# Laboratorio di Linguaggi Formali e Traduttori
# Corso di Studi in Informatica
# A.A. 2014/2015

Luca Padovani e Jeremy Sproston
Dipartimento di Informatica — Università degli Studi di Torino

Versione del 4 dicembre 2014

**Sommario**

Questo documento descrive le esercitazioni di laboratorio e le modalità d'esame del corso di *Linguaggi Formali e Traduttori* per l'A.A. 2014/2015.

## Svolgimento e valutazione del progetto di laboratorio

È consigliato sostenere l'esame nella prima sessione d'esame dopo il corso.

### Forum di discussione e supporto on-line al corso

Sulla piattaforma I-learn sono disponibili forum di discussione dedicati per gli argomenti affrontati durante il corso e per scambiare opinioni tra i vari gruppi di lavoro e con il docente.

L'iscrizione al forum principale è effettuata automaticamente, è possibile disiscriversi ma è consigliabile farlo solo a seguito del superamento dell'esame per poter sempre ricevere in modo tempestivo le comunicazioni effettuate dal docente. Per ognuno dei forum disponibili è disponibile un canale *RSS* che riporta gli ultimi dieci post effettuati.

### Progetto di laboratorio

Il progetto di laboratorio consiste in una serie di esercitazioni assistite mirate allo sviluppo di un semplice traduttore. Il corretto svolgimento di tali esercitazioni presuppone una buona conoscenza del linguaggio di programmazione Java e degli argomenti di teoria del corso Linguaggi Formali e Traduttori.

### Modalità dell'esame di laboratorio

Per sostenere l'esame a un appello è necessario prenotarsi. L'esame di laboratorio è **orale** e **individuale**, anche se il codice è stato sviluppato in collaborazione con altri studenti. L'esame ha una durata massima di **20 minuti** durante i quali vengono accertati: il corretto svolgimento della prova di laboratorio; la comprensione della sua struttura e del suo funzionamento; la comprensione delle parti di teoria correlata al laboratorio stesso.

#### Note importanti

- Per poter discutere il laboratorio è *necessario* aver prima superato la prova scritta relativa al modulo di teoria.

- La presentazione di codice "funzionante" non è condizione necessaria né sufficiente per il superamento della prova di laboratorio. In altri termini, è possibile essere promossi presentando codice non funzionante (se relativamente completo e se lo studente dimostra di aver acquisito le conoscenze richieste) così come è possibile essere respinti presentando codice funzionante (se lo studente dimostra di avere scarsa o nulla familiarità con il codice e i concetti correlati).
- Anche se il codice è stato sviluppato in collaborazione con altri studenti, i punteggi ottenuti dai singoli studenti sono indipendenti. Per esempio, a parità di codice presentato, è possibile che uno studente meriti 30, un altro 25 e un altro ancora sia respinto.
- Dal momento che l'esame orale ha una durata massima stabilita e che durante la prova è possibile che venga richiesto di apportare modifiche al codice del progetto, è opportuno presentarsi all'esame con una perfetta conoscenza del progetto e degli argomenti di teoria correlati.

### Calcolo del voto finale

I voti della prova scritta e della prova di laboratorio sono espressi in trentesimi. Il voto finale è determinato calcolando la media pesata del voto della prova scritta e del laboratorio, secondo il loro contributo in CFU, e cioè

$$\text{voto finale} = \frac{\text{voto dello scritto} \times 2 + \text{voto del laboratorio}}{3}$$

La lode è attribuita agli studenti con voto finale pari a 30 e che abbiano dimostrato particolare brillantezza nello svolgimento delle esercitazioni di laboratorio.

### Validità del presente testo di laboratorio

Il presente testo di laboratorio è valido sino alla sessione di settembre 2015.

# 1 Implementazione di un DFA in Java

Lo scopo di questo esercizio è l'implementazione di un metodo Java che sia in grado di discriminare le stringhe del linguaggio riconosciuto da un automa a stati finiti deterministico (DFA) dato. Il primo automa che prendiamo in considerazione, mostrato in Figura 1, è definito sull'alfabeto $\{0, 1\}$ e riconosce le stringhe in cui compaiono almeno 3 zeri consecutivi.

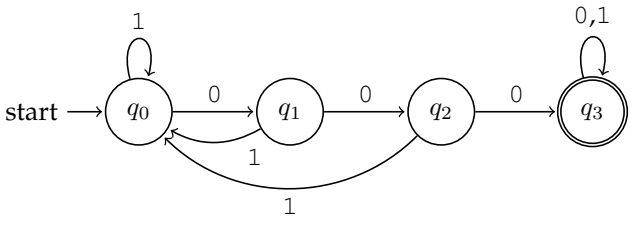



Figura 1: DFA che riconosce stringhe con 3 zeri consecutivi.

L'implementazione Java del DFA di Figura 1 è mostrata in Figura 2. L'automa è implementato nel metodo `scan` che accetta una stringa `s` e restituisce un valore booleano che indica se la stringa appartiene o meno al linguaggio riconosciuto dall'automa. Lo stato dell'automa è rappresentato per mezzo di una variabile intera `state`, mentre la variabile `i` contiene l'indice del prossimo carattere della stringa `s` da analizzare. Il corpo principale del metodo è un ciclo che,

```java
public class TreZeri
{
    public static boolean scan(String s)
    {
        int state = 0;
        int i = 0;

        while (state >= 0 && i < s.length()) {
            final char ch = s.charAt(i++);

            switch (state) {
            case 0:
                if (ch == '0')
                    state = 1;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 1:
                if (ch == '0')
                    state = 2;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 2:
                if (ch == '0')
                    state = 3;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 3:
                if (ch == '0' || ch == '1')
                    state = 3;
                else
                    state = -1;
                break;
            }
        }
        return state == 3;
    }

    public static void main(String[] args)
    {
        System.out.println(scan(args[0]) ? "OK" : "NOPE");
    }
}
```

Figura 2: Implementazione Java del DFA di Figura 1.

analizzando il contenuto della stringa `s` un carattere alla volta, effettua un cambiamento dello stato dell'automa secondo la sua funzione di transizione. Notare che l'implementazione assegna il valore $-1$ alla variabile `state` se viene incontrato un simbolo diverso da `0` e `1`. Tale valore non è uno stato valido, ma rappresenta una condizione di errore irrecuperabile.

**Esercizio 1.1.** Copiare il codice in Figura 2, compilarlo e accertarsi che funzioni correttamente testandolo su un insieme significativo di stringhe, per es. `010101`, `1100011001`, `10214`, ecc.

Come deve essere modificato il codice per riconoscere il linguaggio complementare, ovvero il linguaggio delle stringhe di `0` e `1` che **non** contengono 3 zeri consecutivi? ■

**Esercizio 1.2.** Progettare un DFA che riconosca il linguaggio delle costanti numeriche in virgola mobile. Esempi di tali costanti sono:

```
123    123.5    .567    +7.5    -.7    67e10    1e-2    -.7e2
```

Realizzare il DFA in Java seguendo la costruzione vista in Figura 2, assicurarsi che l'implementazione riconosca il linguaggio desiderato.

In base al particolare stato finale in cui si trova l'automa al termine del riconoscimento, cosa si può dire della costante numerica riconosciuta? ■

**Esercizio 1.3** (opzionale)**.** Modificare l'automa dell'esercizio precedente in modo che riconosca costanti numeriche precedute e/o seguite da sequenze eventualmente vuote di spazi. Modificare l'implementazione Java dell'automa conseguentemente.

**Esercizio 1.4** (opzionale)**.** Progettare e implementare un DFA che riconosca il linguaggio degli identificatori in un linguaggio in stile Java: un identificatore è una sequenza non vuota di lettere, numeri, ed il simbolo di sottolineatura _ che non comincia con un numero e che non può essere composto solo da un _.

**Esercizio 1.5** (opzionale)**.** Progettare e implementare un DFA che riconosca il linguaggio dei numeri binari (stringhe di `0` e `1`) il cui valore è multiplo di 3. Per esempio, `110` e `1001` sono stringhe del linguaggio (rappresentano rispettivamente i numeri 6 e 9), mentre `10` e `111` no (rappresentano rispettivamente i numeri 2 e 7). **Suggerimento:** usare tre stati per rappresentare il resto della divisione per 3 del numero.

## 2 Rappresentazione di DFA in Java

Nella Sezione 1 abbiamo visto come rappresentare uno specifico DFA in Java, definendo una classe con un metodo `scan` che contiene al proprio interno l'implementazione della funzione di transizione dell'automa. In questa sezione vedremo come rappresentare in Java DFA arbitrari. In particolare, definiremo una classe `DFA` che fornisce metodi per costruire un DFA, verificarne alcune proprietà fondamentali come il numero degli stati e l'insieme degli stati finali, e determinare se una stringa è o meno accettata dall'automa. Tale classe comprenderà anche metodi per stampare due rappresentazioni testuali dell'automa, una che può essere convertita in immagini attraverso il tool `dot` per la visualizzazione di grafi, l'altra che genera la classe Java che implementa l'automa, secondo lo schema usato nella Sezione 1.

La rappresentazione di un automa comporta la rappresentazione di tutte le sue componenti, ovvero gli stati, l'alfabeto, lo stato iniziale, l'insieme degli stati finali, e la funzione di transizione. Operiamo dunque le seguenti scelte:

- rappresentiamo gli stati del DFA con numeri interi, cioè valori di tipo **`int`**;
- per convenzione, stabiliamo che lo stato iniziale dell'automa è sempre quello con indice 0;
- come già fatto nella Sezione 1, rappresentiamo i simboli dell'alfabeto del DFA con caratteri, cioè valori di tipo **`char`**.

**Rappresentazione dell'insieme di stati finali.** Avendo scelto di rappresentare gli stati come numeri interi, scegliamo una struttura dati che consenta la rappresentazione di *insiemi* di numeri interi. La libreria standard di Java mette già a disposizione tale classe, il contenitore `HashSet<Integer>`.

Il seguente frammento di codice illustra un esempio di utilizzo della classe `HashSet<Integer>`:

```
HashSet<Integer> s = new HashSet<Integer>();
s.add(3);                       // inserisce 3 nell'insieme s
s.add(4);                       // inserisce 4 nell'insieme s
s.add(3);                       // inserisce 3 nell'insieme s
System.out.println(s.size()); // stampa 2
for (int i : s)                 // itera su tutti gli elementi di s
  System.out.println(i);        // stampa 3 e 4
```

**Rappresentazione della funzione di transizione.** Si ricorda che la funzione di transizione, comunemente indicata con $\delta$, ha tipo

$$\delta : \mathbb{Q} \times \Sigma \to \mathbb{Q}$$

ovvero è una funzione a due argomenti, uno stato di partenza $q \in \mathbb{Q}$ e un simbolo $a \in \Sigma$ dell'alfabeto dell'automa, tale che $\delta(q, a)$ è lo stato $p$ in cui l'automa transisce dopo aver riconosciuto il simbolo $a$ nello stato $q$.

Esistono diverse rappresentazioni possibili della funzione di transizione, più o meno compatte e più o meno efficienti. In questa serie di esercizi utilizzeremo una rappresentazione di $\delta$ come *tabella hash*. Questa scelta offre un buon compromesso tra compattezza ed efficienza ed ha l'ulteriore vantaggio di essere già disponibile nella libreria standard di Java.

Una tabella hash è un'istanza della classe

```
HashMap<K,V>
```

dove `K` rappresenta il tipo delle *chiavi* (*key*) della tabella e `V` rappresenta il tipo dei *valori* della tabella. In altri termini, le chiavi per accedere alla tabella sono istanze di `K` ed i valori associati alle chiavi sono istanze di `V`. La classe `HashMap<K,V>` mette a disposizione due metodi, `put` e `get`, che servono rispettivamente per:

1. inserire associazioni (chiave, valore) nella tabella;
2. recuperare il valore associato a una chiave dalla tabella.

Il seguente frammento di codice Java illustra un utilizzo della classe come "traduttore" da un (piccolo) dizionario di parole inglesi alle corrispondenti italiane:

```
HashMap<String, String> m = new HashMap<String, String>();
m.put("hello", "ciao");
m.put("world", "mondo");
System.out.println(m.get("hello")); // stampa "ciao"
System.out.println(m.get("world")); // stampa "mondo"
```

Notare che una invocazione `m.get(k)` ritorna un oggetto valido corrispondente alla chiave `k` solo se è stata precedentemente eseguita una operazione `m.put(k, v)` che associa la chiave `k` al valore `v`. Nel caso in cui si richieda, attraverso `get`, il valore di una chiave non inserita nella tabella si otterrà come risultato `null`.

Chiavi e valori di una tabella hash non devono necessariamente avere lo stesso tipo. Per esempio, il seguente codice crea una tabella hash contenente la corrispondenza tra numeri e nomi dei giorni della settimana:

```
HashMap<String, Integer> m = new HashMap<String, Integer>();
m.put("monday",    1);
m.put("tuesday",   2);
m.put("wednesday", 3);
```

```
m.put("thursday",  4);
m.put("friday",    5);
m.put("saturday",  6);
m.put("sunday",    7);
System.out.println(m.get("saturday")); // stampa 6
```

La corrispondenza inversa è ottenibile in modo analogo (si noti `HashMap<Integer, String>` al posto di `HashMap<String, Integer>`):

```
HashMap<Integer, String> m = new HashMap<Integer, String>();
m.put(1, "monday");
m.put(2, "tuesday");
m.put(3, "wednesday");
m.put(4, "thursday");
m.put(5, "friday");
m.put(6, "saturday");
m.put(7, "sunday");
System.out.println(m.get(6)); // stampa "saturday"
```

Tornando al problema della rappresentazione della funzione di transizione $\delta$ di un DFA, e ricordando che $\delta$ ha tipo $\mathbb{Q} \times \Sigma \to \mathbb{Q}$, è naturale allora utilizzare una tabella hash le cui chiavi sono coppie $\mathbb{Q} \times \Sigma$ fatte da uno stato dell'automa (lo stato di partenza di una transizione) e da un simbolo dell'alfabeto (l'etichetta della transizione) e i cui valori sono lo stato di arrivo.

Per poter rappresentare una funzione di transizione $\delta$ come una tabella hash con chiavi $\mathbb{Q} \times \Sigma$ e valori $\mathbb{Q}$ ci occorre dunque una rappresentazione delle coppie $\mathbb{Q} \times \Sigma$. A tal fine, definiamo la classe `Move` seguente:[1]

Listing 1: `Move.java`

```
/**
 * Un oggetto della classe <code>Move</code> rappresenta una mossa di
 * un automa a stati finiti, ovvero una coppia costituita da uno stato
 * di partenza e da un simbolo dell'alfabeto dell'automa.
 */
public class Move
{
    /** Lo stato di partenza. */
    final int start;
    /** Il simbolo che etichetta la transizione. */
    final char ch;

    /**
     * Crea una mossa con stato di partenza e simbolo dati.
     * @param start Lo stato di partenza.
     * @param ch Il simbolo che etichetta la transizione.
     */
    public Move(int start, char ch) {
        this.start = start;
        this.ch = ch;
    }

    /**
     * Confronta due mosse.
     * @param o La mossa da confrontare a questa.
     * @return <code>true</code> se le due mosse sono uguali, ovvero
     *         hanno lo stesso stato di partenza e lo stesso simbolo,
```

[1] Si sconsiglia di fare copia e incolla del codice dal documento, in quanto alcuni simboli potrebbero non essere copiati correttamente. Tutti i listati presentati in questo documento sono scaricabili dalla pagina del corso sulla piattaforma Moodle.

```
     *          <code>false</code> altrimenti.
     */
    public boolean equals(Object o) {
        if (o instanceof Move) {
            Move m = (Move) o;
            return (start == m.start && ch == m.ch);
        } else
            return false;
    }

    /**
     * Calcola il valore hash della mossa.
     * @return Il valore hash.
     */
    public int hashCode() {
        return start ^ (int) ch;
    }
}
```

Una mossa è una coppia $(\text{stato}, \text{simbolo})$ che viene usata per identificare una transizione dell'automa. Per esempio, il DFA in Figura 1 ha una mossa $(q_2, 0)$ che identifica una transizione da $q_2$ a $q_3$ etichettata con il simbolo `0`. La classe `Move` contiene due campi `start` e `ch` che contengono rispettivamente lo stato iniziale ed il simbolo che identificano la transizione. Per esempio, la mossa $(q_2, 0)$ menzionata sopra è rappresentata come la seguente istanza di `Move`:

```
new Move(2, '0')
```

**NOTA**

Il corretto funzionamento delle istanze della classe `Move` come chiavi di una tabella hash richiede la ridefinizione dei metodi `equals` e `hashCode`, la cui comprensione *non* è richiesta né necessaria ai fini del corso di Linguaggi Formali e Traduttori.

Il listato seguente definisce la classe `DFA`:

Listing 2: `DFA.java`

```
import java.util.HashSet;
import java.util.HashMap;

/**
 * Un oggetto della classe DFA rappresenta un automa a stati finiti
 * deterministico
 */
public class DFA
{
    /**
     * Numero degli stati dell'automa. Ogni stato e` rappresentato da
     * un numero interno non negativo, lo stato con indice 0 e` lo
     * stato iniziale.
     */
    private int numberOfStates;

    /** Insieme degli stati finali dell'automa. */
    private HashSet<Integer> finalStates;

    /**
```

```
     * Funzione di transizione dell'automa, rappresentata come una
     * mappa da mosse a stati di arrivo.
     */
    private HashMap<Move, Integer> transitions;

    /**
     * Crea un DFA con un dato numero di stati.
     * @param  n Il numero di stati dell'automa.
     */
    public DFA(int n) {
        numberOfStates = n;
        finalStates = new HashSet<Integer>();
        transitions = new HashMap<Move, Integer>();
    }

    /**
     * Aggiunge uno stato all'automa.
     * @return L'indice del nuovo stato creato
     */
    public int newState() {
        return numberOfStates++;
    }

    /**
     * Aggiunge una transizione all'automa.
     * @param  p  Lo stato di partenza della transizione.
     * @param  ch Il simbolo che etichetta la transizione.
     * @param  q  Lo stato di arrivo della transizione.
     * @return <code>true</code> se lo stato di partenza e lo stato di
     *         arrivo sono validi, <code>false</code> altrimenti.
     */
    public boolean setMove(int p, char ch, int q) {
        if (!validState(p) || !validState(q))
            return false;

        transitions.put(new Move(p, ch), q);
        return true;
    }

    /**
     * Aggiunge uno stato finale.
     * @param  p Lo stato che si vuole aggiungere a quelli finali.
     * @return <code>true</code> se lo stato e` valido,
     *         <code>false</code> altrimenti.
     */
    public boolean addFinalState(int p) {
        if (validState(p)) {
            finalStates.add(p);
            return true;
        } else
            return false;
    }

    /**
     * Determina se uno stato e` valido oppure no.
     * @param  p Lo stato da controllare.
     * @return <code>true</code> se lo stato e` valido,
     *         <code>false</code> altrimenti.
```

```
     * @see #numberOfStates
     */
    public boolean validState(int p) {
        return (p >= 0 && p < numberOfStates);
    }

    /**
     * Determina se uno stato e` finale oppure no.
     * @param  p Lo stato da controllare.
     * @return <code>true</code> se lo stato e` finale,
     *         <code>false</code> altrimenti.
     * @see #finalStates
     */
    public boolean finalState(int p) {
        return finalStates.contains(p);
    }

    /**
     * Restituisce il numero di stati dell'automa.
     * @return Numero di stati.
     */
    public int numberOfStates() {
        return numberOfStates;
    }

    /**
     * Restituisce l'alfabeto dell'automa, ovvero l'insieme di simboli
     * che compaiono come etichette delle transizioni dell'automa.
     * @return L'alfabeto dell'automa.
     */
    public HashSet<Character> alphabet() {
        HashSet<Character> alphabet = new HashSet<Character>();
        for (Move m : transitions.keySet())
            alphabet.add(m.ch);
        return alphabet;
    }

    /**
     * Esegue una mossa dell'automa.
     * @param p  Stato di partenza prima della transizione.
     * @param ch Simbolo da riconoscere.
     * @return Stato di arrivo dopo la transizione, oppure
     *         <code>-1</code> se l'automa non ha una transizione
     *         etichettata con <code>ch</code> dallo stato
     *         <code>p</code>.
     */
    public int move(int p, char ch) {
        Move move = new Move(p, ch);
        if (transitions.containsKey(move))
            return transitions.get(move);
        else
            return -1;
    }
}
```

Di seguito è mostrato un esempio di istanziazione della classe `DFA` che rappresenta l'automa della Figura 1.

```
DFA tz = new DFA(4);
```

```
tz.setMove(0, '1', 0);
tz.setMove(0, '0', 1);
tz.setMove(1, '1', 0);
tz.setMove(1, '0', 2);
tz.setMove(2, '1', 0);
tz.setMove(2, '0', 3);
tz.setMove(3, '1', 3);
tz.setMove(3, '0', 3);
tz.addFinalState(3);
```

**NOTA**

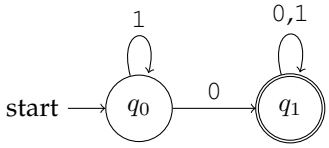
Per risolvere gli esercizi proposti nel seguito di questa sezione, non è necessario ed anzi **è vietato** modificare in qualunque modo i metodi forniti nelle classi `Move` e `DFA`.

**Esercizio 2.1.** Leggere attentamente la documentazione delle classi `HashMap` ed `HashSet`, in particolare quella riguardante i metodi di tali classi utilizzati nei listati precedenti.

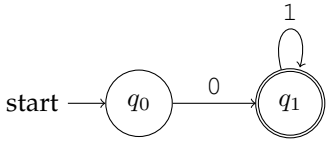
**Esercizio 2.2.** Aggiungere alla classe `DFA` un metodo `scan` che accetta una stringa $s$ e ritorna **`true`** se $s$ è riconosciuta dall'automa, **`false`** altrimenti. Scrivere un semplice programma di prova che crea una istanza della classe `DFA`, costruisce la rappresentazione del DFA mostrato in Figura 1, e verifica se una stringa data in input (passata dalla linea di comando o letta da tastiera) è o meno riconosciuta dall'automa.

**Esercizio 2.3.** Ripetere l'esercizio precedente con il DFA dell'esercizio 1.2.

**Esercizio 2.4** (opzionale). Aggiungere alla classe `DFA` un metodo `complete` che ritorna **`true`** se la funzione di transizione di un DFA è definita per tutti gli stati dell'automa e i simboli del suo alfabeto di riferimento, **`false`** altrimenti. Per esempio, l'invocazione del metodo `complete` sulla rappresentazione dell'automa




deve ritornare **`true`**, perché l'alfabeto di riferimento dell'automa è $\{0,1\}$ e ogni stato ha una transizione uscente per ogni simbolo dell'alfabeto, mentre l'invocazione del metodo `complete` sulla rappresentazione dell'automa




deve ritornare **`false`**, perché l'alfabeto di riferimento dell'automa è sempre $\{0,1\}$ ma manca una transizione etichettata `1` uscente dallo stato $q_0$ e una transizione etichettata `0` uscente dallo stato $q_1$.

**Esercizio 2.5.** Aggiungere alla classe `DFA` un metodo `toDOT` che stampa una rappresentazione testuale dell'automa compatibile con l'input del tool `dot` di GraphViz (vedi `http://www.graphviz.org`). Per esempio, il metodo `toDOT` invocato con il parametro `"nome_automa"` su una istanza di `DFA` che rappresenta l'automa di Figura 1 deve produrre il seguente testo:

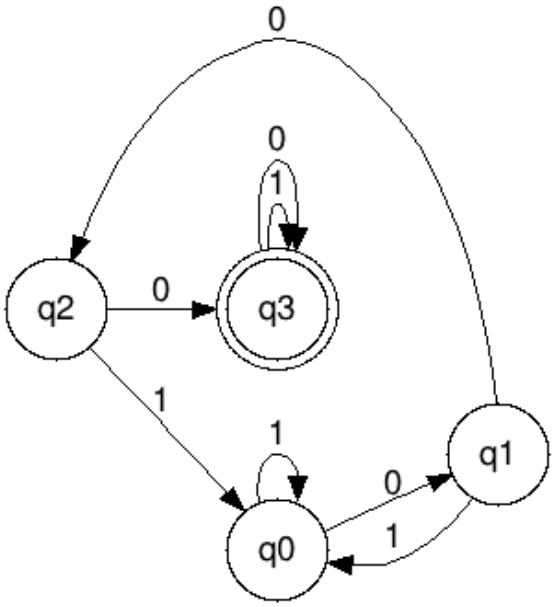



Figura 3: Rendering `dot` dell'automa di Figura 1.

```
digraph automa {
  rankdir=LR;
  node [shape = doublecircle];
  q3;
  node [shape = circle];
  q0 -> q1 [ label = "0" ];
  q1 -> q0 [ label = "1" ];
  q0 -> q0 [ label = "1" ];
  q1 -> q2 [ label = "0" ];
  q2 -> q3 [ label = "0" ];
  q3 -> q3 [ label = "1" ];
  q2 -> q0 [ label = "1" ];
  q3 -> q3 [ label = "0" ];
}
```

Verificare la correttezza della rappresentazione prodotta visualizzando il grafo con il tool `dot`, disponibile anche online agli indirizzi:

- `http://graphviz-dev.appspot.com`
- `http://www.webgraphviz.com`
- `http://stamm-wilbrandt.de/GraphvizFiddle/`

Il testo di cui sopra produce il grafo mostrato in Figura 3.

**Esercizio 2.6** (opzionale)**.** Come raffinamento dell'esercizio precedente, fare in modo che nella rappresentazione testuale dell'automa nel formato `dot` ogni coppia di stati $(p, q)$ sia collegata da al massimo un arco etichettato con l'elenco di tutti i simboli che compaiono come etichette delle transizioni da $p$ a $q$. Per esempio, l'istanza di `DFA` che rappresenta l'automa di Figura 1 deve produrre il seguente testo:

```
digraph automa {
  rankdir=LR;
  node [shape = doublecircle];
  q3;
  node [shape = circle];
  q0 -> q1 [ label = "0" ];
  q1 -> q0 [ label = "1" ];
  q0 -> q0 [ label = "1" ];
  q1 -> q2 [ label = "0" ];
  q2 -> q3 [ label = "0" ];
  q3 -> q3 [ label = "0,1" ];
  q2 -> q0 [ label = "1" ];
}
```

in cui vi è un solo arco da $q_3$ a $q_3$ etichettato con i due simboli `0` e `1` invece che due archi, ciascuno etichettato con un solo simbolo, come in Figura 3.

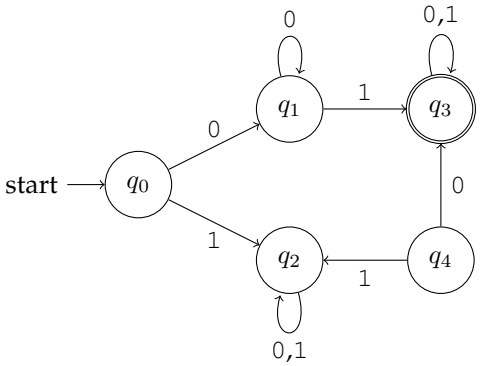
**Esercizio 2.7.** Aggiungere alla classe `DFA` un metodo `toJava` che accetta come argomento una stringa `name` e stampa la classe Java di nome `name` con un metodo `scan` che riconosce tutte e sole le stringhe accettate dall'automa, secondo lo schema visto in Sezione 1. Per esempio, l'esecuzione del seguente programma

```
DFA tz = new DFA(4);
tz.setMove(0, '1', 0);
tz.setMove(0, '0', 1);
tz.setMove(1, '1', 0);
tz.setMove(1, '0', 2);
tz.setMove(2, '1', 0);
tz.setMove(2, '0', 3);
tz.setMove(3, '1', 3);
tz.setMove(3, '0', 3);
tz.addFinalState(3);
tz.toJava("TreZeri");
```

deve produrre il codice in Figura 2.

# 3 Problemi di raggiungibilità

Se consideriamo il DFA raffigurato qui sotto




notiamo una particolarità che riguarda lo stato $q_4$: questo stato non è raggiungibile da alcun altro stato dell'automa e pertanto è uno stato *inutile*, nel senso che $q_4$ può essere eliminato senza modificare il linguaggio riconosciuto dall'automa. In questa sezione studieremo un algoritmo per individuare gli eventuali stati inutili di un DFA.

Il primo passo per la definizione dell'algoritmo che scopre gli stati inutili è la caratterizzazione formale di tali stati. Dato un DFA $A = (\mathbb{Q}, \Sigma, q_0, \delta, F)$ definiamo $\mathcal{R}_A(q)$ come l'insieme degli stati $p$ che possono essere *raggiunti* dallo stato $q$ attraverso un percorso etichettato $w$ nell'automa $A$. Formalmente:

$$\mathcal{R}_A(q) = \{p \in \mathbb{Q} \mid \exists w \in \Sigma^* : \hat{\delta}(q, w) = p\} \tag{3.1}$$

Per esempio, nel DFA precedente abbiamo:

$$\begin{aligned} \hat{\delta}(q_0, \varepsilon) &= q_0 \\ \hat{\delta}(q_0, \texttt{0}) &= q_1 \\ \hat{\delta}(q_0, \texttt{1}) &= q_2 \\ \hat{\delta}(q_0, \texttt{01}) &= q_3 \end{aligned}$$

dunque deduciamo che gli stati $q_0$, $q_1$, $q_2$ e $q_3$ sono tutti raggiungibili da $q_0$. Al contrario, lo stato $q_4$ *non* è raggiungibile da $q_0$, in quanto $\hat{\delta}(q_0, w) \neq q_4$ per ogni $w \in \Sigma^*$. Notiamo che gli stati $q_2$ e $q_3$ sono raggiungibili anche da $q_4$.

Diciamo che uno stato $q$ dell'automa $A$ è:

- *inutile* se $q \notin \mathcal{R}_A(q_0)$; uno stato inutile non viene mai raggiunto dallo stato iniziale dell'automa, dunque non gioca nessun ruolo nel determinare il linguaggio accettato dall'automa e può essere eliminato. Nell'automa visto poc'anzi, lo stato $q_4$ è inutile.
- *pozzo* se $\mathcal{R}_A(q) \cap F = \emptyset$; una volta raggiunto uno stato pozzo, l'automa non è più in grado di accettare alcuna stringa. Nell'automa visto poc'anzi, $q_2$ è l'unico stato pozzo.

  Possiamo ricondurre il problema di verificare se un DFA riconosce il linguaggio vuoto al problema di verificare se il suo stato iniziale è pozzo. Siccome nell'automa qui sopra lo stato iniziale $q_0$ non è uno stato pozzo, possiamo affermare che l'automa riconosce almeno una stringa, per esempio `01`.

Gli esercizi che seguono richiedono l'implementazione di un metodo che accetta come argomento uno stato `q` di un DFA e restituisce l'insieme degli stati raggiungibili da `q`. A tal fine, la definizione (3.1) non è particolarmente utile, in quando in generale $\Sigma^*$ contiene infinite stringhe e dunque non è possibile determinare l'insieme degli stati raggiungibili da $q$ semplicemente provando *tutti* i possibili percorsi. L'algoritmo mostrato in Tabella 1 permette di calcolare in tempo finito tutti gli stati raggiungibili da uno stato $q$ dato.

**Esercizio 3.1.** Aggiungere un metodo `reach` alla classe `DFA` che implementa l'algoritmo di raggiungibilità (Tabella 1). Usare il metodo `reach` per implementare:

- un metodo `empty` che ritorna **`true`** se e solo se l'automa riconosce il linguaggio vuoto.
- un metodo `sink` che ritorna l'insieme degli stati pozzo dell'automa.

Testare il corretto funzionamento dei metodi `reach`, `empty` e `sink` su alcuni DFA appositamente costruiti.

**Esercizio 3.2.** Implementare un metodo `samples` che ritorna un insieme di stringhe campione accettate dall'automa, una per ogni stato finale dell'automa. **Suggerimento:** La struttura del metodo `samples` è fondamentalmente identica a quella del metodo `reach`: basta raffinare l'algoritmo di raggiungibilità in modo da tenere traccia, per ogni stato $p$ raggiungibile dallo stato iniziale $q_0$ dell'automa, di un esempio di stringa $w$ che consente di raggiungere $p$ da $q_0$, ovvero tale che $\hat{\delta}(q_0, w) = p$. Per fare ciò, cambiare il vettore `r` in modo che contenga stringhe invece che **`boolean`** e in questo vettore usare il valore **`null`** invece che **`false`** per marcare gli stati irraggiungibili.

**Input:** uno stato `q` dell'automa.
**Output:** l'insieme `s` degli stati dell'automa raggiungibili da `q`.
**Procedura:**

1. Allocare un vettore `r` di **`boolean`** con tanti elementi quanti sono gli stati dell'automa.

   *Commento: l'elemento `r[i]` del vettore serve a indicare se lo stato (con indice) `i` è raggiungibile da `q` oppure no.*

2. Inizializzare il vettore in modo tale che tutti gli elementi con indice diverso da `q` siano **`false`** e l'elemento con indice `q` sia **`true`**.

   *Giustificazione: lo stato `q` è sempre raggiungibile da se stesso, attraverso il percorso $\varepsilon$ (la stringa vuota). Per gli altri stati, in questo passo dell'algoritmo assumiamo che essi **non** siano raggiungibili da `q`, salvo eventualmente scoprire che lo sono nei passi successivi.*

3. Per ogni indice `i` tale che `r[i]` è **`true`** e ogni carattere `ch` tale che l'automa ha una transizione da `i` a `j` etichettata `ch`, si pone l'elemento `r[j]` a **`true`**.

   *Giustificazione: se `i` è raggiungibile da `q` e c'è una transizione da `i` a `j`, allora anche `j` è raggiungibile da `q`.*

4. Ripetere il passo precedente fintantoché vengono scoperti nuovi stati raggiungibili.

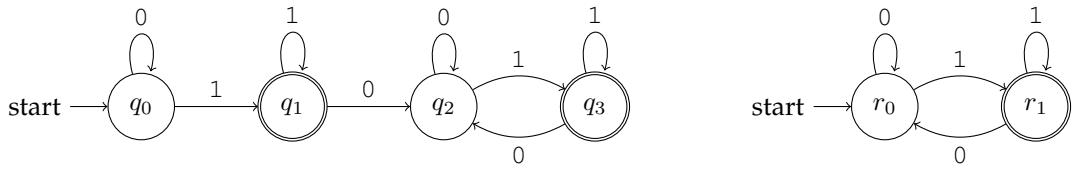
   *Commento: il numero di stati è finito, dunque prima o poi si arriva a un punto in cui non vengono più scoperti stati raggiungibili.*

5. Allocare e ritornare una istanza `s` della classe `HashSet<Integer>` contenente tutti e soli gli indici `i` del vettore `r` tali che `r[i]` è **`true`**.

Tabella 1: Algoritmo di raggiungibilità.

# 4 Minimizzazione di DFA

I DFA $A$ e $B$ raffigurati qui sotto




sono equivalenti, nel senso che $A$ che $B$ riconoscono lo stesso linguaggio $L$ delle stringhe di `0` e `1` che terminano con almeno un `1`. Tuttavia, il DFA $B$ è preferibile ad $A$ in quanto ha un numero inferiore di stati. Di più, si può dimostrare che $B$ è il DFA *minimo* (ovvero con il numero *minimo* di stati) che riconosce il linguaggio $L$.

In questa sezione affronteremo il problema di determinare l'automa minimo equivalente ad un DFA dato. In particolare, definiremo un metodo `minimize` che, invocato su una istanza `x` della classe `DFA`, restituisce una nuova istanza di `DFA` che rappresenta l'automa minimo equivalente a quello rappresentato da `x`.

Per l'implementazione di `minimize` seguiremo l'algoritmo di minimizzazione detto "algoritmo riempi-tabella" del libro di testo. Tale algoritmo consta di due fasi distinte:

1. Nella prima fase, l'algoritmo determina una relazione di *indistinguibilità* tra stati dell'automa. Intuitivamente, due stati $p$ e $q$ di un DFA $A$ sono indistinguibili se le stringhe che portano da $p$ a uno stato finale sono anche stringhe che portano $q$ a uno stato finale (non necessariamente uguale a quello di prima) e viceversa. Formalmente, fissato $A = (\mathbb{Q}, \Sigma, q_0, \delta, F)$, diciamo che $p$ e $q$ sono indistinguibili se per ogni $w \in \Sigma^*$ abbiamo

   $$\hat{\delta}(q, w) \in F \iff \hat{\delta}(p, w) \in F$$

   È facile dimostrare che l'indistinguibilità tra stati è una *relazione di equivalenza*. In conseguenza di ciò, essa determina una *partizione* degli stati dell'automa tale che stati appartenenti alla stessa partizione sono indistinguibili tra loro, mentre stati appartenenti a partizioni diverse sono distinguibili.

2. Nella seconda fase, l'algoritmo costruisce un nuovo DFA ottenuto come "quoziente" del DFA originario rispetto alla relazione di indistinguibilità. L'aspetto cruciale di questa fase è la scelta di un *rappresentante canonico* di ogni partizione di stati del DFA originario. Ricordando che gli stati di un automa vengono rappresentati con numeri interi, faremo la scelta di usare, come rappresentante canonico di una partizione, lo stato con indice più piccolo.

**Esercizio 4.1.** Aggiungere un metodo `minimize` alla classe `DFA` che implementa l'algoritmo di minimizzazione (Tabella 2). Verificare il corretto funzionamento del metodo `minimize` su alcuni DFA appositamente costruiti. **Nota:** l'input dell'algoritmo in Tabella 2 è l'istanza di `DFA` su cui `minimize` viene invocato. In altri termini, `minimize` è un metodo di `DFA` *senza argomenti*.

**NOTA**

Non è detto che il numero `k` determinato al passo 6 dell'algoritmo in Tabella 2 sia effettivamente più piccolo di `n`, anche se il DFA originario contiene stati indistinguibili. Ciò che conta, ai fini della minimizzazione, è il numero di chiavi della tabella `transitions`.

**Esercizio 4.2.** Aggiungere un metodo `equivalentTo` alla classe `DFA` che determina se due DFA sono equivalenti (fare riferimento al libro di testo per la descrizione dell'algoritmo). Organizzare l'implementazione dell'algoritmo in modo da evitare la duplicazione di parti consistenti di codice nella classe `DFA`, eventualmente definendo opportuni metodi ausiliari privati. **Nota:** il metodo **non** deve modificare in alcun modo l'istanza di `DFA` su cui viene invocato.

**Input:** un DFA $A$.
**Output:** il DFA minimo equivalente ad $A$.
**Procedura:**

1. Allocare una matrice `eq` di elementi di tipo **`boolean`** e dimensioni $n \times n$, dove $n$ è il numero di stati dell'automa $A$.

   *Commento: scegliamo di rappresentare la relazione di indistinguibilità tra gli stati dell'automa $A$ come una matrice (simmetrica)* `eq`*. Per ogni coppia di stati (*`i`*,* `j`*) l'elemento* `eq[i][j]` *indica se* `i` *e* `j` *sono indistinguibili o no.*

2. Inizializzare la matrice in modo tale che l'elemento `eq[i][j]` sia **`true`** se `i` e `j` sono entrambi finali o entrambi non finali, **`false`** altrimenti.

   *Giustificazione: l'algoritmo assume inizialmente che due stati entrambi (non) finali sono indistinguibili, salvo eventualmente scoprire che sono distinguibili nei passi successivi. Due stati di cui solo uno è finale sono certamente distinguibili: da uno è possibile riconoscere la stringa vuota, dall'altro no.*

3. Per ogni coppia di stati `i` e `j` ed ogni carattere `ch` tali che `eq[i][j]` è **`true`** ed `eq[move(i,ch)][move(j,ch)]` è **`false`**, si pone l'elemento `eq[i][j]` a **`false`**.

   *Giustificazione: se da due stati* `i` *e* `j` *assunti indistinguibili si raggiungono, per mezzo di transizioni etichettate con lo stesso simbolo* `ch`*, altri due stati distinguibili, allora anche* `i` *e* `j` *sono distinguibili.*

4. Ripetere il passo precedente fintantoché vengono scoperte nuove coppie di stati distinguibili.

   *Commento: il numero di stati è finito, dunque prima o poi si arriva a un punto in cui non vengono più scoperte nuove coppie di stati distinguibili.*

5. Allocare un vettore `m` di elementi di tipo **`int`** e dimensione $n$ e inizializzarlo in modo tale che l'elemento `i`-esimo sia lo stato indistinguibile da `i` con indice più piccolo.

   *Giustificazione: il vettore* `m` *mappa ogni stato* `i` *al rappresentante canonico* `m[i]` *della classe di equivalenza a cui* `i` *appartiene. Scegliamo come rappresentate canonico di una classe di equivalenza lo stato della classe con indice più piccolo.*

6. Sia `k` l'elemento più grande del vettore `m`. Allocare e inizializzare un DFA $B$ con `k + 1` stati e tale che per ogni transizione da `i` a `j` etichettata `ch` in $A$ esiste una transizione da `m[i]` a `m[j]` etichettata `ch` in $B$. Fare in modo che, se `i` è finale in $A$, allora `m[i]` sia finale in $B$.

   *Giustificazione: l'automa $B$ è l'immagine dell'automa $A$ attraverso la relazione di indistinguibilità rappresentata da* `eq`*.*

7. Ritornare il DFA $B$.

Tabella 2: Algoritmo di minimizzazione.

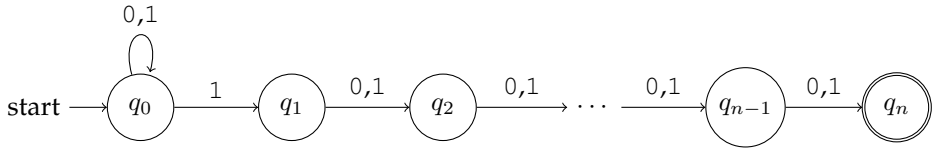



Figura 4: Esempio di NFA con equivalente deterministico sfavorevole.

# 5 Rappresentazione di $\varepsilon$-NFA in Java

In questa sezione vedremo come rappresentare in Java $\varepsilon$-NFA arbitrari. In particolare, definiremo una classe `NFA` che fornisce metodi per costruire un $\varepsilon$-NFA ed il DFA equivalente. La caratteristica saliente che distingue DFA ed $\varepsilon$-NFA è che la funzione di transizione di questi ultimi ha tipo

$$\delta : \mathbb{Q} \times (\Sigma \cup \{\varepsilon\}) \to \wp(\mathbb{Q})$$

ovvero è una funzione che, dato uno stato $p$ dell'automa ed un simbolo dell'alfabeto oppure il simbolo speciale $\varepsilon$, ritorna l'*insieme degli stati* in cui l'automa *può* transire partendo da $p$ ed eventualmente riconoscendo un simbolo dell'alfabeto (nel caso di $\varepsilon$, l'automa esegue una transizione "spontanea" da uno stato ad un altro). Conseguentemente, la rappresentazione di $\varepsilon$-NFA in Java sarà molto simile a quella già utilizzata per i DFA, salvo appunto per qualche dettaglio che riguarda la rappresentazione della funzione di transizione.

Nella sezione 2 abbiamo scelto di rappresentare la funzione di transizione di un DFA con una tabella hash `HashMap<K,V>` le cui chiavi rappresentano elementi del *dominio* della funzione $\delta$ e i cui valori rappresentano elementi del *codominio* della funzione $\delta$. Nel caso di $\varepsilon$-NFA, operiamo le seguenti scelte:

- I valori della tabella hash sono istanze di `HashSet<Integer>`, in accordo con il fatto che in un $\varepsilon$-NFA la funzione di transizione ritorna un *insieme* di stati, invece di un solo stato.
- Le chiavi della tabella hash continuano ad essere istanze della classe `Move`, ma utilizziamo il carattere nullo `'\0'` per rappresentare l'etichetta di una $\varepsilon$ transizione.

Il listato 3 illustra l'implementazione (incompleta) della classe `NFA` con questa nuova rappresentazione della funzione di transizione.

**Esercizio 5.1.** Implementare tutti i metodi incompleti della classe `NFA` mostrata nel Listato 3. Per l'implementazione del metodo `epsilonClosure` fare riferimento all'algoritmo in Tabella 3. Verificare il funzionamento del metodo `dfa` su alcune istanze di `NFA` rappresentanti automi non deterministici, alcuni con ed altri senza $\varepsilon$-transizioni. **Nota:** se un metodo è incompleto nel Listato 3 significa che differisce in qualche modo dalla sua controparte nella classe `DFA`.

**Esercizio 5.2.** Implementare un metodo *statico* `nth` della classe `NFA` che, dato un numero naturale $n$, produce l'automa non deterministico avente $n + 1$ stati che riconosce le stringhe di `0` e `1` tali che l'$n$-esimo simbolo *da destra* sia `1` (Figura 4). Verificare il fenomeno dell'esplosione degli stati stampando, per ogni numero `i` in un intervallo $[1, N]$ ragionevole, il numero di stati dell'NFA ritornato da `NFA.nth(i)`, il numero di stati di `NFA.nth(i).dfa()`, ed il numero di stati di `NFA.nth(i).dfa().minimize()`.

**Esercizio 5.3.** Studiare l'implementazione del metodo `dfa`.

**Esercizio 5.4** (opzionale)**.** Dotare la classe `NFA` di un metodo `toDOT` analogo a quello già realizzato per la classe `DFA`. Ha senso dotare la classe `NFA` anche del metodo `toJava`? Argomentare.

**Esercizio 5.5** (opzionale)**.** Individuare le modifiche minime da apportare all'algoritmo in Tabella 3 affinché calcoli *direttamente* l'$\varepsilon$-chiusura di un *insieme di stati* invece che di uno stato solo.

**Input:** uno stato `q` dell'automa.
**Output:** l'$\varepsilon$-chiusura di `q`.
**Procedura:**

1. Allocare un vettore `r` di **`boolean`** con tanti elementi quanti sono gli stati dell'automa.

   *Commento: l'elemento* `r[i]` *del vettore serve a indicare se lo stato (con indice)* `i` *è raggiungibile da* `q` *per mezzo di zero o più $\varepsilon$-transizioni.*

2. Inizializzare il vettore in modo tale che tutti gli elementi con indice diverso da `q` siano **`false`** e l'elemento con indice `q` sia **`true`**.

   *Giustificazione: lo stato* `q` *è sempre raggiungibile da se stesso per mezzo di zero $\varepsilon$-transizioni. Per gli altri stati, in questo passo dell'algoritmo assumiamo che essi* ***non*** *siano raggiungibili da* `q` *per mezzo di $\varepsilon$-transizioni, salvo eventualmente scoprire che lo sono nei passi successivi.*

3. Per ogni indice `i` tale che `r[i]` è **`true`** e ogni $\varepsilon$-transizione da `i` a `j`, si pone l'elemento `r[j]` a **`true`**.

   *Giustificazione: se* `i` *è raggiungibile da* `q` *per mezzo di $\varepsilon$-transizioni e c'è una $\varepsilon$-transizione da* `i` *a* `j`*, allora anche* `j` *è raggiungibile da* `q` *per mezzo di $\varepsilon$-transizioni.*

4. Ripetere il passo precedente fintantoché vengono scoperti nuovi stati raggiungibili da `q` per mezzo di $\varepsilon$-transizioni.

   *Commento: il numero di stati è finito, dunque prima o poi si arriva a un punto in cui non vengono più scoperti stati raggiungibili.*

5. Allocare e ritornare una istanza `s` della classe `HashSet<Integer>` contenente tutti e soli gli indici `i` del vettore `r` tali che `r[i]` è **`true`**.

Tabella 3: Algoritmo di calcolo dell'$\varepsilon$-chiusura di uno stato.

Listing 3: `NFA.java`

```
import java.util.HashSet;
import java.util.HashMap;
import java.util.Stack;

/**
 * Un oggetto della classe NFA rappresenta un automa a stati finiti
 * non deterministico con epsilon transizioni
 */
public class NFA
{
    /**
     * Usiamo il carattere nullo per rappresentare una epsilon
     * transizione
     */
    public static final char EPSILON = '\0';

    /**
     * Numero degli stati dell'automa. Ogni stato e` rappresentato da
     * un numero interno non negativo, lo stato con indice 0 e` lo
     * stato iniziale.
     */
    private int numberOfStates;

    /** Insieme degli stati finali dell'automa. */
    private HashSet<Integer> finalStates;

    /**
     * Funzione di transizione dell'automa, rappresentata come una
     * mappa da mosse a insiemi di stati di arrivo.
     */
    private HashMap<Move, HashSet<Integer>> transitions;

    /**
     * Crea un NFA con un dato numero di stati.
     * @param  n Il numero di stati dell'automa.
     */
    public NFA(int n) {
        numberOfStates = n;
        finalStates = new HashSet<Integer>();
        transitions = new HashMap<Move, HashSet<Integer>>();
    }

    /**
     * Aggiunge uno stato all'automa.
     * @return L'indice del nuovo stato creato
     */
    public int newState() {
        return numberOfStates++;
    }

    /**
     * Aggiunge uno stato finale.
     * @param  p Lo stato che si vuole aggiungere a quelli finali.
     * @return <code>true</code> se lo stato e` valido,
     *         <code>false</code> altrimenti.
     */
```

```
    public boolean addFinalState(int p) {
        if (validState(p)) {
            finalStates.add(p);
            return true;
        } else
            return false;
    }

    /**
     * Determina se uno stato e' valido oppure no.
     * @param  p Lo stato da controllare.
     * @return <code>true</code> se lo stato e' valido,
     *         <code>false</code> altrimenti.
     * @see #numberOfStates
     */
    public boolean validState(int p) {
        return (p >= 0 && p < numberOfStates);
    }

    /**
     * Determina se uno stato e' finale oppure no.
     * @param  p Lo stato da controllare.
     * @return <code>true</code> se lo stato e' finale,
     *         <code>false</code> altrimenti.
     * @see #finalStates
     */
    public boolean finalState(int p) {
        return finalStates.contains(p);
    }

    /**
     * Restituisce il numero di stati dell'automa.
     * @return Numero di stati.
     */
    public int numberOfStates() {
        return numberOfStates;
    }

    /**
     * Aggiunge una transizione all'automa.
     * @param  p  Lo stato di partenza della transizione.
     * @param  ch Il simbolo che etichetta la transizione.
     * @param  q  Lo stato di arrivo della transizione.
     * @return <code>true</code> se lo stato di partenza e lo stato di
     *         arrivo sono validi, <code>false</code> altrimenti.
     */
    public boolean addMove(int p, char ch, int q) {
        // IMPLEMENTARE
    }

    /**
     * Determina se c'e' uno stato finale in un insieme di stati.
     * @param  s L'insieme di stati da controllare.
     * @return <code>true</code> se c'e' uno stato finale in
     *         <code>s</code>, <code>false</code> altrimenti.
     * @see #finalStates
     */
    private boolean finalState(HashSet<Integer> s) {
```

```
        for (int p : s)
            if (finalState(p))
                return true;
        return false;
    }

    /**
     * Restituisce l'alfabeto dell'automa, ovvero l'insieme di simboli
     * che compaiono come etichette delle transizioni
     * dell'automa. Notare che <code>EPSILON</code> non e' un simbolo.
     * @return L'alfabeto dell'automa.
     */
    public HashSet<Character> alphabet() {
        // IMPLEMENTARE
    }

    /**
     * Esegue una mossa dell'automa.
     * @param p  Stato di partenza prima della transizione.
     * @param ch Simbolo da riconoscere.
     * @return Insieme di stati di arrivo dopo la transizione. Questo
     *         insieme puo' essere vuoto.
     */
    public HashSet<Integer> move(int p, char ch) {
        // IMPLEMENTARE
    }

    /**
     * Esegue una mossa dell'automa.
     * @param s  Insieme di stati di partenza prima della transizione.
     * @param ch Simbolo da riconoscere.
     * @return Insieme di stati di arrivo dopo la transizione. Questo
     *         insieme puo' essere vuoto.
     */
    public HashSet<Integer> move(HashSet<Integer> s, char ch) {
        HashSet<Integer> qset = new HashSet<Integer>();
        for (int p : s)
            qset.addAll(move(p, ch));
        return qset;
    }

    /**
     * Calcola la epsilon chiusura di un insieme di stati dell'automa.
     * @param s  Insieme di stati di cui calcolare l'epsilon chiusura.
     * @return Insieme di stati raggiungibili da quelli contenuti in
     *         <code>s</code> per mezzo di zero o piu' epsilon
     *         transizioni.
     */
    public HashSet<Integer> epsilonClosure(HashSet<Integer> s) {
        HashSet<Integer> qset = new HashSet<Integer>();
        for (int p : s)
            qset.addAll(epsilonClosure(p));
        return qset;
    }

    /**
     * Calcola la epsilon chiusura di uno stato dell'automa. E' un
     * caso specifico del metodo precedente.
```

```
     * @param p  Insieme di cui calcolare l'epsilon chiusura.
     * @return Insieme di stati raggiungibili da <code>p</code> per
     *         mezzo di zero o piu' epsilon transizioni.
     * @see #epsilonClosure
     */
    public HashSet<Integer> epsilonClosure(int p) {
        // IMPLEMENTARE
    }

    /**
     * Calcola l'automa a stati finiti deterministico equivalente.
     * @return DFA equivalente.
     */
    public DFA dfa()
    {
        // la costruzione del DFA utilizza due tabelle hash per tenere
        // traccia della corrispondenza (biunivoca) tra insiemi di
        // stati del NFA e stati del DFA
        HashMap<HashSet<Integer>, Integer> indexOfSet =
            new HashMap<HashSet<Integer>, Integer>();    // NFA -> DFA
        HashMap<Integer, HashSet<Integer>> setOfIndex =
            new HashMap<Integer, HashSet<Integer>>();    // DFA -> NFA

        DFA dfa = new DFA(1);                            // il DFA
        Stack<Integer> newStates = new Stack<Integer>(); // nuovi stati del DFA
        HashSet<Character> alphabet = alphabet();

        indexOfSet.put(epsilonClosure(0), 0); // stati dell'NFA corrisp. a q0
        setOfIndex.put(0, epsilonClosure(0));
        newStates.push(0);                    // nuovo stato da esplorare

        while (!newStates.empty()) { // finche' ci sono nuovi stati da visitare
            final int p = newStates.pop(); // ne considero uno e lo visito
            final HashSet<Integer> pset = setOfIndex.get(p); // stati del NFA corrisp.
            for (char ch : alphabet) { // considero tutte le possibili transizioni
                HashSet<Integer> qset = epsilonClosure(move(pset, ch));
                if (indexOfSet.containsKey(qset)) { // se qset non e' nuovo...
                    final int q = indexOfSet.get(qset); // recupero il suo indice
                    dfa.setMove(p, ch, q);          // aggiungo la transizione
                } else {                            // se invece qset e' nuovo
                    final int q = dfa.newState();   // creo lo stato nel DFA
                    indexOfSet.put(qset, q);        // aggiorno la corrispondenza
                    setOfIndex.put(q, qset);
                    newStates.push(q);              // q e' da visitare
                    dfa.setMove(p, ch, q);          // aggiungo la transizione
                }
            }
        }

        // stabilisco gli stati finali del DFA
        for (int p = 0; p < dfa.numberOfStates(); p++)
            if (finalState(setOfIndex.get(p)))
                dfa.addFinalState(p);

        return dfa;
    }
}
```

# 6 Compilazione di espressioni regolari

In questa sezione vedremo come rappresentare espressioni regolari in Java e implementare uno schema di traduzione per compilarle in $\varepsilon$-NFA. Le espressioni regolari che prendiamo in considerazione sono i termini $E$ generati dalla grammatica seguente:

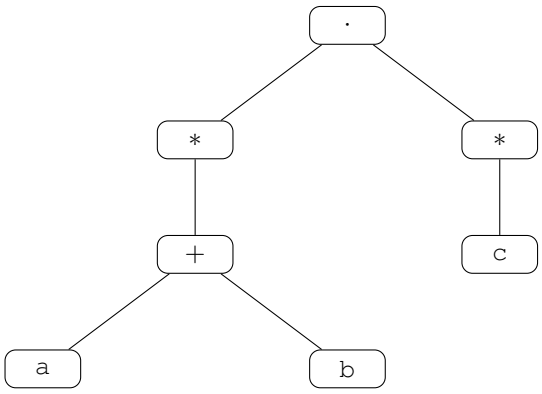
$$E \ ::= \ a \mid \varnothing \mid \varepsilon \mid EE \mid E + E \mid E^*$$

con la semantica usuale (si veda il libro di testo per la definizione di *linguaggio generato* da un'espressione regolare). In questa grammatica, il simbolo $a$ denota un arbitrario carattere rappresentabile con il tipo **`char`** in Java. Come esempio, l'espressione regolare

$$(\mathtt{a} + \mathtt{b})^*\mathtt{c}^* \tag{6.1}$$

denota il linguaggio delle stringhe di `a`, `b` e `c` tali che nessun simbolo `a` o `b` segue un'occorrenza del simbolo `c`.

Al fine di individuare una rappresentazione naturale delle espressioni regolari in Java, è conveniente pensare alla struttura ad albero di un'espressione regolare. Per esempio, l'espressione (6.1) ha la seguente struttura




in cui i vari operatori delle espressioni regolari (composizione sequenziale, scelta, chiusura di Kleene) sono nodi interni dell'albero, mentre le singole occorrenze dei simboli che compongono l'espressione regolare sono foglie dell'albero. Dunque la radice dell'albero è un nodo $\cdot$ che rappresenta la composizione sequenziale di due sotto-espressioni regolari, quelle rappresentate dai due sotto-alberi del nodo: il sotto-albero sinistro rappresenta la chiusura di Kleene $*$ della scelta $+$ tra `a` e `b`; il sotto-albero destro rappresenta la chiusura di Kleene di `c`.

Questa rappresentazione ad albero delle espressioni regolari si presta bene ad essere realizzata in Java per mezzo di oggetti appartenenti ad un'opportuna gerarchia di classi. Più precisamente, rappresenteremo un'espressione regolare come un *albero di oggetti*, ciascuno istanza di una *classe* che denota il tipo di nodo o di foglia rappresentato. Definiremo dunque il seguente insieme di classi Java:

- `RegExpSymbol` per rappresentare una foglia contenente un simbolo come `a` o `b`;
- `RegExpEpsilon` per rappresentare una foglia $\varepsilon$;
- `RegExpEmpty` per rappresentare una foglia $\varnothing$;
- `RegExpSequence` per rappresentare un nodo $\cdot$, la composizione sequenziale;
- `RegExpChoice` per rappresentare un nodo $+$, la scelta;
- `RegExpStar` per rappresentare un nodo $*$, la chiusura di Kleene.

Definiamo inoltre una *interfaccia* `RegExp` che descrive i metodi comuni a tutte le espressioni regolari e che tutte le classi elencate qui sopra devono implementare. L'interfaccia `RegExp` è definita come segue:

Listing 4: `RegExp.java`

```
interface RegExp {
    NFA compile();
}
```

Ovvero, ogni classe `RegExp*` deve esportare un metodo `compile` la cui funzione è quella di *compilare* l'espressione regolare rappresentata nel corrispondente $\varepsilon$-NFA.

Di seguito è mostrata l'implementazione della classe `RegExpSymbol`.

Listing 5: `RegExpSymbol.java`

```
public class RegExpSymbol implements RegExp {
    private char ch;

    RegExpSymbol(char ch) {
        this.ch = ch;
    }

    public NFA compile() {
        NFA a = new NFA(2);
        a.addMove(0, ch, 1);
        a.addFinalState(1);
        return a;
    }
}
```

Notiamo che la classe contiene un campo privato `ch` che serve per memorizzare il particolare simbolo che vogliamo generare attraverso l'espressione regolare. Per esempio, l'istanza

```
new RegExpSymbol('a')
```

rappresenta in Java l'espressione regolare `a`, mentre

```
new RegExpSymbol('1')
```

rappresenta in Java l'espressione regolare `1`. L'invocazione del metodo `compile` deve produrre l'$\varepsilon$-NFA corrispondente a una espressione regolare. Dunque

```
new RegExpSymbol('a').compile()
```

è un'istanza di `NFA` costruita come indicato nel metodo `compile` del Listato 5: si tratta di un $\varepsilon$-NFA con due stati, lo stato iniziale con indice 0 e uno stato finale con indice 1, collegati da una transizione etichettata `a`. Per costruzione, tale $\varepsilon$-NFA riconosce la sola stringa `a`. Per semplificare l'implementazione del metodo `compile` nelle altre classi `RegExp*`, faremo le seguenti scelte:

1. l'istanza di `NFA` ritornata dal metodo `compile` ha sempre un **solo** stato finale;
2. tale stato finale ha sempre indice 1.

Sapendo che lo stato iniziale di un automa a stati finiti, secondo la nostra rappresentazione, è sempre quello con indice 0, l'uso di queste convenzioni ci permette di individuare facilmente gli "ingressi" e le "uscite" di ogni $\varepsilon$-NFA ottenuto attraverso il metodo `compile`, in modo che sia semplice collegare tra loro gli $\varepsilon$-NFA nelle costruzioni più complesse.

La compilazione di espressioni regolari *composte* (composizione sequenziale, scelta, chiusura di Kleene) richiede la costruzione di $\varepsilon$-NFA a partire da altri $\varepsilon$-NFA più piccoli. Al fine di realizzare tale costruzione in modo incrementale, è conveniente dotare la classe `NFA` di un metodo `append` che fonde la struttura di un $\varepsilon$-NFA a quella dell'automa rappresentato da **`this`**. Il metodo `append` è implementato come segue

```
    public int append(NFA a) {
        final int n = numberOfStates;
        numberOfStates += a.numberOfStates();
        for (Move m : a.transitions.keySet())
            for (int q : a.transitions.get(m))
                addMove(n + m.start, m.ch, n + q);
        return n;
    }
```

ed è una semplice variante della procedura di unione di automi che abbiamo già incontrato nel risolvere l'esercizio 4.2. Notare che il metodo `append` ritorna il numero `n` di stati dell'automa *prima* della modifica. Sapendo che lo stato iniziale dell'$\varepsilon$-NFA `a` è quello con indice 0, ciò significa che nel nuovo automa lo stato iniziale della copia di `a` è diventato `n`. Notare inoltre che gli stati finali di `a` **non** vengono marcati come finali nell'automa risultante. L'individuazione degli stati finali è lasciata alle implementazioni del metodo `compile` delle varie classi `RegExp*`.

Grazie al metodo `append` possiamo realizzare la classe `RegExpSequence` come segue.

Listing 6: `RegExpSequence.java`

```
public class RegExpSequence implements RegExp {
    private RegExp e1;
    private RegExp e2;

    RegExpSequence(RegExp e1, RegExp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    public NFA compile() {
        NFA a = new NFA(2);
        final int n = a.append(e1.compile());
        final int m = a.append(e2.compile());
        a.addMove(0, NFA.EPSILON, n);
        a.addMove(n + 1, NFA.EPSILON, m);
        a.addMove(m + 1, NFA.EPSILON, 1);
        a.addFinalState(1);
        return a;
    }
}
```

In particolare, usiamo la convenzione sulla numerazione degli stati degli $\varepsilon$-NFA ritornati da `compile` ed il numero ritornato da `append` per individuare con facilità l'indice degli stati che devono essere connessi tra loro.

**Esercizio 6.1.** Implementare le classi `RegExpEpsilon`, `RegExpEmpty`, `RegExpChoice`, `RegExpStar` seguendo la traccia data qui sopra. Verificare il corretto funzionamento del metodo `compile` producendo l'$\varepsilon$-NFA corrispondente a un insieme ragionevole di espressioni regolari di complessità crescente. Per esempio:

```
new RegExpChoice(new RegExpSymbol('a'),
                 new RegExpSymbol('b')).compile().toDOT();
```

Si consiglia l'implementazione del metodo `toDOT` anche per la classe `NFA` come richiesto nell'esercizio 5.4, in modo da poter visualizzare gli $\varepsilon$-NFA risultanti dalla compilazione delle espressioni regolari per mezzo del tool `dot`.

**Esercizio 6.2.** Individuare un'espressione regolare $E$ definita sull'alfabeto $\{/, *, c\}$ che generi le sequenze di almeno 4 caratteri che iniziano con `/*`, che finiscono con `*/`, e che contengono una sola occorrenza della sequenza `*/`, quella finale. Istanziare oggetti delle classi `RegExp*` per rappresentare $E$ in Java e produrre l'$\varepsilon$-NFA corrispondente attraverso il metodo `compile`. Verificare la correttezza dell'$\varepsilon$-NFA corrispondente, nonché del DFA equivalente, e del DFA minimo

corrispondente a quest'ultimo. Per esempio, verificare che il DFA accetti le stringhe `/****/` e `/*c*c*/` ma non `/*/` oppure `/**/**/`.

**Esercizio 6.3.** Giustificare l'apparente stranezza del DFA ottenuto eseguendo l'istruzione:

```
new RegExpStar(new RegExpChoice(new RegExpSymbol('a'),
                                new RegExpSymbol('b')))
  .compile().dfa().minimize().toDOT();
```

**Esercizio 6.4.** Implementare un metodo statico

```
public static RegExp range(char from, char to);
```

che, dati due caratteri `from` e `to`, crea l'espressione regolare che genera tutti i caratteri nell'intervallo da `from` a `to`, estremi inclusi. Utilizzando tale metodo, costruire l'espressione regolare che genera le costanti numeriche (Esercizio 1.2), ottenere il DFA minimo corrispondente e verificarne la correttezza.