([01]), in modo tale che qualsiasi operatore vada considerato applicato al tutto. Per esempio, ([01])+ indica che la sequenza "zero o uno seguito da spazio" si può ripetere un numero arbitrario di volte. Viene quindi accettata la stringa '0 1 0 0 1 ' ma non '0 ', come previsto. Dato che lo spazio alla fine non ci va, l'espressione va poi modificata in ([01])*[01], che indica che la sequenza formata da un bit seguito da uno spazio viene ripetuta da zero a un numero qualsiasi di volte, ma che alla fine occorre che ci sia un singolo bit non seguito da spazio.

2.2.2 Sottostringhe, divisioni e sostituzioni

Vediamo ora funzioni che permettono di ricercare stringhe all'interno di testi e modificarli.

`search`
La funzione `re.search(espressione,stringa)` cerca all'interno del parametro `stringa` una sequenza di caratteri coerente con l'espressione regolare rappresentata da `espressione`. Per esempio, `re.search('a+', 'ccdaaeee')` dà risultato positivo, dal momento che la stringa contiene (in mezzo) `aa`, che è nella forma prevista dall'espressione `a+`, ossia `a` ripetuta una o più volte. Se si volesse verificare se l'intera stringa soddisfa l'espressione regolare, si aggiunge all'inizio e alla fine dell'espressione i due caratteri speciali `^` e `$`, che indicano l'inizio e la fine della stringa. Si può anche usare uno solo di questi due caratteri. I seguenti esempi illustrano l'utilizzo di questi operatori.

- `^abcd$` indica che l'intera stringa deve coincidere con abcd; pertanto non vengono accettate le stringhe xxxabcd, abcdxxx e nemmeno 00abcd00.
- `^titolo`: indica una stringa che inizia con la sequenza di caratteri titolo:. Viene quindi accettata la stringa "titolo: I promessi sposi", ma non viene accettata la stringa "questo titolo: I promessi sposi" perché la stringa 'titolo:' non è all'inizio.
- `[1-9]$` indica una stringa che finisce con una cifra diversa da zero, per cui '120' non sarebbe accettata anche se ha cifre diverse da zero, perché non lo è l'ultima.

La ricerca di sottostringhe che collimano con espressioni regolari consente di effettuare altre operazioni, come la divisione della stringa e la sostituzione. In particolare per questo scopo si utilizzano le due funzioni `split` e `sub`.

`split`
Questa funzione divide una stringa in parti, usando come punti di divisione le sottostringhe che collimano con l'espressione regolare. Per esempio, la divisione di `'abcd;efgh:wsl'` usando l'espressione regolare `[;:]` effettuata con la seguente sequenza di istruzioni

```
1 import re
2 p = re.compile('[;:]')\\
3 print p.split('abcd;efgh:wsl')}
```

produce la lista [abcd, efgh, wsl]. Infatti la prima stringa è composta dall'inizio fino al simbolo ';' escluso. La seconda e la terza sono la sottostringa dai simboli ';' fino a ':', e la sottostringa che segue il simbolo ':'.

`sub`
Questa funzione rimpiazza le sottostringhe che collimano con l'espressione regolare con un'altra stringa. Per esempio, il seguente frammento di programmma rimpiazza con Gianni ogni sottostringa nella forma (Aldo|Luca|Anna) (in altre parole sostituisce Gianni ogni volta che compare Aldo, Luca o Anna).

```
1 import re
2 p = re.compile('(Aldo|Luca|Anna)')
3 print(p.sub('Gianni', "Questo l'ha fatto Aldo, questo Anna, questo Luca"))
```

Osserviamo che per le due funzioni precedenti, la sottostringa è sempre quella più lunga possibile che rispetta l'espressione regolare. Ad esempio, se si vuole cambiare il nome di una persona con 'luca' in una stringa che contiene nome e cognome in minuscolo, si può pensare di usare la seguente espressione regolare per identificare il nome: `^[a-z]+[ ]` . Infatti l'espressione denota una sequenza di almeno una lettera minuscola seguita da uno spazio, tutto questo all'inizio della stringa. Questo funziona, per esempio, nel caso di 'marco ranetti', dato che l'unica sottostringa che soddisfa l'espressione è 'marco ', cioè il nome seguito dallo spazio. Non è però così con 'giorgio michele gioveretti lunardi', in cui il nome è doppio. Si può pensare di risolvere il problema inserendo lo spazio fra i caratteri del nome (usando l'espressione `[a-z ]*[ ]`) oppure ripetendo l'intera espressione almeno una volta (ovvero, `([a-z]+[ ])+`). In entrambi i casi, più sottostringhe collimano con l'espressione regolare:

```
'giorgio '
'giorgio michele '
'giorgio michele gioveretti '
```

Pertanto il codice

```
1 import re
2 p = re.compile('([a-z]+[ ])+')
3 print(p.sub('luca', "giorgio michele gioveretti lunardi"))
```

produce la stringa `'lucalunardi'`, dato che si rimpiazza la sottostringa più lunga. Viene quindi cambiato non solo il nome doppio ma anche il primo cognome.

Esercizi

Esercizio 2.1. Scrivere l'espressione regolare che collima con tutte le stringhe composte solo da caratteri a,b e c e che iniziano oppure terminano con a.

Esercizio 2.2. Riscrivere l'espressione regolare $aa|aab|aac|aabc$ senza usare l'operatore |.

Esercizio 2.3. Dimostrare che la concatenazione non è commutativa.

Esercizio 2.4. 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 2.5. Data l'espressione regolare $R = ab*|aab+$ determinare l'espressione regolare R' che collima con le stringhe invertite. Ad esempio se ab collima con R allora ba collima con R' e viceversa.

Esercizio 2.6. Scrivere l'espressione regolare che identifica l'ora del giorno espressa come HH:MM. Tenere presente che le ore arrivano a 23 e i minuti a 59.

Esercizio 2.7. In un file di testo si vogliono cercare tutte le frasi che contengono l'articolo "il" e terminano con "vero". Scrivere l'espressione regolare necessaria.

Esercizio 2.8. Nel file prova.html le immagini sono identificate da stringhe del tipo `<img src="..." />` dove fra virgolette c'è la URL dell'immagine (una stringa che non contiene virgolette). Completare il seguente programma Python che dovrebbe stampare queste URL.

```
1 import re
2 s = open('prova.html').read()
3 l = re.findall(..., s)
4 for i in l:
5   print i
```

Esercizio 2.9. Dire quante e quali stringhe collimano con l'espressione regolare in Python ^a{2,3}b?—b[ca]$. Si ricorda che i simboli ^ e $ nelle espressioni regolari Python indicano l'inizio e e la fine della stringa.

Capitolo 3

Macchine di Turing e automi a stati finiti

SOMMARIO

In questo capitolo introduciamo inizialmente la macchina ideale proposta da Alan Turing. Successivamente studieremo una restrizione delle macchine di Turing, gli automi a stati finiti che leggono in input una stringa e, al termine della lettura, decidono se accettare o meno la stringa. Infine studieremo le relazioni tra automi a stati finiti ed espressioni regolari.

3.1 La macchina di Turing

UNA MACCHINA di Turing è un modello di calcolo abbastanza simile agli odierni calcolatori e proposta da Alan Turing negli anni venti del secolo scorso. L'obiettivo di Turing era proporre un metodo ideale di calcolo per comprendere le possibilità e i limiti del calcolo automatico. Per questa ragione Turing non realizzò la sua macchina. Sorprendentemente però gli studi di Turing hanno avuto una profonda influenza nell'informatica.

Analogamente ai calcolatori moderni, la macchina di Turing possiede un'unità di elaborazione centrale (CPU, dall'inglese *Central Processing Unit*) e una memoria su cui poter leggere e scrivere (si veda la Figura 3.1). In particolare, la CPU di una macchina di Turing è composta da un **registro di stato**, contenente lo stato attuale della macchina, e da un **programma** contenente le istruzioni che essa deve eseguire.[1]. La memoria di una macchina di Turing è composta da un **nastro** infinito, suddiviso in **celle** e al quale la CPU può accedere attraverso una **testina** di lettura/scrittura.

Inizialmente, il nastro contiene la stringa di **input** preceduta e seguita da una serie infinita di simboli vuoti (in queste dispense, il **simbolo vuoto** è indicato con □), la testina è posizionata sul primo simbolo della stringa di input e la CPU si trova in uno stato speciale, detto **stato iniziale**.

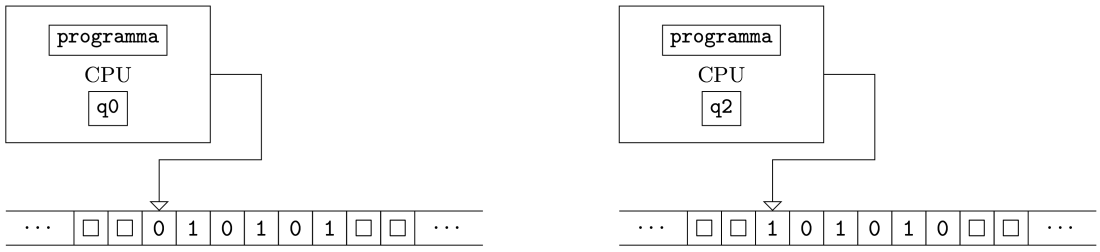
Sulla base dello stato in cui si trova la CPU e del simbolo letto dalla testina, la macchina esegue un'istruzione del programma che può modificare il simbolo attualmente scandito dalla testina, spostare la testina a destra oppure a sinistra e cambiare lo stato della CPU. La macchina prosegue nell'esecuzione del programma fino a quando la CPU non viene a trovarsi in uno di un insieme di stati particolari, detti **stati finali**, oppure fino a quando non esistano istruzioni del programma che sia possibile eseguire.

Nel caso in cui il programma termini perché la CPU ha raggiunto uno stato finale, il contenuto della porzione di nastro racchiusa tra la posizione attuale della testina e il primo simbolo □ alla sua destra rappresenta la stringa di **output**.

Pur nella sua semplicità, la macchina di Turing risulta essere un modello di calcolo molto potente, al punto che Turing stesso dimostrò come fosse possibile con tale modello sviluppare quello che oggigiorno chiameremmo sistema operativo e che, all'epoca, Turing chiamò macchina universale. Non studieremo, in queste dispense, il potere computazionale delle macchine di Turing, e rinviamo il lettore interessato a consultare i libri di testo a cui abbiamo fatto riferimento alla fine della prefazione.

[1] Una macchina di Turing esegue un particolare programma, ma possiamo definire infinite macchine di Turing così come possiamo scrivere infiniti programmi Python.

Figura 3.1: rappresentazione schematica di una macchina di Turing.



Esempio 3.1: una macchina di Turing per il complemento bit a bit

Consideriamo una macchina di Turing la cui CPU può assumere tre possibili stati `q0`, `q1` e `q2`, di cui il primo è lo stato iniziale e l'ultimo è l'unico stato finale. L'obiettivo della macchina è quello di calcolare la stringa binaria ottenuta eseguendo il complemento bit a bit della stringa binaria ricevuta in input: il programma di tale macchina è il seguente.

1. Se lo stato è `q0` e il simbolo letto non è □, allora complementa il simbolo letto e sposta la testina a destra.
2. Se lo stato è `q0` e il simbolo letto è □, allora passa allo stato `q1` e sposta la testina a sinistra.
3. Se lo stato è `q1` e il simbolo letto non è □, allora sposta la testina a sinistra.
4. Se lo stato è `q1` e il simbolo letto è □, allora passa allo stato `q2` e sposta la testina a destra.

La prima istruzione fa sì che la macchina scorra l'intera stringa di input, complementando ciascun bit incontrato, mentre le successive tre istruzioni permettono di riportare la testina all'inizio della stringa modificata. Ad esempio, tale macchina, con input la stringa binaria `010101`, inizia la computazione nella configurazione mostrata nella parte sinistra della Figura 3.1 e termina la sua esecuzione nella configurazione mostrata nella parte destra della figura.

Formalmente, una macchina di Turing è definita specificando l'insieme degli stati (indicando quale tra di essi è quello iniziale e quali sono quelli finali) e l'insieme delle transizioni da uno stato a un altro: questo ci conduce alla seguente definizione.

Definizione 3.1: macchina di Turing

Una *macchina di Turing* è costituita da un **alfabeto di lavoro** Σ contenente il simbolo □ e da un **grafo delle transizioni**, ovvero un grafo etichettato $G = (V, E)$ tale che:

- $V = \{q_0\} \cup F \cup Q$ è l'insieme degli **stati** (q_0 è lo stato **iniziale**, F è l'insieme degli stati **finali** e Q è l'insieme degli stati che non sono iniziali né finali);
- E è l'insieme delle **transizioni** e, a ogni transizione, è associata un'etichetta formata da una lista l di triple (σ, τ, m), in cui σ e τ sono simboli appartenenti a Σ e $m \in \{R, L, S\}$, tale che non esistono due triple in l con lo stesso primo elemento.

I simboli σ e τ che appaiono all'interno di una tripla dell'etichetta di una transizione indicano, rispettivamente, il simbolo attualmente letto dalla testina e il simbolo da scrivere, mentre il valore m specifica il movimento della testina: in particolare, `R` corrisponde allo spostamento a destra, `L` allo spostamento a sinistra e `S` a nessun spostamento. Nel seguito, per evitare di dover specificare ogni volta quale sia lo stato iniziale e quali siano gli stati finali di una macchina di Turing adotteremo la convenzione di specificare, nella rappresentazione grafica del grafo delle transizioni, lo stato iniziale affiancando a esso una freccia rivolta verso destra e di rappresentare gli stati finali con un doppio cerchio.

Grafi

Un **grafo** G è una coppia di insiemi finiti (V, E) tale che $E \subseteq V \times V$: V è l'insieme dei **nodi**, mentre E è l'insieme degli **archi**. Se $(u, v) \in E$, allora diremo che u è **collegato** a v: il numero di nodi a cui un nodo x è collegato è detto **grado in uscita** di x, mentre il numero di nodi collegati a x è detto **grado in ingresso** di x. Un grafo $G' = (V', E')$ è un **sotto-grafo** di un grafo $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$. Un grafo **etichettato** è un grafo $G = (V, E)$ con associata una funzione $l : E \to L$ che fa corrispondere a ogni arco un elemento dell'insieme L delle etichette: tale insieme può essere, ad esempio, un insieme di valori numerici oppure un linguaggio. Un grafo $G = (V, E)$ è detto **completo** se $E = V \times V$, ovvero se ogni nodo è collegato a ogni altro nodo. Dato un grafo G e due nodi v_0 e v_k, un **cammino** da v_0 a v_k di lunghezza k è una sequenza di archi (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) tale che, per ogni i e j con $0 \leqslant i < j \leqslant k$, $v_i \neq v_j$ se $i \neq 0$ o $j \neq k$: se $v_0 = v_k$ allora la sequenza di archi è detta essere un **ciclo**. Un **albero** è un grafo senza cicli.

Esempio 3.2: il grafo delle transizioni della macchina per il complemento bit a bit

Il grafo corrispondente alla macchina di Turing introdotta nell'Esempio 3.1 è mostrato nella Figura 3.2. Come si può vedere, ciascuna transizione corrisponde a un'istruzione del programma descritto nell'Esempio 3.1. Ad esempio, l'arco che congiunge lo stato `q0` allo stato `q1` corrisponde alla seconda istruzione: in effetti, nel caso in cui lo stato attuale sia `q0` e il simbolo letto sia $\square$, allora il simbolo non deve essere modificato, la testina si deve spostare a sinistra e la CPU deve passare nello stato `q1`.

3.1.1 Rappresentazione tabellare di una macchina di Turing

Oltre alla rappresentazione basata sul grafo delle transizioni a cui abbiamo fatto riferimento nella Definizione 3.1, una macchina di Turing viene spesso specificata facendo uso di una rappresentazione tabellare. In base a tale rappresentazione, a una macchina di Turing viene associata una tabella di tante righe quante sono le triple contenute nelle etichette degli archi del grafo delle transizioni corrispondente alla macchina. Se (σ, τ, m) è una tripla contenuta nell'etichetta dell'arco che collega lo stato q allo stato p, la corrispondente riga della tabella sarà formata da cinque colonne, contenenti, rispettivamente, q, σ, p, τ e m. In altre parole, ogni riga della tabella corrisponde a un'istruzione del programma della macchina di Turing: la prima e la seconda colonna della riga specificano, rispettivamente, lo stato in cui si deve trovare la CPU e il simbolo che deve essere letto dalla testina affinché l'istruzione sia eseguita, mentre la terza, la quarta e la quinta colonna specificano, rispettivamente, il nuovo stato in cui si troverà la CPU, il simbolo che sostituirà quello appena letto e il movimento della testina che sarà effettuato eseguendo l'istruzione.

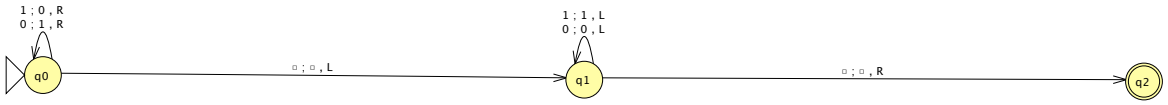
Osserviamo che, in generale, la tabella non contiene tante righe quante sono le possibili coppie formate da uno stato e un simbolo dell'alfabeto di lavoro della macchina, in quanto per alcune (generalmente molte) di queste coppie il comportamento della macchina può non essere stato specificato. Se la macchina si trova in uno stato q e la testina di lettura legge un carattere x e per la coppia (q, x) non è definita la transizione allora la macchina si arresta.

Esempio 3.3: rappresentazione tabellare della macchina per il complemento bit a bit

Facendo riferimento alla macchina di Turing introdotta nell'Esempio 3.1 e mostrata nella Figura 3.2, la corrispondente rappresentazione tabellare di tale macchina è la seguente.

stato	simbolo	stato	simbolo	movimento
q0	0	q0	1	R
q0	1	q0	0	R
q0	$\square$	q1	$\square$	L
q1	0	q1	0	L
q1	1	q1	1	L
q1	$\square$	q2	$\square$	R

Figura 3.2: macchina di Turing per il complemento bit a bit.



3.2 Automi a stati finiti

Un automa a stati finiti è un caso particolare di macchina di Turing, in cui, ad ogni passo, la testina avanza di un carattere verso destra. Formalmente abbiamo la seguente definizione.

Definizione 3.2: automa a stati finiti

Un *automa a stati finiti* è una macchina di Turing in cui, per ogni transizione, abbiamo $\mathtt{m} = \mathtt{R}$.

Un automa a stati finiti non può scrivere sul nastro[2] e, pertanto, non è in grado di effettuare calcoli. L'automa legge la stringa di input e, alla fine della lettura, si può trovare o in uno stato finale o in uno stato non finale. Diciamo che una stringa in ingresso è accettata dall'automa se e solo se, al termine della lettura, l'automa si trova in uno stato finale. In tutti gli altri casi, diciamo che la stringa non è accettata. Quindi, un automa definisce naturalmente un linguaggio dato dalle stringhe in ingresso che sono accettate.

Definizione 3.3: linguaggio accettato da un automa a stati finiti

Dato un automa a stati finiti A, l'insieme delle stringhe accettate da A definisce il linguaggio $L(A)$ accettato da A.

Il numero di transizioni di un automa a stati finiti è pari al massimo al numero dei caratteri in ingresso. Infatti, nel caso che la sequenza non sia accettata prima che tutto l'input sia stato esaminato, allora il numero di transizioni è inferiore. Questo caso si verifica, ad esempio, quando l'input contiene un simbolo che non appartiene all'alfabeto di lavoro Σ dell'automa.

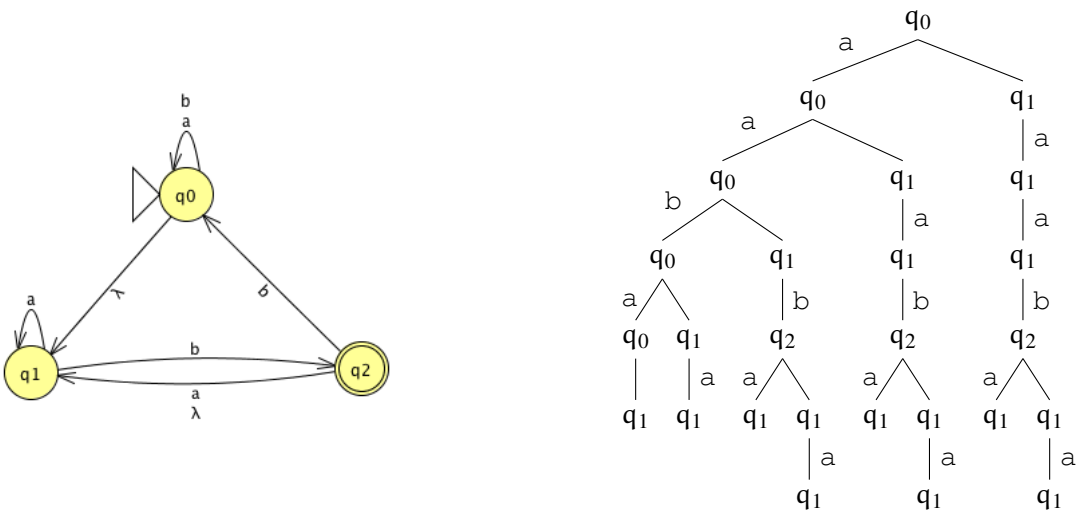
3.2.1 Automi a stati finiti non deterministici

Un automa a stati finiti non deterministico è un automa a stati finiti 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. In particolare, in un automa a stati finiti non deterministico, può esistere almeno uno stato q ed un valore di ingresso x per cui da q con ingresso x si giunge in due stati diversi. In altre parole, il grafo delle transizioni di automa a stati finiti non deterministico può includere due o più archi distinti uscenti dallo stato q le cui etichette includano ognuna almeno una tripla con lo stesso primo simbolo.

Chiaramente, la computazione di un automa a stati finiti A per un dato ingresso non è più rappresentabile mediante una sequenza di configurazioni degli stati di A. Tuttavia, essa può essere vista come un albero, detto *albero delle computazioni*, i cui nodi corrispondono alle configurazioni di A e i cui archi corrispondono alla produzione di una configurazione da parte di un'altra configurazione. Ciascuno dei cammini dell'albero che parte dalla sua radice (ovvero, dalla configurazione iniziale) è detto essere un **cammino di computazione**.

[2] Più precisamente, un automa a stati finiti può scrivere su una cella ma non può, poi, leggerne di nuovo il contenuto. Quindi, in pratica è come se un automa a stati finiti non potesse scrivere.

Figura 3.3: un automa a stati finiti non deterministico e un suo albero delle computazioni.



Un ingresso x è **accettato** da un automa a stati finiti non deterministica A 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.

Nel caso degli automi a stati finiti il non determinismo è generalmente esteso includendo la possibilità di λ-**transizioni**, ovvero transizioni che avvengono senza leggere alcun simbolo di input.

Esempio 3.4: un automa a stati finiti con transizioni nulle

Nella parte sinistra della Figura 3.3 è mostrato un automa non deterministico con λ-transizioni. Infatti nello stato q_0 è possibile passare direttamente nello stato q_1 senza leggere un carattere in ingresso. Pertanto se l'automa si trova nello stato q_0 e il carattere in ingresso è a ci sono due possibilità: nella prima l'automa legge a spostando la testina sul prossimo carattere in ingresso e rimane in q_0. Nella seconda può passare nello stato q_1 senza leggere il carattere a. L'albero delle computazioni corrispondente alla stringa `aaba` è mostrato nella parte destra della figura: come si può vedere, la stringa non viene accettata.

La possibilità per gli automi non deterministici di riuscire a indovinare (se esiste) la transizione giusta che porta in uno stato finale, potrebbe far 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, ovvero tale che $L(T) = L(T')$.

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 3.5: 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 3.4).

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 3.4).

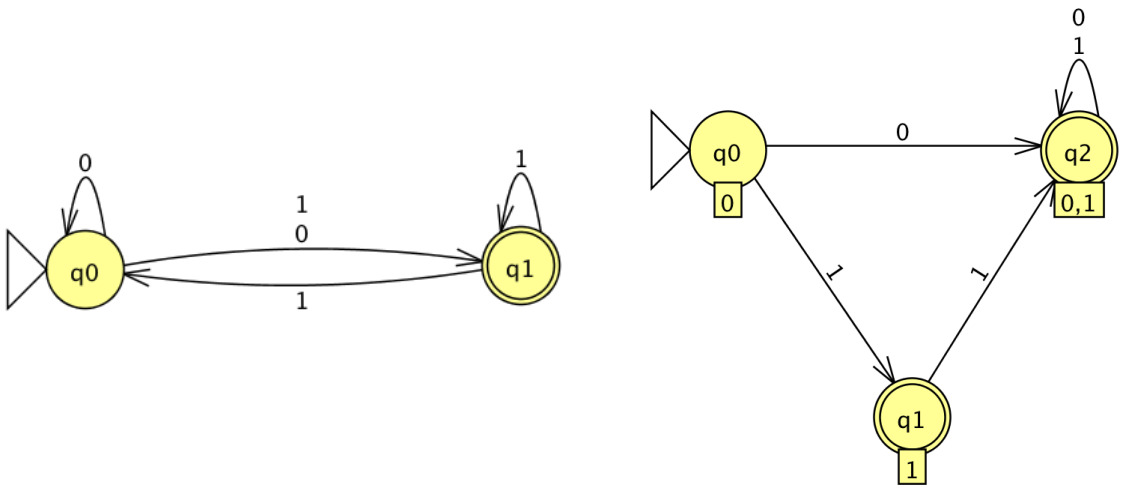
stato	simbolo	stato
Q0	0	Q1
Q0	1	Q2
Q1	0	Q1
Q1	1	Q1
Q2	1	Q1

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 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 $\bigcup_{i=1}^{k} N(s_i, \sigma)$.

3.2.2 Espressioni regolari ed automi a stati finiti

Nel capitolo 2 abbiamo considerato le espressioni regolari che utilizzano un approccio algebrico per definire linguaggi: una specifica espressione regolare definisce un linguaggio. Analogamente un automa

Figura 3.4: trasformazione di un automa a stati finiti non deterministico in uno deterministico.



a stati finiti definisce il linguaggio delle stringhe che sono accettate dall'automa utlizzando un approccio di tipo algoritmico. I due diversi approcci permettono di definire la stessa classe di linguaggi. Infatti è possibile mostrare che

1. per ogni espressione regolare R esiste un automa a stati finiti che accetta tutte e sole le stringhe che appartengono al linguaggio $L(R)$ generato da R (si veda il prossimo teorema);

2. per ogni automa a stati finiti A esiste un'espressione regolare che definisce il linguaggio $L(A)$ riconosciuto da A (la dimostrazione di questo punto esula dagli scopi di queste dispense e invitiamo il lettore interessato a consultare uno dei libri di testo a cui facciamo riferimento alla fine della prefazione).

Teorema 3.2

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

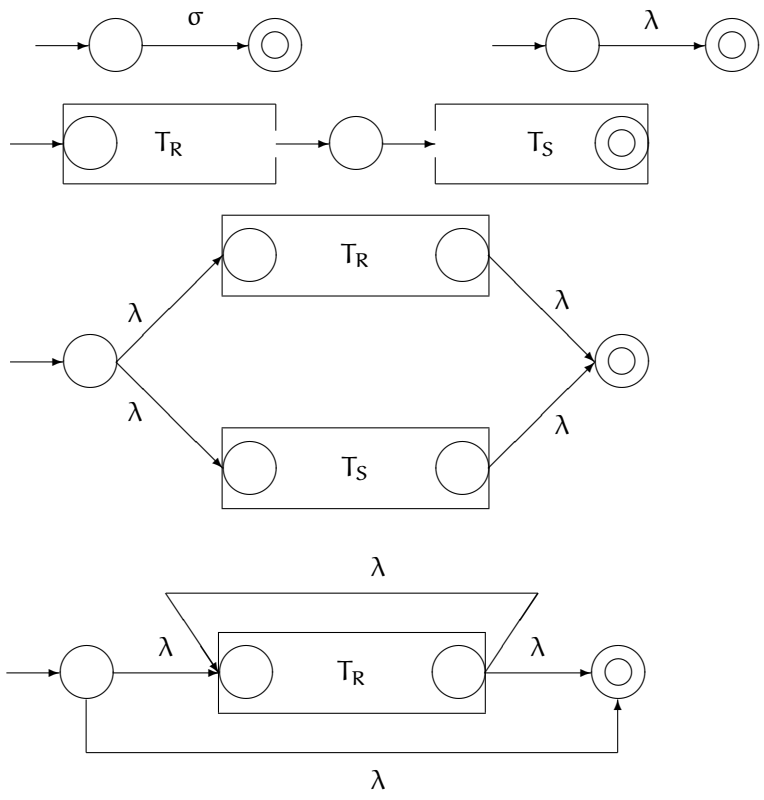
Dimostrazione. La costruzione si basa sulla definizione induttiva delle espressioni regolari fornendo una macchina oppure un'interconnessione di macchine corrispondente a ogni passo della definizione. La costruzione, che tra l'altro assicura che l'automa ottenuto avrà un solo stato finale diverso dallo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

- 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 accettano, 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 accettano, 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.

Figura 3.5: da espressioni regolari ad automi non deterministici



- 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 accetta $L(R)$, 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.

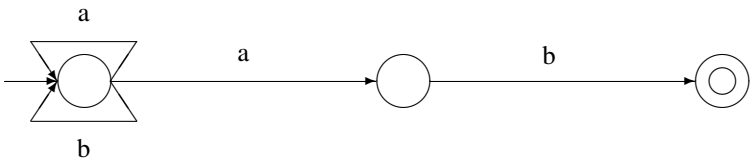
È facile verificare che la costruzione sopra descritta permette di definire un automa a stati finiti non deterministico equivalente a una specifica espressione regolare. ◇

Dall'automa a stati finiti non deterministico ottenuto dalla costruzione precedente possiamo ottenere un automa a stati finiti equivalente all'espressione regolare di partenza, applicando il Teorema 3.1.

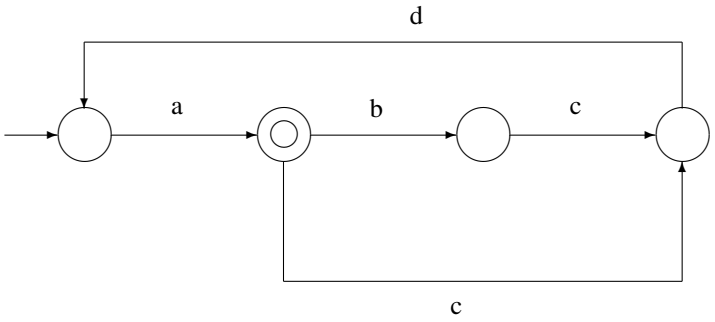
Esercizi

Esercizio 3.1. Trasformare l'automa a stati finiti non deterministico della Figura 3.3 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. 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 3.4. Definire un'espressione regolare che generi il linguaggio accettato dal seguente automa a stati finiti.



Capitolo 4

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.

4.1 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. Ad esempio, se consideriamo la frase *il cane morde la mucca pigra* sappiamo che essa 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'unità (di proporzioni variabili) della struttura sintattica di un enunciato. Nel contesto di una frase, si dicono sintagmi dei costituenti strutturali, composti da elementi appartenenti a diverse categorie lessicali (o parti del discorso).

Ad esempio, nel seguito 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 più 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 dette **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` $\rightarrow$ `Sintagma-nominale Sintagma-verbale`
- `Sintagma-nominale` $\rightarrow$ `Determinante Nome Aggettivo`
- `Sintagma-verbale` $\rightarrow$ `Verbo Sintagma-nominale`
- `Determinante` $\rightarrow$ `il | un | la | una` | ϵ
- `Nome` $\rightarrow$ `cane | mucca`
- `Aggettivo` $\rightarrow$ `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 4.1: 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 4.1: una 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 $x = a^{n+1} b^{n+1}$.

Nel secondo esempio consideriamo una grammatica per le date.

Esempio 4.2: una grammatica generativa per le date -1

Consideriamo la grammatica $G = (V, T, S, P)$, dove $V = \{S, \mathrm{Cifra}\}$, $T = \{0,1,2,3,4,5,6,7,8,9,/\}$ e P contiene le seguenti produzioni:

$$S \to \mathrm{CifraCifra/CifraCifra/CifraCifraCifraCifra}$$

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

È facile verificare che il linguaggio $L(G)$ permette di generare stringhe che iniziano con due cifre seguite dal simbolo / seguite da quattro cifre e così via.

Osserviamo che la grammatica precedente permette di generare anche stringhe del tipo 00/99/2016 e 45/00/2017.

Se vogliamo generare una grammatica in cui questo non è ammesso dobbiamo considerare una grammatica più complessa in cui introduciamo altri simboli non terminali G, M, A che rappresentano rispettivamente il giorno, il mese e l'anno, CifraNZ che rappresenta una cifra diversa zero.

Esempio 4.3: una grammatica generativa per le date -2

$$S \to G/M/A$$

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

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

Per definire il giorno usiamo le seguenti produzioni

$$G \to 0CifraNZ$$

$$G \to 1Cifra$$

$$G \to 2Cifra$$

$$G \to 30$$

$$G \to 31$$

La prima produzione elimina la possibilità che il giorno sia 00; la seconda e la produzione permettono di generare i giorni compresi fra 10 e 29, mentre le ulitme du produzioni permettono di definire i giorni 30 e 31.
Per definire il mese usiamo le seguenti produzioni

$$M \to 0CifraNZ$$

$$G \to 11$$

$$G \to 12$$

È facile verificare che in questo modo possiamo solo generare un numero di due cifre fra 01 e 12.
Per definire l'anno non abbamo particolari restrizioni e quindi possiamo aggiungere la seguente produzione

$$A \to CifraCifraCifraCifra$$

Pertanto raccogliendo tutte le produzioni otteniamo una grammatica che genera 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 $GG/MM/AAAA$ che abbiamo considerato all'inizio del capitolo 2.

4.2 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.

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

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

È facile verificare che la grammatica dell'Esempio **??**) è una grammatica regolare mentre quello dell'l'Esempio **??**) è libera dal contesto.

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 $i-1$. IÈ 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, il linguaggio dell'Esempio 4.1 è di tipo 2 ma non di tipo 3 e il linguaggio del prossimo esempio è di tipo 1 ma non di tipo 2.

Definizione 4.2: 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.

Esempio 4.4: 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:

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

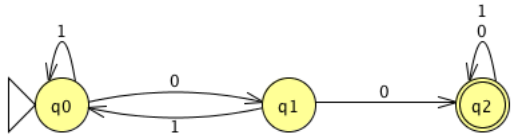
Chiaramente, mediante la prima produzione possiamo generare la stringa 012. Per generare, invece, la stringa $0^n1^n2^n$ con $n > 1$ possiamo anzitutto applicare la seconda produzione (ottenendo 0A12), la terza produzione (per spostare il simbolo A dopo il simbolo 1), la quarta produzione (ottenendo 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 $0^3 1^3 2^3$ 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.

4.3 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 4.1

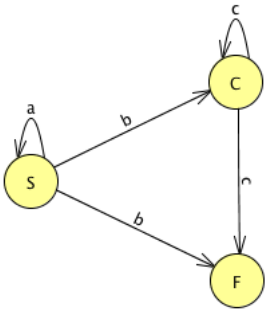
Figura 4.1: un automa a stati finiti e la corrispondente grammatica regolare.



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

Figura 4.2: una grammatica regolare e il corrispondente automa a stati finiti.

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



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

Dimostrazione. Sia A un automa a stati finiti e sia $L = L(A)$ il linguaggio deciso da A. 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 A: 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 $L(A) = L(G)$ (si veda ad esempio, la Figura 4.1).

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 A tale che $L(A) = L$: in base a quanto visto nei paragrafi precedenti, questo dimostra che esiste un automa a stati finiti equivalente a G. L'automa A 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, A avrà una transizione dallo stato corrispondente a X allo stato corrispondente a Y leggendo il simbolo a (in questo modo, possiamo avere che A sia non deterministico). Inoltre, per ogni produzione del tipo $X \to a$ di G, A avrà una transizione dallo stato corrispondente a X all'unico stato finale F leggendo il simbolo a. Di nuovo, è facile verificare che $L(A) = L(G)$ (si veda ad esempio, la Figura 4.2). ◇

Il precedente teorema mostra che per ogni espressione regolare esiste una grammatica regolare destra equivalente. In altre parole, per ogni espressione regolare si può costruire una grammatica regolare destra e viceversa, e in queste trasformazioni le stringhe accettate/generate sono sempre le stesse.

Lo stesso vale per le grammatiche regolari sinistre. Quindi possiamo affermare che un linguaggio L è regolare se e solo se

- L è riconosciuto da un automa a stati finiti deterministico
- L è riconosciuto da un automa a stati finiti non deterministico
- L è definito da una espressione regolare

- L è definito da una grammatica regolare

Si tratta quindi di quattro modi diversi per separare le stringhe in due categorie: quelle accettate/generate/riconosciute e quelle che non lo sono. Sono quattro modi diversi ma permettono di fare le stesse cose, dato che si può passare dall'uno all'altro.

4.3.1 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 regolare allora è deciso da un automa a stati finiti A, con 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 A che visita lo stesso stato più di una volta. In altre parole possiamo dire che un automa a stati finiti con n stati non è in grado di distinguere $(n+1)$ situazioni diverse.

Consideriamo il linguaggio L dell'Esempio 4.1 costituito da tutte e sole le stringhe del tipo $a^n b^n$, per $n > 0$. In altre parole, consideriamo il linguaggio di aa e b in cui le a sono tante quante le b (una sequenza di a seguita da una sequenza di b di lunghezza uguale). Chiaramente questo linguaggio è infinito.

Questo linguaggio non è un linguaggio regolare; la dimostrazione formale di questo è oltre gli scopi di questa dispensa e nel seguito discutiamo in modo informale possibili approcci. Il primo modo utilizza il fatto che un linguaggio regolare è riconosciuto da un automa a stati finiti; supponiamo che tale automa esista e sia n il numero dei suoi stati. Si noti che un automa con n stati è in grado di distinguere solo n diverse possibilità; in altre parole un automa con n stati non è in grado di contare fino a $n+1$. Questo vuol dire che se mettiamo in ingresso all'automa $(n+1)$ stringhe di sole a, a, aa, aaa,... in cui l'ultima e composta da $(n+1)$ a; poiché l'autoam ha solo n s stati non è in grado di riconoscere queste $n+1$) diverse situazioni. Quindi, necessariamente alla fine di due di queste stringhe S_1 e S_2 di sole a l'automa si deve trovare nello stesso stato; supponiamo che S_1 sia composta da n_1 caratteri a e S_2 da n_2 caratteri a. Sia S_3 una sequenza di n_1 caratteri b. Poiché dopo aver letto S_1 e S_2 l'automa si trova nello stesso stato si deve trovare nello stesso stato con input dato dalla concatenazione di S_1S_3 e di S_{2S3}.

Quindi l'automa non ' in gradi di decidere il linguaggio.

Per meglio convincerci vediamo ora come particolari approcci per definire una grammatica regolare non funzionano. In particolare potremmo considerare la seguente grammatica regolare

$X \to a$
$X \to aX$
$X \to bY$
$Y \to b$
$Y \to bY$

Questa grammatica regolare destra permette di generare aaaabbbb ma anche aabbbb, in cui le b sono più delle a. Se si vogliono invece solo le stringhe in cui le a sono tante quante le b, questa grammatica non va bene.

Per quanto discusso in precedenza, non esiste nemmeno un automa a stati finiti (deterministico o no) o una espressione regolare che accetti solo le stringhe che sono fatte di una sequenza di a seguita da una sequenza della stessa lunghezza di b. L'espressione regolare a^*b^* non funziona anche se impone una sequenza di a seguita da una sequenza di b perché le due sequenze possono essere di lunghezza diversa.

Mentre le stringhe dell'Esempio 4.1 non sono molto comuni, esiste una categoria diffusissima che è quella dei delimitatori. Per esempio, in una espressione numerica le parentesi devono essere simmetriche, e questo si può realizzare solo con regole del tipo:

$X \to (Y)$

Questa regola implica che le parentesi siano sempre e solo introdotte insieme: quando si applica la regola, viene creata sia '(' che ')'. In questo modo, le parentesi non possono mai essere asimmetriche.

Si noti che con le grammatiche regolari, potremmo scrivere $X \to (Y$, che è regolare destra, ma poi $Y \to Z)$ sarebbe regolare sinistra, mentre nelle grammatiche regolari ci sono o solo regole destre o solo regole sinistre.

Questo tipo di stringhe compaiono di frequente in molti tipi di file, come i file html, xml e i suoi derivati (docx, xlsx, odt, svg, ecc.) e la maggior parte dei linguaggi di programmazione. Per esempio, in html una tabella viene rappresentata in questo modo:

`<table> celle-della-tabella </table>`

All'interno di una tabella è possibile inserire un'altra tabella. Questo genera una struttura del tipo:

`<table> celle <table> celle-sottotabella </table> celle </table>`

Quasi tutte le pagine web attualmente esistenti usa questo genere di struttura a tabelle l'una dentro l'altra per disporre le loro parti nelle posizioni volute, anche se le tabelle non sono in genere riconoscibili come tali.

Quando una tabella si trova all'interno di un'altra si dice nidificata. In informatica, il termine nidificato indica che qualcosa va all'interno di un'altra cosa, che a sua volta potrebbe contenerne un'altra ancora e che questo si può teoricamente portare avanti all'infinito. Per esempio, in Python si può mettere un ciclo dentro un altro ciclo, e allora si dice che sono nidificati; il tutto potrebbe stare in un altro ciclo ancora, ecc.

La nidificazione si ottiene in modo molto semplice con le grammatiche non regolari. Nel caso delle tabelle html, si tratta di regole del tipo:

`testo` $\to$ `<table> contenutotabella </table>`

4.4 La forma di Backus e Naur (BNF)

I manuali che descrivono i linguaggi di programmazione di solito usano la notazione nota come BNF (Backus-Naur Form) per descrivere la sintassi di un linguaggio.

La forma di Backus e Naur è un modo di tipo generativo per definire linguaggi ed è molto diffusa per specificare la sintassi dei linguaggi di programmazione.

Nel seguito tra parentesi angolari "<′′ e ">" indichiamo concetti che non rappresentano parti del linguaggio ma sono utilizzati per la definizione. I concetti tra parentesi angolari sono i simboli non terminali.

Vediamo ad esempio come sia possibile definire l'identificatore di una variabile in Python. Sappiamo che i nomi delle variabili Python possono essere generati da lettere e numeri ('A' alla 'Z', 'a' a 'z' e '0' a '9') e il simbolo underscore (_). Un nome di variabile valido inizia con una lettera e può quindi seguire con qualsiasi numero di simboli dell'alfabeto. Così, ad esempio, `io_non_sono_una_variabile` è un nome di variabile valido, mentre non lo è `10_valore`.

```
<lettera>::= a | b | ...  z | A | B | ...  | Z
<cifra> ::= 0 | 1 | .  .  .| 9
<carattere finale> ::= <lettera> | <cifra>
<trattino >::= _
<carattere> ::= <lettera> | <cifra> | <trattino>
<variabile> ::= <lettera>
<variabile> ::= <lettera> <carattere>* <carattere finale>
```

Le regole sono da considerare come definizioni. In particolare il simbolo '`::=`' è il simbolo definizione, il simbolo '|' viene letto come 'oppure', il simbolo '+' denota la ripetizione da 1 ad un numero qualunque di volte del simbolo precedente. Infine simboli adiacenti denotano concatenazione.

Così, per esempio, la prima definizione indica che una `<lettera>` è un carattere alfabetico minuscolo o maiuscolo, la seconda che una cifra è un un numero fra 0 e 9; la terza definizione afferma che un carattere finale è una lettera o una cifra. Infine l'ultima definizione precisa che una stringa è una variabile

valida se è una lettera oppure se è una stringa di caratteri che inizia con una lettera seguita da una sequenza qualunque di di lettere cifre e trattino e finisce con una cifra o una lettera. I nomi tra parentesi angolari non appaiono in stringhe valide, ma possono essere utilizzati in derivazione di tali stringhe applicando una delle regole di riscrittura e sono i simboli non terminali.

Esercizi

Esercizio 4.1. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti un numero pari di simboli `1`.
Esercizio 4.2. 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 4.3. 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 4.4. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno tre simboli `1`.
Esercizio 4.5. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie del tipo $0^n 1 2^n$ (n 0 seguiti da un 1 seguito da n 2).
Esercizio 4.6. 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`.

Capitolo 5

I compilatori e l'analisi sintattica

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 definiamo in questo capitolo mostrando come possano essere applicate al caso del linguaggio Python.

SOMMARIO

In questo capitolo introduciamo i compilatori e vedremo sommariamente i diversi passi atraverso cui si realizza la traduzione in linguaggio macchina di un programma scirtto in un linguaggio ad alto livello. Successivamente, concentreremo la nostra attenzione sull'analisi sintattica vedendo le principali problematiche di realizzazione dei 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.*

5.1 I compilatori

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 Python, 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*.

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.

- 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).

5.2 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 Python

ALBERTO Tarsformare in Python

```
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.

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. Questa informazione è necessaria per realizzare le fasi successive del processo di traduzione in cui abbiamo necessità di distinguere ciascuna variabile.

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 fornisce le informazioni utilizzate per la generazione della traduzione in linguaggio macchina.

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.

5.3 L'analisi sintattica e gli alberi di derivazione

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

$$\mathtt{E} \to \mathtt{E}+\mathtt{E} \quad \mathtt{E} \to \mathtt{E}-\mathtt{E} \quad \mathtt{E} \to \mathtt{E}*\mathtt{E}$$
$$\mathtt{E} \to \mathtt{E}/\mathtt{E} \quad \mathtt{E} \to (\mathtt{E}) \quad \mathtt{E} \to \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9}$$

In questa grammatica E è il simbolo non terminale che rappresenta una espressione aritmetica e **id** rappresenta l'identificatore di una variabile.

Facendo uso di tale grammatica, analizziamo l'espressione $(4+2)/(4-2)$. L'anaizzatore sintattico deve individuare le produzioni che , a partire dal simbolo iniziale E permettono di ottenere $(4+2)/(4-2)$.

Quest'espressione è una frazione il cui numeratore è $(\mathbf{4}+\mathbf{2})$ e il cui denominatore è $(\mathbf{4}-\mathbf{2})$: pertanto, la prima produzione che usiamo è $\mathtt{E} \to \mathtt{E}/\mathtt{E}$ come mostrato nella parte sinistra della Figura 5.1. Osserviamo che la produzione è rappresentata mediante un albero in cui la parte a sinistra della produzione è

Figura 5.1: lo sviluppo di un albero di derivazione.

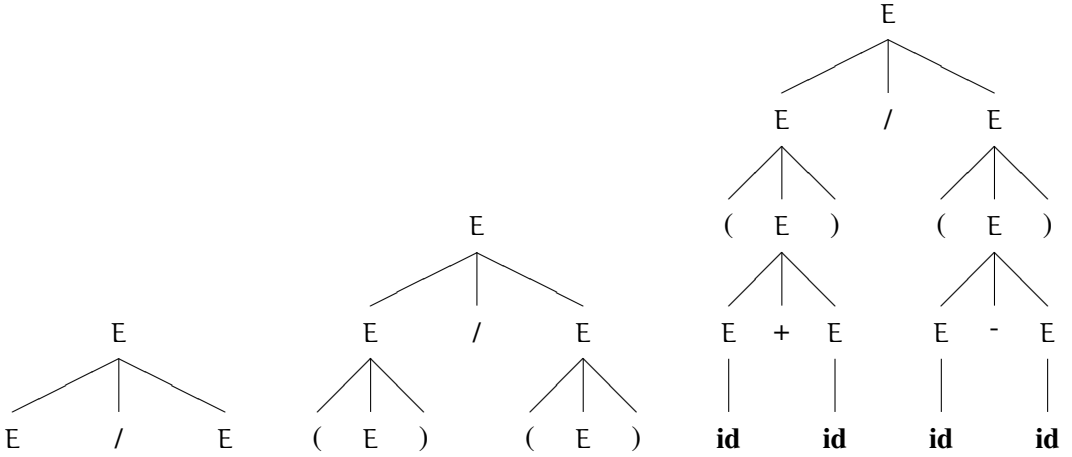
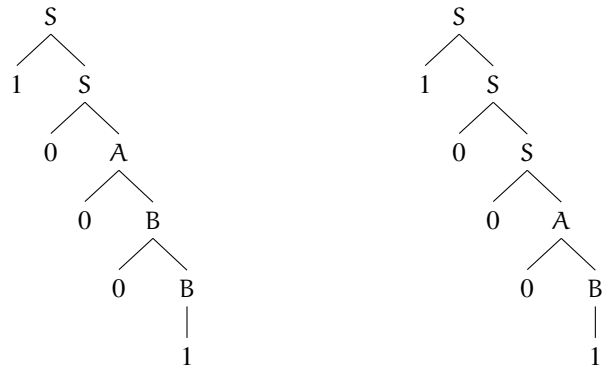


Figura 5.2: alberi di derivazione.



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 $E \to (E)$ come mostrato nella parte centrale della Figura 5.1. Il contenuto della parentesi al numeratore è una somma: quindi, sostituiamo la E all'interno della parentesi facendo uso della produzione $E \to E + E$. Il denominatore invece è una differenza: quindi, sostituiamo la E all'interno della parentesi con la produzione $E \to E - E$. A questo punto, le E rimanenti devono produrre i token identificatori terminali: quindi, facciamo uso della regola $E \to$ **id** come mostrato nella parte destra della Figura 5.1.

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.

E' evidente che l'individuazione dell'albero di derivazione fornisce le informazioni necessarie per eseguire tutte le operazioni che permettono di calcolare il valore dell'espressione data. Infatti esaminando le foglie dell'albero dobbiamo prima eseguire la somma $4+2$ e la sottrazione $4-2$; quindi dobbiamo eseguire la divisione fra i risultati ottenuti.

Esempio 5.1: 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 5.2, mentre un altro esempio di albero di derivazione della stessa stringa è mostrato nella parte destra della figura.

5.4 Grammatiche ambigue

Nei linguaggi naturali coeme l'italiano, l'ambiguità sintattica è una proprietÃ di quelle frasi per le quali può essere fornita più di un'interpretazione. Al contrario dell'ambiguità semantica, che nasce dalla gamma di significati diversi che possiamo associare ad una parola, l'ambiguità sintattica deriva dalla relazione tra le parole e le componenti sintattiche di una frase. Quando sono possibili più modi di interpretare la stessa frase con strutture sintattiche diverse, il testo è ambiguo.

Consideriamo i seguenti esempi di frasi ambigue che possono avere due interpretazioni diverse:

- Rapina in banca con rivoltella da centomila euro (una rapina effettuata con una rivoltella costosa, o una rapina con refuriva di centomila euro?)
- Luigi ha visto Ada nel parco con il cannocchiale (chi sta usando il cannocchiale? Luigi o Ada?)
- Una vecchia legge la regola (il soggetto è una donna anziana - una vecchia - oppure il soggetto è una vecchia legge?)
- Si vendono letti di ferro per bambini con palle d'ottone (senza commento).

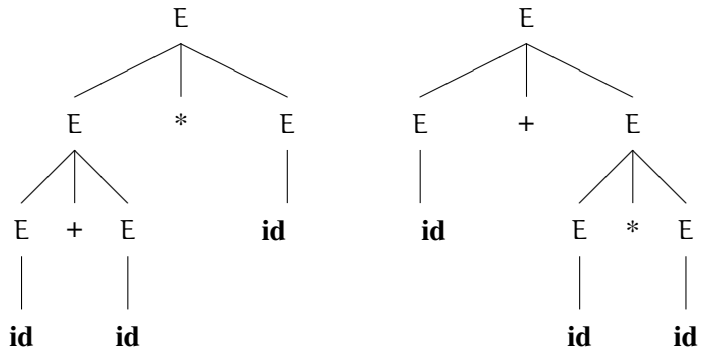
Nella maggioranza dei casi pratici il significato di una frase sintatticamente ambigua si risolve utilizzando il contesto che chiarisce il vero significato della frase. Ovviamente questo non sempre succede; ad esempio motli ritengono alcuni discorsi di uomni politici volutamente ambigui.

Purtroppo però nel caso dei linguaggi di programmazione il problema dell'ambiguità non può essere risolto dal contesto. Infatti se un programma Python deve essere tradotto in un programma eseguibile non possiamo ammettere che il nostro programma possa essere intrpretato in due modi diversi. In ltre parole *nei linguaggi di programmazione è necessario che per ogni programma sia possibile una e una sola interpretazione e che quindi la grammatica utilizzata per definire il linguaggio non permetta ambiguità*.

Analizziamo ora con maggior dettaglio l'esempio precedente. Esso 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 5.2. 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

Figura 5.3: due alberi di derivazione in una grammatica ambigua.



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 5.2 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 interpretazione 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:

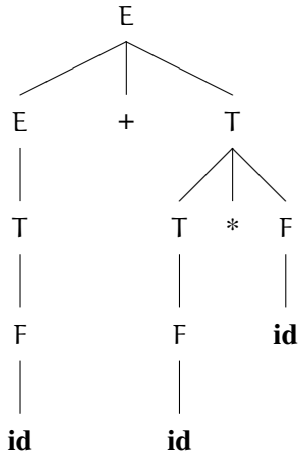
$$\begin{array}{lll} E \to E+T & E \to E-T & E \to T \\ T \to T*F & T \to T/F & T \to F \\ F \to (E) & F \to \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9} & \end{array}$$

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 5.4 (considereremo di nuovo tale grammatica nel seguito).

5.4.1 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:

Figura 5.4: albero di derivazione in una grammatica non ambigua.

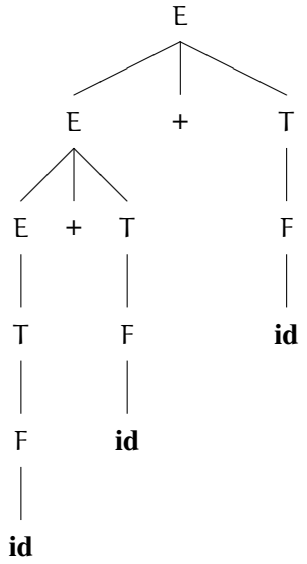


$$
\begin{aligned}
E &\rightarrow E/E \\
&\rightarrow E/(E) \\
&\rightarrow E/(E-E) \\
&\rightarrow E/(E-\mathbf{id}) \\
&\rightarrow E/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (E)/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (E+E)/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (E+\mathbf{id})/(\mathbf{id}-\mathbf{id})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

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 $E/(E-E)$ avevamo tre scelte di 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:

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

Figura 5.5: l'albero di derivazione di **id** + **id** + **id**.



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.

5.5 Analisi top-down e i parser LL

L'analisi top-down Ã¨ una strategia di analisi sintattica in cui si cerca la produzione al più alto livello della struttura di analisi e si genera l'albero di derivazione a partire dalla radice utilizzando le regole di riscrittura di una grammatica formale.

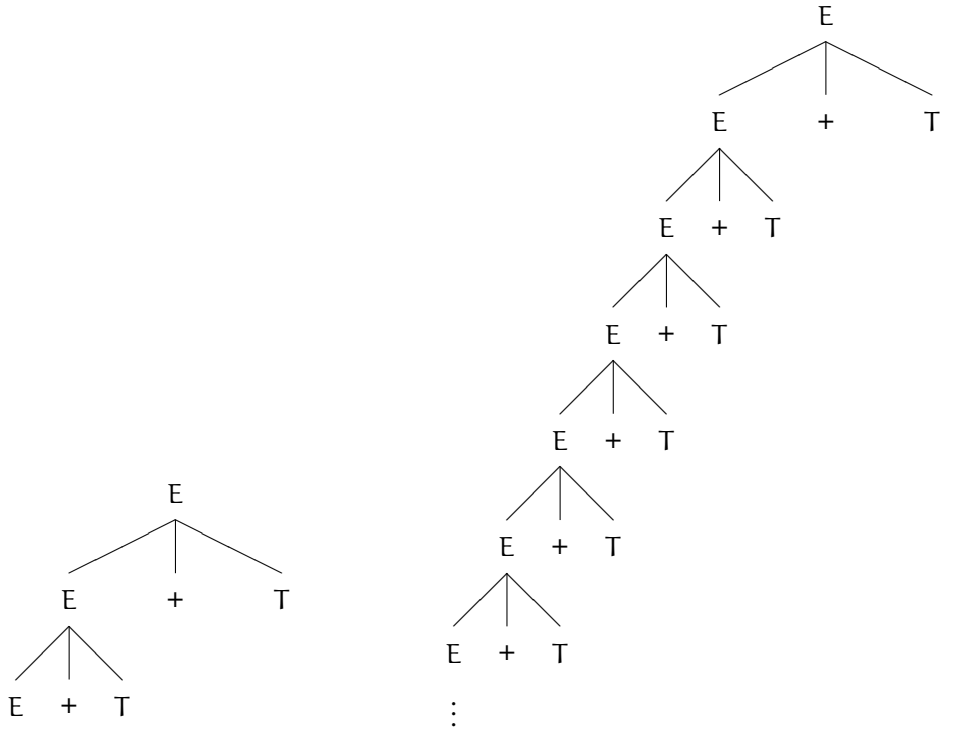
In particolare, un analizzatore **top-down** parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero che porta alla data sequenza di simboli terminali: nel fare ciò, ricostruisce una derivazione sinistra. Il parser top-down deve iniziare dalla radice dell'albero e 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.

Questo approccio richiede che la grammatica verifichi delle condizioni. Una trattazione dettagliata va oltre gli scopi di questa dispensa; nel seguito ci limitiamo ad osservare con un esempio che illustra come, per utilizzare questo metodo, le produzioni della grammatic devono essere in una particolare forma.

Ricorsione sinistra

La scansione della sequenza di simboli terminali da sinistra a destra ci crea subito dei problemi. Consideriamo la seguente grammatica (una versione semplificata della grammatica precedentemente introdotta

Figura 5.6: il problema della ricorsione sinistra.



per risolvere il problema delle ambiguità).

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

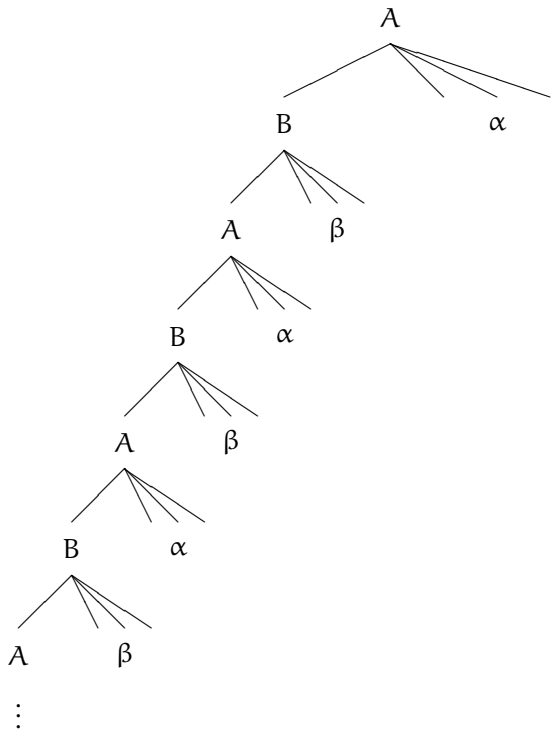
Supponiamo di stare analizzando l'espressione $\mathbf{id}+\mathbf{id}+\mathbf{id}$, il cui albero di derivazione è mostrato nella Figura 5.5. Un parser top-down per questa grammatica partirà tentando di espandere E con la produzione $E \to E+T$. Quindi tenterà di espandere il nuovo E nello stesso modo: questo ci darà l'albero mostrato nella parte sinistra della Figura 5.6 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

Figura 5.7: ricorsione sinistra non immediata.



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.

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 5.7. 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.

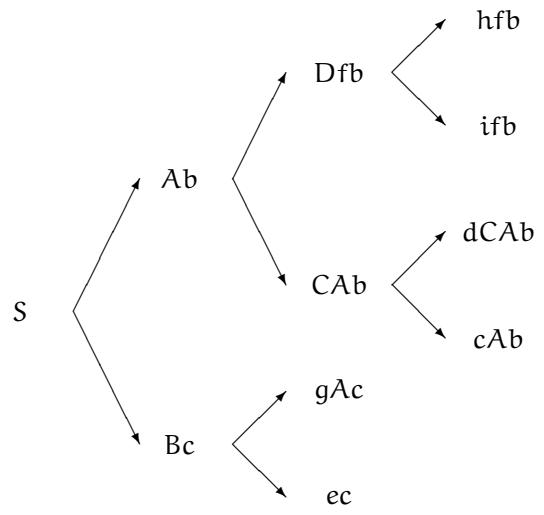
5.5.1 Parser predittivi e grammatiche LL(1)

Un parser predittivo è un parser top-down che è in grado di individuare la produzione che deve essere applicata senza errore.

Ad esempio, 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:

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

Figura 5.8: terminali derivabili dal simbolo iniziale.



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

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 molti tentativi 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.

Per poter decidere quale produzione applicare l'analizzatore sintattico deve avere 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 5.8. 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 $\mathrm{FIRST}(Ab)$ (che è uguale a $\{c, d, h, i\}$) e $\mathrm{FIRST}(Bc)$ (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 $\mathrm{FIRST}(Ab)$ e la produzione $S \to Bc$ se appartiene a $\mathrm{FIRST}(Bc)$. Se tale simbolo terminale non appartiene a $\mathrm{FIRST}(S) = \mathrm{FIRST}(Ab) \cup \mathrm{FIRST}(Bc)$, allora la sequenza è grammaticalmente scorretta e può essere rifiutata.

I parser che usano gli insiemi FIRST sono 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 $A \to \lambda$.

Fortunatamente la soluzione utilizzata nella grande nmaggioranza dei linguaggi di progrmmazione non è molto più complessa. Infatti le grammatiche dei linguaggi di programmazione (come ad esempio Python) permette di decidere senza ambiguità la produzione da applicare esaminando oltre al pirmo simbolo (quesllo in FIRST il simbolo successivo. Non entriamo nei dettagli e ci limitiamo ad osservare che la classe di grammatiche LL(1), sono le grammatiche libere dal contesto per le quali è possibile definire un parse predittivo esaminando solo il token in FIRST e il successivo token di ingresso. Osserviamo, infine, che le grammatiche escludono completamente ogni possibile ambiguità

Esercizi

Esercizio 5.1. 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 5.2. 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 5.3. Ogni linguaggio definito da una grammatica regolare può essere generato da una grammatica LL(1). Spiegare perché.
Esercizio 5.4. Scrivere la grammatica che genera le stringhe a e bbâb (sequenza di b di lunghezza arbitraria, anche zero).
Esercizio 5.5. Convertire la seguente grammatica non contestuale in una regolare. $S \to aaX$
$X \to bX$
$X \to bc$
Esercizio 5.6. La seguente frase è errata; spiegare il perché.
Tutte le grammatiche non contestuali si possono convertire in grammatiche regolari.
Esercizio 5.7. Dire se le seguente grammatica è ambigua. Spiegare il perché.
$S \to aA$
$S \to Aa$
$A \to \epsilon$