Capitolo 2

Analisi Lessicale

Il compito della fase di analisi lessicale è quello di fornire alle fasi successive uno stream di token a partire dallo stream di caratteri che rappresenta il programma che si vuole compilare. In Figura 2.1 è rappresentato il ruolo dell'analizzatore lessicale rispetto alle componenti del compilatore con cui coopera.

Tramite una chiamata di un metodo (o funzione in C) `get_next_token()` dell'analizzatore, il parser ottiene un nuovo token. Per far questo, l'analizzatore dovrà leggere i caratteri di input e analizzarli fino al riconoscimento di un token. Se questo è un identificatore, si occupa di creare una riga per lui nella tabella dei simboli. Nel caso che nessun token sia riconosciuto, viene segnalato un errore (in una implementazione Java, ad esempio, si potrebbe usare un'eccezione apposita).

L'analizzatore lessicale provvede ad eliminare dal testo gli spazi bianchi[1], che separano i token, ed i commenti. Questi ultimi, infatti, sono utili per il programmatore, ma vengono ignorati dal compilatore.

Un altro compito importante è quello di associare i giusti numeri di riga con i messaggi di errore, in modo tale che il programmatore possa trovare facilmente il punto del programma in cui è stato trovato un errore.

Si noti che l'analisi lessicale potrebbe venire facilmente considerata una parte dell'analisi sintattica. Le seguenti considerazioni sono alcuni dei principali motivi per cui conviene separarle:

- Semplicità della progettazione. Separare le due fasi permette di semplificare una delle due: un parser che non debba tener conto degli spazi bianchi o dei commenti è più semplice da scrivere.

- Efficienza della compilazione migliorata. Un analizzatore lessicale come parte distinta permette di utilizzare tecniche specifiche per questo tipo

[1] Si considerano spazi bianchi i blank, i caratteri di tabulazione ed i caratteri di newline.

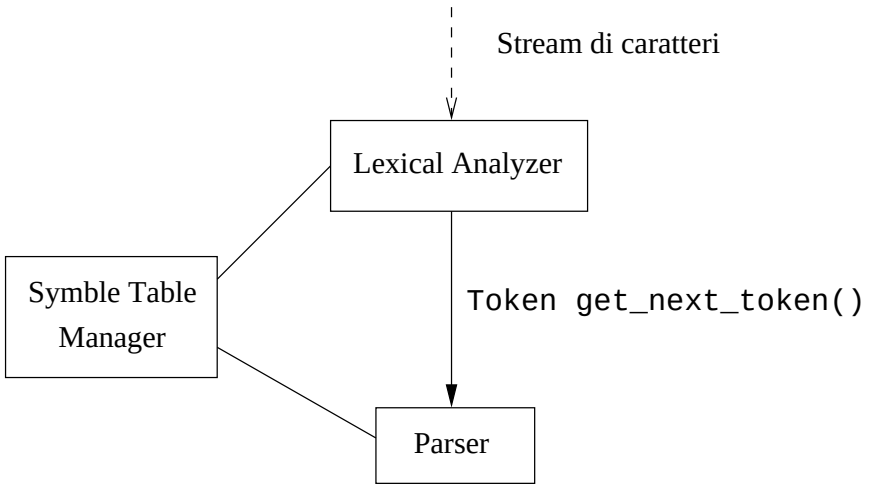


Figura 2.1: Interazioni dell'analizzatore lessicale.

di analisi che migliorano le prestazioni (es. tecniche di bufferizzazione). Inoltre, i formalismi e gli algoritmi usati per descrivere/riconoscere i token sono più semplici rispetto a quelli che sono necessari per riconoscere la struttura del programma. Ne consegue una implementazione più semplice ed efficiente.

2.1 Token, pattern e lexeme

In questa sezione diamo un significato più preciso a dei termini che useremo spesso. Abbiamo già visto informalmente che cosa si intende per token.

In generale ad ogni token dato in output dall'analizzatore lessicale corrisponde un insieme di stringhe dell'input. Questo insieme di stringhe viene descritto tramite da una regola che viene chiamata *pattern* associato al token. Diciamo che un pattern "fa *match*"[2] con ogni stringa nell'insieme associato al token. Un *lexeme* è una sequenza di caratteri nel programma sorgente che fa match con il pattern di un certo token. Ad esempio, nella frase Pascal `const pi = 3.1416;` la sottostringa `pi` è un lexeme per il token "identificatori".

In Tabella 2.2 ci sono una serie di token con relativo pattern (specificato per ora in maniera informale) e alcuni esempi di lexeme.

I token sono trattati come simboli terminali della grammatica che genera il linguaggio sorgente. Come convenzione useremo sempre il grassetto per indicare i token (e i simboli terminali). Ogni token può essere pensato come

[2]In alcuni casi si potrebbe essere tentati di coniare nuovi idiomi tipo "mecciare" (io meccio, tu mecci, egli meccia...), ma in queste note si è preferita la linea linguistica purista.

TOKEN	ESEMPI DI LEXEME	DESCR. INFORM. DEL PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	`<` o `<=` o `=` $\cdots$
id	`pi, count, D2`	lettera seguita da cifre e/o lettere
num	`3.1416, 0, 6.02E32`	Una qualsiasi costante numerica
literal	`" Inserire codice: "`	Tutti i caratteri tra " e " eccetto "
punt	`; , : ( ) { }`	Un carattere di punteggiatura

Figura 2.2: Esempi di token, pattern e lexeme.

una *categoria lessicale* in cui tutti i lexeme con un significato comune vengono raggruppati. Il raggruppamento dipende dal tipo di linguaggio e dalle scelte del progettista. Ad esempio si può creare una categoria lessicale che raggruppa tutte le parole riservate del linguaggio oppure creare una categoria lessicale per ogni parola riservata.

Il ritorno di un token da parte dell'analizzatore lessicale viene sempre implementato ritornando un numero intero che corrisponde al token. Noi denoteremo tale numero intero con il nome della categoria lessicale in grassetto (es. **id**). Vedremo nel seguito che oltre a questo numero, in alcuni casi, l'analizzatore ritorna anche un altro valore.

Un pattern è una regola che descrive l'insieme dei lexeme che possono essere riconosciuti come token all'interno del programma sorgente. Il pattern per il token **const** in Tabella 2.2 è semplicemente lo spelling carattere per carattere dell'unico lexeme della categoria lessicale, ovvero la sequenza `const`. Il pattern per il token **relation** è un insieme di sequenze di caratteri (in questo caso gli operatori relazionali del Pascal). Per descrivere precisamente pattern più complicati, come ad esempio quelli per **id** o **num**, utilizzeremo il formalismo delle *espressioni regolari*, che introdurremo nel seguito.

2.2 Attributi per i token

Quando diversi lexeme fanno match con uno stesso pattern l'analizzatore lessicale deve dare un'informazione ulteriore oltre al numero identificativo del token corrispondente. Questa informazione è importante per le fasi successive della compilazione. Ad esempio sia il carattere `1` che il carattere `0` fanno match con il pattern associato a **num**, ma è essenziale per il generatore di codice intermedio (fase successiva) conoscere esattamente quale numero era presente nel codice sorgente.

Questo tipo di informazioni aggiuntive per i token vengono trattate dal-

l'analizzatore lessicale come *attributi* dei token stessi. Si può dire che il token influenza le decisioni del parser, mentre gli attributi influenzano la traduzione vera e propria dei token. Nella pratica ogni token ha al più un attributo che è un puntatore ad una entrata della tabella dei simboli in cui sono memorizzate tutte le varie informazioni relative al token.

Esempio 2.1 *Consideriamo il seguente statement Fortran:*

```
E = M * C ** 2
```

I seguenti sono i token che sono generati dall'analizzatore lessicale in corrispondenza di questa riga:

- <**id**, *puntatore all'entrata per* `E` *nella tabella dei simboli* >
- <**assign_op**, >
- <**id**, *puntatore all'entrata per* `M` *nella tabella dei simboli* >
- <**mult_op**, >
- <**id**, *puntatore all'entrata per* `C` *nella tabella dei simboli* >
- <**exp_op**, >
- <**num**, *numero intero 2* >

Dall'esempio precedente si vede che in alcuni casi non è necessario specificare nessun attributo perché la prima componente della coppia è sufficiente per identificare il lexeme. Inoltre in questo esempio il valore dell'attributo per il settimo token **num** è il numero intero 2 (cioè la costante intera corrispondente al lexeme). Equivalentemente l'analizzatore lessicale avrebbe potuto inserire il lexeme "`2`" nella tabella dei simboli e mettere come valore dell'attributo per **num** il puntatore all'entrata corrispondente. È una scelta del progettista del compilatore.

2.3 Errori lessicali

L'analizzatore lessicale ha una visione molto locale del programma sorgente e quindi non ha la possibilità di accorgersi di molti errori. Ad esempio, se in un programma C viene incontrata la stringa `fi` nel contesto

```
fi (a== f(x) ) ···
```

l'analizzatore lessicale non è in grado di dire se `fi` è in realtà la stringa `if` scritta male oppure un identificatore non ancora apparso nel programma sorgente. Dato che `fi` è un identificatore valido, esso viene inserito nella

tabella dei simboli e l'errore verrà riconosciuto più tardi in una delle fasi successive (probabilmente l'analisi sintattica).

L'unica situazione in cui l'analizzatore lessicale può riconoscere un errore è quando nessuno dei suoi pattern fa match con un prefisso dell'input rimanente. In questo caso viene segnalato un errore e l'analizzatore può ad esempio utilizzare la strategia "panic mode" per andare avanti e riconoscere il prossimo token.

2.4 Specificare i token

Quella delle espressioni regolari è una notazione importante per specificare i pattern. Le espressioni regolari sono usate in molti contesti, oltre a quello che vedremo in questo corso. Esse denotano insiemi di stringhe e possono essere considerate un metalinguaggio per specificare linguaggi (regolari).

2.4.1 Stringhe e linguaggi

Cominciamo ad introdurre alcune convenzioni che serviranno da qui in poi. I termini *alfabeto* o *classe di caratteri* denotano un qualsiasi insieme *finito* di simboli. Ad esempio l'insieme $\{0, 1\}$ è l'alfabeto binario. ASCII e Unicode sono esempi di alfabeti molto utilizzati nei computer.

Una *stringa su un alfabeto* è una sequenza *finita* di simboli presi dall'alfabeto. Nell'ambito della teoria dei linguaggi i termini "parola" o "frase" sono sinonimi di "stringa". Stringhe generiche verranno nel seguito indicate con le variabili $s, s', s'', \ldots, s_0, s_1, s_2, \ldots$ oppure $x, y, z, w, \ldots, x', x'', \ldots, x_0, x_1, \ldots$ La lunghezza di una stringa s verrà indicata con $|s|$ ed è il numero di simboli che costituiscono la sequenza s. Ad esempio, `banana` è una stringa lunga 6. La *stringa vuota* è denotata con ϵ e ha lunghezza 0. I seguenti sono altri termini usati in questo contesto con le relative definizioni:

- *Prefisso di s* Una stringa ottenuta togliendo zero o più simboli dalla fine di s. Es. `ban` è prefisso di `banana`.

- *Suffisso di s* Una stringa ottenuta togliendo zero o più simboli dalla testa di e. Es. `nana` è suffisso di `banana`.

- *Sottostringa di s* Una stringa ottenuta togliendo da s sia un suo suffisso che un suo prefisso. Es. `ana` è una sottostringa di `banana`. Si noti che ogni prefisso o suffisso di s è anche una sottostringa di s (ma non è vero il contrario). Si noti anche che s stessa è anche suo prefisso, suffisso e sottostringa.

- *Prefisso, suffisso e sottostringa propri di s* Una qualsiasi stringa non vuota x che sia, rispettivamente, un prefisso, un suffisso o una sottostringa di s e tale che $s \neq x$.

- *Sottosequenza di s* Una qualsiasi stringa ottenuta cancellando zero o più caratteri, non necessariamente contigui, da s. Es. `baaa` è una sottosequenza di `banana`.

Il termine *linguaggio* denota un qualsiasi insieme (finito o infinito) di stringhe su un certo alfabeto fissato. Questa definizione è molto generale. Ad esempio l'insieme vuoto ($\emptyset$ o $\{\}$) è un linguaggio (chiamato *linguaggio vuoto*) sotto questa definizione. Un altro linguaggio particolare è il linguaggio $\{\epsilon\}$, che contiene una sola stringa, la stringa vuota. Altri esempi possono essere: tutti i programmi Pascal sintatticamente ben formati o addirittura tutte le frasi in italiano che sono grammaticalmente corrette. Tuttavia c'è subito da notare che quest'ultimo linguaggio è molto difficile da definire precisamente, mentre altri linguaggi non banali possono essere definiti matematicamente (come facciamo noi con i linguaggi di programmazione tramite le grammatiche). Come ultima cosa si noti che la definizione che abbiamo dato non richiede che sia associato nessun significato particolare alle stringhe del linguaggio. La disciplina che si occupa dei metodi per dare significato alle stringhe di un linguaggio viene chiamata generalmente *semantica*.

Richiamiamo alcune operazioni di base sulle stringhe. Se x e y sono due stringhe, allora la *concatenazione* di x ed y, scritta come xy, è una stringa ottenuta attaccando y in fondo a x. Es. $x =$ `buon` e $y =$ `giorno` $xy =$ `buongiorno`. L'elemento neutro per la concatenazione è la stringa vuota: $\epsilon s = s\epsilon = s$ per ogni stringa s. Se pensiamo alla concatenazione come ad un prodotto, possiamo definire l'analogo dell'elevamento a potenza come segue:

- $s^0 = \epsilon$ per ogni stringa s

- $S^i = ss^{i-1}$ per ogni stringa s e per ogni $i > 0$

Ad esempio $\texttt{otto}^0 = \epsilon$, $\texttt{otto}^1 =$ `otto`, $\texttt{otto}^3 =$ `ottoottootto`.

2.4.2 Operazioni sui linguaggi

Ci sono diverse importanti operazioni che possono essere applicate ai linguaggi. Per quanto riguarda l'analisi lessicale ci limiteremo a considerare le seguenti:

- *Unione* $L_1 \cup L_2 = \{s \mid s \in L_1 \vee s \in L_2\}$

- *Concatenazione* $L_1 L_2 = \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$

- *Esponenziazione* $L^0 = \{\epsilon\}$, $L^i = LL^{i-1}$ se $i > 0$
- *Chiusura (o stella) di Kleene* $L^* = \bigcup_{i=0}^{\infty} L^i$
- *Chiusura (o stella) positiva di Kleene* $L^+ = \bigcup_{i=1}^{\infty} L^i$

Esempio 2.2 *Sia* L *l'insieme* $\{A, B, \ldots, Z, a, b, \ldots, z\}$ *e* D *l'insieme* $\{0, 1, \ldots, 9\}$. L *e* D *possono essere visti in due modi: come alfabeti finiti o come linguaggi (finiti) composti da parole che sono tutte lunghe un carattere. Vediamo alcuni linguaggi che possono essere definiti a partire da* L *e* D *tramite le operazioni che abbiamo appena introdotto:*

- $L \cup D$ *è l'insieme delle lettere e delle cifre*
- LD *è l'insieme di tutte le stringhe di lunghezza due formate da una lettera seguita da una cifra*
- L^4 *contiene tutte le stringhe di lunghezza 4 che sono formate da lettere*
- L^* *è un insieme infinito di stringhe ognuna delle quali ha una lunghezza finita ed è composta da lettere. Questo linguaggio contiene, per definizione, anche la stringa vuota* ϵ.
- $L(L \cup D)^*$ *è l'insieme di tutte le stringhe di lunghezza strettamente maggiore di 0, che iniziano con una lettera e sono composte, dopo la prima lettera, da cifre o lettere.*
- D^+ *è l'insieme di tutte le stringhe di cifre formate da almeno una cifra*

2.4.3 Espressioni regolari

In questa sezione definiamo un formalismo che ci permetterà di definire un certo tipo di linguaggi. Ad esempio potremo definire il linguaggio che contiene tutte le stringhe che possono essere viste come identificatori nel modo seguente:

letter (**letter** | **digit**)*

La barra verticale significa "or", le parentesi sono usate per raggruppare sottoespressioni, l'asterisco indica "zero o più occorrenze" dell'espressione fra parentesi e la giustapposizione di **letter** con il resto dell'espressione significa concatenazione.

L'insieme delle espressioni regolari su un certo alfabeto Σ è definito induttivamente dalle regole specificate nel seguito. Ogni espressione regolare su Σ definisce in maniera univoca un linguaggio su Σ. La definizione induttiva specifica (usando l'induzione sulla struttura delle espressioni) anche come viene formato il linguaggio definito da ogni espressione regolare:

Definizione 2.3 (Espressioni Regolari) *Sia Σ un alfabeto finito. L'insieme delle espressioni regolari su Σ è costituito da tutte e sole le espressioni generate induttivamente come segue. Associamo ad ogni espressione anche il linguaggio denotato.*

1. *ϵ è una espressione regolare che denota il linguaggio $\{\epsilon\}$*
2. *Se a è un simbolo di Σ allora a è un'espressione regolare che denota il linguaggio $\{a\}$*
3. *Siano r ed s due espressioni regolari che denotano i linguaggi $L(r)$ ed $L(s)$ rispettivamente. Allora:*
   - *$(r)|(s)$ è una espressione regolare che denota il linguaggio $L(r) \cup L(s)$*
   - *$(r)(s)$ è una espressione regolare che denota il linguaggio $L(r)L(s)$*
   - *$(r)^*$ è una espressione regolare che denota il linguaggio $L(r)^*$*
   - *(r) è una espressione regolare che denota il linguaggio $L(r)$*

Un linguaggio denotato da una espressione regolare viene chiamato insieme regolare.

Per evitare di scrivere troppe parentesi, assegniamo una precedenza ed una regola di associatività agli operatori che costruiscono espressioni regolari così da poter scrivere espressioni con meno parentesi, ma che abbiano un significato univoco. La precedenza è la seguente:

1. L'operatore unario * lega maggiormente
2. La concatenazione è il secondo operatore che lega maggiormente ed è associativa a sinistra (es. $abc = (ab)c$)
3. L'operatore che lega meno di tutti è l'or ($|$) ed è anch'esso associativo a sinistra

Con queste convenzioni, ad esempio, l'espressione $(a)|((b)^*(c))$ può essere scritta come $a|b^*c$. Entrambe denotano il linguaggio delle stringhe che o sono a oppure sono una sequenza, eventualmente vuota, di b seguite da una c.

Esempio 2.4 *Sia $\Sigma = \{a, b\}$.*

- *$a|b$ denota il linguaggio $\{a, b\}$*
- *$(a|b)(a|b)$ denota il linguaggio $\{aa, ab, ba, bb\}$,*
  denotabile anche con $aa|ab|ba|bb$

ASSIOMA	DESCRIZIONE
$r\|s = s\|r$	$\|$ è commutativo
$r\|(s\|t) = (r\|s)\|t$	$\|$ è associativo
$(rs)t = r(st)$	la concatenazione è associativa
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	la concatenazione è distributiva rispetto a $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ è l'elemento neutro per la concatenazione
$r^* = (r\|\epsilon)^*$	relazione tra * e $\|$
$r^{**} = r^*$	* è idempotente

Figura 2.3: Proprietà algebriche delle espressioni regolari

- a^* *denota il linguaggio* $\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
- $(a|b)^*$ *denota il linguaggio di tutte le stringhe formate dalla concatenazione di un numero qualsiasi di* a *e* b *messe in un qualsiasi ordine, più la stringa vuota. Ad esempio* $\epsilon, aab, baababa, a, b, bbbb, aaaa, abab, \ldots$

Se due espressioni regolari denotano lo stesso linguaggio diciamo che sono *equivalenti.*

Le espressioni regolari possono essere viste come un'algebra i cui termini (sui simboli di Σ e su ϵ) sono formati attraverso le tre operazioni. Come tipicamente si fa per un'algebra, diamo, in Figura 2.3, delle equazioni che descrivono alcune proprietà degli operatori.

2.4.4 Definizioni regolari

Per motivi di chiarezza della notazione molto spesso è utile identificare delle espressioni regolari e dar loro un nome. Questo nome può poi essere usato all'interno di altre espressioni regolari negli stessi posti in cui puó essere inserito un simbolo dell'alfabeto. Per ottenere l'espressione regolare senza queste "macro" basta rimpiazzare ogni nome con la relativa espressione regolare. Una serie di definizione di nomi di espressioni regolari si chiama *definizione regolare* ed è una sequenza del tipo:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

dove i d_i $(i = 1, 2, \ldots, n)$ sono tutti nomi distinti e ogni r_i $(i = 2, 3, \ldots, n)$ è una espressione regolare sull'alfabeto $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$. Si noti che per

definire l'espressione per un certo nome d_i è possibile usare i nomi definiti in precedenza.

Per distinguere i nomi definiti dai simboli dell'alfabeto vero e proprio, useremo la convenzione di scrivere i primi in grassetto.

Esempio 2.5 *Scriviamo una definizione regolare per gli identificatori e le costanti numeriche Pascal (queste ultime possono essere intere come ad esempio 343, razionali come 45.56 o con esponente: 6.343E4, 1.4322E-20).*

$$\begin{aligned}
\textbf{letter} &\rightarrow 0|1|\cdots|9 \\
\textbf{digit} &\rightarrow A|B|\cdots|Z|a|b|\cdots|z \\
\textbf{id} &\rightarrow \textbf{letter}(\textbf{letter}|\textbf{digit})^* \\
\textbf{digits} &\rightarrow \textbf{digit digit}^* \\
\textbf{optional_fraction} &\rightarrow .\textbf{digits}|\epsilon \\
\textbf{optional_exponent} &\rightarrow (E(+|-|\epsilon)\textbf{digits})|\epsilon \\
\textbf{num} &\rightarrow \textbf{digits optional_fraction optional_exponent}
\end{aligned}$$

I nomi **id** e **num**, una volta sostituiti ricorsivamente tutti i nomi contenuti all'interno delle loro espressioni regolari associate, corrisponde ad una definizione regolare che denota il linguaggio degli identificatore e delle costanti numeriche Pascal, rispettivamente.

2.5 Automi a stati finiti

In questa sezione definiamo gli automi a stati finiti non deterministici e deterministici. Essi sono dei riconoscitori di stringhe di un certo linguaggio. Vedremo che tutti i linguaggi regolari, cioè quelli denotabili da espressioni regolari, possono essere accettati anche da un automa a stati finiti (è vero anche il viceversa). Dopo la definizione degli automi vedremo due algoritmi importanti: la costruzione dei sottoinsiemi per trasformare un automa non deterministico in deterministico e l'algoritmo per minimizzare il numero degli stati di un automa deterministico.

2.5.1 Automi non deterministici

Definiamo la classe degli automi finiti non deterministici NFA (Non-deteriministic Finite Automata). Sia Σ un alfabeto finito.

Definizione 2.6 (NFA) *Un* NFA *su* Σ *è una tupla* $\langle S, \Sigma, move, s_0, F\rangle$ *dove:*

- S *è un insieme finito di* stati

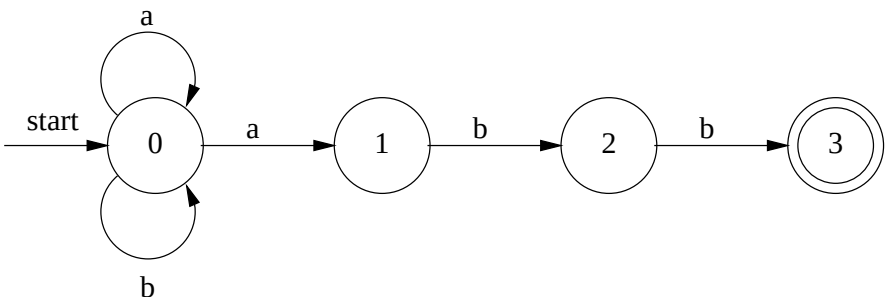


Figura 2.4: Un NFA.

- Σ *è l'alfabeto dei simboli*
- $s_0 \in S$ *è lo* stato iniziale
- $F \subseteq F$ *è l'insieme degli* stati di accettazione *o* stati finali
- *move*: $(S \times (\Sigma \cup \{\epsilon\})) \longrightarrow \wp(S)$ *è una funzione che specifica per ogni stato* s *e per ogni simbolo* x *di* Σ *le* transizioni etichettate x *dallo stato* s *ad un insieme, anche vuoto, di stati di destinazione (c'è una transizione per ogni stato di destinazione).*

Un NFA può essere rappresentato graficamente tramite un diagramma in cui i cerchi sono gli stati, le frecce etichettate sono le transizioni, lo stato iniziale ha una freccia entrante con scritto *start* e gli stati finali hanno una doppia cerchiatura. Ogni freccia può essere etichettata da un simbolo di Σ o da ϵ, la stringa vuota. Le transizioni di quest'ultimo tipo si chiamano ϵ-transizioni.

Un primo esempio di NFA si trova in Figura 2.4. S è l'insieme $\{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, lo stato iniziale è 0, l'unico stato finale è 3 ($F = \{3\}$) e non ci sono ϵ-transizioni. Le frecce rappresentano le transizioni: ad esempio le due frecce etichettate a che partono dallo stato 0 e vanno una nello stato 0 e l'altra nello stato 1 indicano che $move(0, a) = \{0, 1\}$. Inoltre $move(0, b) = \{0\}$, $move(1, a) = \{\}$ eccetera.

Un NFA può riconoscere le stringhe di un certo linguaggio. Per chiarire in quale modo un NFA accetta una stringa definiamo la nozione di cammino etichettato.

Definizione 2.7 (Cammino etichettato) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Un cammino etichettato di lunghezza* $k \geq 0$ *su* N *è una sequenza* $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} s_k$ *dove:*

- s_0 *è lo stato iniziale*

- $\forall i \in \{1, 2, \ldots, k\}.\ s_i \in move(s_{i-1}, x_i)$

La stringa $x_1 x_2 \cdots x_k$ *si chiama* stringa associata al cammino *ed è in generale una stringa di* $\Sigma*$. *Se* $k = 0$ *allora il cammino è costituito solo dallo stato iniziale e la stringa associata è la stringa vuota* ϵ.

Si noti che un cammino di lunghezza k *consiste di una sequenza di* $k+1$ *stati dell'automa ed ha una stringa associata lunga al più* k. *Questo perchè qualcuno (o anche tutti) i simboli* x_i *potrebbero essere* ϵ.

Definizione 2.8 (Stringa accettata) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Una stringa* $\alpha \in \Sigma^*$ *è accettata dall'automa* N **se e solo se** *esiste un cammino etichettato* $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} s_k$ *su* N *tale che* α *è la stringa associata al cammino e lo stato* s_k *è uno stato finale (in formule* $s_k \in \mathcal{F} \wedge \alpha = x_1 x_2 \cdots x_k$*). Se lo stato iniziale è di accettazione allora anche la stringa vuota* ϵ *è accettata dall'automa.*

Possiamo ora introdurre la nozione di linguaggio accettato dall'automa.

Definizione 2.9 (Linguaggio accettato) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Il linguaggio accettato dall'automa è l'insieme*

$$L(N) = \{\alpha \in \Sigma^* \mid \alpha \text{ è accettata da } N\}$$

Esempio 2.10 *Consideriamo l'automa in Figura 2.4.*

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

è un cammino etichettato la cui stringa associata è aabb. Lo stato in cui il cammino termina, 3, è anche uno stato finale e quindi questa stringa è accettata dall'automa.

Essendo l'automa non deterministico, tuttavia, esiste un altro cammino etichettato con la stringa precedente:

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

in questo caso lo stato 0 non è finale e quindi questo cammino non porta all'accettazione della stringa. Si noti che, comunque, la definizione richiede che ci sia almeno *un cammino etichettato che termina in uno stato finale per determinare se la stringa associata è accettata o no.*

Facendo altri esempi e considerando la struttura dell'automa è facile convincersi che il linguaggio accettato è quello denotato da $(a|b)^*abb$.

La costruzione di un cammino etichettato per una data stringa può essere vista come un algoritmo di riconoscimento di stringhe. Ad esempio, supponiamo che vogliamo controllare se la stringa *aba* fa parte del linguaggio

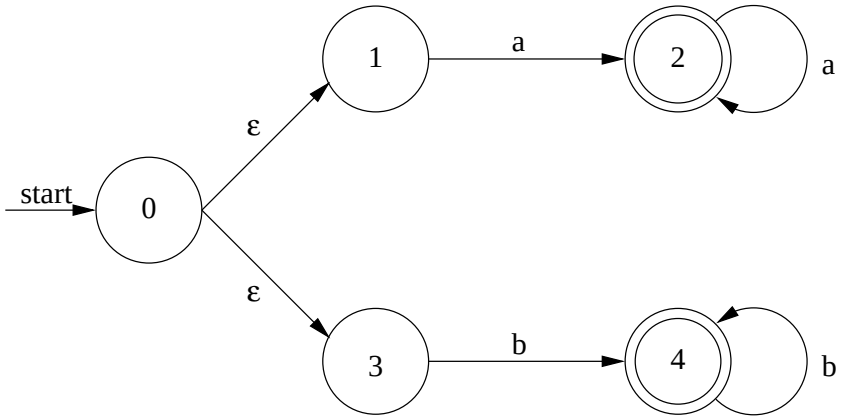


Figura 2.5: Un NFA contenente anche ϵ-transizioni.

accettato dall'automa di Figura 2.5. Partiamo dallo stato iniziale e notiamo subito che, senza considerare nessun simbolo, possiamo effettuare delle ϵ-transizioni. Manteniamo un insieme di stati "correnti" e poniamolo in questo momento a $\{0, 1, 3\}$ dato che questi sono gli stati in cui mi posso trovare cercando di costruire un cammino etichettato per la stringa *aba*.

Consideriamo ora il primo simbolo della stringa: *a*. A questo punto guardiamo quali sono gli stati in cui possiamo andare seguendo una transizione etichettata *a* uscente da uno qualsiasi degli stati correnti. L'insieme degli stati che posso raggiungere con la prima *a* è $\{2\}$ poichè dagli stati 0 e 3 non ci sono transizioni uscenti etichettate con *a*, mentre $move(1, a) = \{2\}$. Notiamo che dallo stato 2 non escono ϵ-transizioni e quindi nel processo di costruzione del cammino per *aba*, a questo stadio, possiamo trovarci solamente nello stato 2 (se ci fossero, dovremmo aggiungere agli stati correnti tutti gli stati raggiungibili con ϵ-transizioni).

Eliminiamo il primo simbolo dalla stringa in esame e consideriamo quello seguente: *b*. A partire da ogni stato in $\{2\}$ controlliamo in quali stati posso andare seguendo transizioni etichettate con *b*. Si ha che $move(2, b) = \{\}$ e quindi non possiamo più andare avanti nel riconoscimento della stringa. In questo caso diciamo che l'automa è *bloccato* e l'algoritmo di riconoscimento della stringa *termina senza accettare*.

Consideriamo invece la stringa *bb*. A partire da uno degli stati in $\{0, 1, 3\}$ posso arrivare, con una *b*, nell'insieme di stati $\{4\}$. Eliminiamo la prima *b* e consideriamo la seguente. A partire dallo stato 4 posso, con una *b*, arrivare nello stato 4 (utilizzando la transizione *self-loop* etichettata con *b* dello stato 4). Il nuovo insieme degli stati correnti è $\{4\}$.

A questo punto abbiamo esaurito la stringa in esame. L'algoritmo termina. Per dire se termina con accettazione o senza, basta controllare, in

generale, se uno degli stati correnti è finale. In questo caso è vero: 4 è uno stato finale e quindi la stringa *bb* è accettata dall'automa.

Un cammino etichettato che porta all'accettazione della stringa è il seguente:

$0 \xrightarrow{\epsilon} 3 \xrightarrow{b} 4 \xrightarrow{b} 4$

Facendo altre prove non è difficile convincersi che l'automa accetta il linguaggio denotato da $a^*|b^*$. Formalizzeremo meglio questo algoritmo di riconoscimento più avanti.

2.5.2 Automi finiti deterministici

La definizione di automa che abbiamo dato è quella più generale, cioè quella di automa non deterministico. Diamo ora una caratterizzazione degli automi deterministici DFA (Deterministic Finite Automata).

Definizione 2.11 (Automa deterministico) *Un automa finito deterministico (DFA) è una tupla* $\langle S, \Sigma, move, s_0, F \rangle$ *dove:*

- S *è un insieme finito di stati.*
- Σ *è un alfabeto finito di simboli.*
- $move: (S \times \Sigma) \longrightarrow \wp(S)$ *è una funzione di transizione tale che per ogni stato* $s \in S$ *e per ogni simbolo* $x \in \Sigma$ *l'insieme degli stati* $move(s, x)$ *o è vuoto oppure contiene un solo stato.*
- $s_0 \in S$ *è lo stato iniziale.*
- $F \subseteq S$ *è l'insieme degli stati finali.*

In altre parole un automa è deterministico se da ogni stato non escono mai due transizioni etichettate con lo stesso simbolo e non ci sono ϵ-transizioni.

La nozione di cammino etichettato e di accettazione per un DFA è analoga a quella di un NFA. Dato che in un DFA non ci sono ϵ-transizioni e, dato uno stato e un simbolo è possibile al più una transizione etichettata con quel simbolo, si ha che per ogni stringa accettata da un DFA esiste un unico cammino etichettato con la stringa e che termina in uno stato finale.

L'algoritmo di riconoscimento di una data stringa è uguale a quello di un NFA, ma in questo caso ad ogni passo l'insieme degli stati correnti contiene uno ed un solo stato.

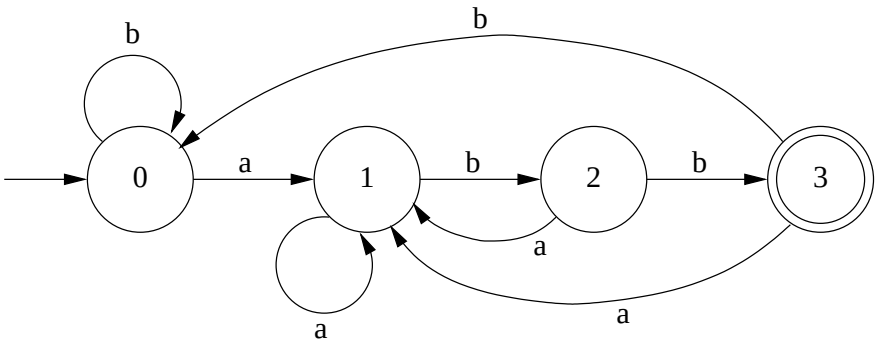


Figura 2.6: Un automa deterministico.

Esempio 2.12 *Consideriamo l'automa disegnato in Figura 2.6. Esso è un automa deterministico che accetta lo stesso linguaggio dell'automa di Figura 2.4, cioè* $(a|b)^*abb$. *L'unico cammino di accettazione per la stringa aabb è*

$0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Il cammino di accettazione per la stringa abaabb è:

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Una funzione di transizione di un NFA o di un DFA può essere rappresentata anche con una tabella in cui le righe sono gli stati e le colonne sono i simboli dell'alfabeto (e anche ϵ nel caso non deterministico). La cella ad una riga T e ad una colonna x della tabella è l'insieme di stati che risulta da $move(T, x)$.

2.5.3 Costruzione dei sottoinsiemi

Gli automi deterministici e quelli non deterministici hanno lo stesso potere espressivo. Questo significa che se un linguaggio può essere accettato da un NFA allora esiste anche un DFA che lo accetta, e viceversa.

E' chiaro che l'algoritmo di riconoscimento di una certa stringa di un DFA è più immediato poichè non bisogna tener traccia di un insieme di stati (il cammino di accettazione, se esiste è unico). D'altra parte, da un punto di vista di progettazione, il non determinismo rende più facile scrivere un automa e/o determinare che tipo di linguaggio accetta, oltre a permettere di scrivere automi più concisi.

Basta ad esempio guardare i due automi delle Figure 2.4 e 2.6 che accettano lo stesso linguaggio. Il primo è non deterministico nello stato 0 e specifica in maniera naturale che si possono riconoscere a o b in qualsiasi numero ed ordine prima di passare ad una stringa finale obbligatoria abb.

Il secondo invece deve esplicitare una sorta di backtracking negli stati: la prima a che incontra potrebbe essere quella della stringa finale obbligatoria e quindi l'automa entra nello stato 1. Se il simbolo seguente non è una b allora l'automa continua a ciclare nello stato 1. Nello stato 2, se il simbolo seguente non è l'ultima b l'automa ritorna nello stato 1 ad aspettare di leggere un'altra a possibile candidata ad essere il primo simbolo della stringa finale obbligatoria. Infine nello stato 3 l'automa deve ritornare nello stato 0 o 1 se ci sono ancora simboli b o a, rispettivamente. Questo perchè i simboli letti precedentemente potrebbero essere il prefisso di una stringa che poi terminerà con la stringa finale obbligatoria.

In questa sezione formalizziamo una variante dell'algoritmo di cui abbiamo già parlato a parole nella Sezione 2.5.1. L'algoritmo che scriveremo è noto come *costruzione dei sottoinsiemi* (subset construction) e serve per costruire, a partire da un NFA dato, un DFA equivalente che simula il non determinismo, ma è deterministico.

Per specificare gli algoritmi, anche nel seguito, useremo uno pseudo-codice in cui le strutture dati vere e proprie non saranno specificate (in ogni caso non è difficile implementare questi algoritmi in un linguaggio di programmazione: basta definire le opportune strutture dati e le varie funzioni/procedure che specifichiamo).

Algoritmo 2.1 (Subset construction) Costruzione di un DFA a partire da un NFA.

Input: Un NFA $N = \langle S, \Sigma, move, s_0, F \rangle$

Output: Un DFA $D = \langle Dstates, \Sigma, Dtran, s'_0, F' \rangle$ equivalente a N e tale che $Dstates \subseteq \wp(S)$.

Metodo: Costruiamo la funzione di transizione *Dtran* simulando, attraverso insiemi di stati di N, i comportamenti di N "in parallelo" dovuti al non determinismo.

L'algoritmo è uno dei classici algoritmi di calcolo di un punto fisso. In pratica partiamo da un insieme di stati *Dstates* contentente solo uno stato iniziale non marcato. Ad ogni passo selezioniamo uno stato non marcato da *Dstates* e ne calcoliamo le transizioni uscenti aggiungendo a *DStates*, non marcati, eventuali nuovi stati trovati. Prima o poi, dato che il numero di stati possibili è finito (il numero di elementi di $\wp(S)$) non ci saranno più stati non marcati da considerare e l'algoritmo terminerà.

Operazioni utilizzate:

- ϵ-*closure* : $S \longrightarrow \wp(S)$ tale che ϵ-*closure*(s) è un insieme composto da s

stesso più eventuali stati raggiungibili, in N, partendo da s e seguendo solo sequenze di ϵ-transizioni.

- ϵ-*closure* : $\wp(S) \longrightarrow \wp(S)$ tale che ϵ-*closure*(T) è un insieme composto dagli elementi di T stesso più eventuali stati raggiungibili, in N, partendo da un qualsiasi stato s di T e seguendo solo sequenze di ϵ-transizioni. In pratica è il lift dell'operazione sugli elementi alla stessa operazione su insiemi.

- *move*: $(\wp(S) \times \Sigma) \longrightarrow \wp(S)$ tale che $move(T, x) = \bigcup_{s \in T} move(s, x)$. In pratica si mettono in uno stesso insieme tutti gli stati raggiungibili con uno stesso simbolo x a partire da uno stato qualunque di T.

Algoritmo:

all'inizio ϵ-*closure*(s_0) è l'unico stato di *DStates* e non è marcato;
while c'è uno stato non marcato T in *DStates* **do begin**
  marca T;
  for each simbolo di input $x \in \Sigma$ **do begin**
    $U := \epsilon$-*closure*$(move(T, x))$;
    if U non è in *DStates* **then**
      aggiungi U, non marcato, a *DStates*;
    $Dtran(T, x) := U$;
  end;
lo stato iniziale di D è ϵ-*closure*(s_0);
gli stati finali di D sono tutti quelli che contengono almeno uno stato finale di N **end**

L'operazione di ϵ-*closure* può essere realizzata semplicemente con l'uso di una pila (stack). In Figura 2.7 è specificato un semplice algoritmo di esplorazione di un grafo in profondità adattato al caso specifico.

Esempio 2.13 *Consideriamo l'automa in Figura 2.8. Il linguaggio accettato dall'automa è quello denotato dall'espressione regolare* $(a|b)^*abb$. *Applichiamo l'algoritmo della costruzione dei sottoinsiemi e troviamo un DFA equivalente.*

Calcoliamo lo stato iniziale: ϵ-*closure*$(0) = \{0, 1, 2, 4, 7\}$. *Chiamiamo questo insieme, per convenienza, A. A è il primo stato non marcato che fa parte di DStates.*

Input: Un NFA N e un insieme T di stati di N

Output: ϵ-*closure*(T)

Algoritmo:

```
metti tutti gli stati di T nella pila;
ε-closure(T) := T;
while la pila non è vuota do begin
    estrai lo stato t dalla testa della pila;
    for each stato u di N tale che t --ε--> u in N do
        if u non è in ε-closure(T) then begin
            aggiungi u ad ε-closure(T);
            inserisci u in testa alla pila;
        end
end
```

Figura 2.7: Algoritmo per il calcolo di ϵ-*closure*(T).

Alla prima iterazione selezioniamo per forza lo stato A e lo marchiamo. Calcoliamo:

- ϵ-*closure*$(move(A, a)) = \epsilon$-*closure*$(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$
- ϵ-*closure*$(move(A, b)) = \epsilon$-*closure*$(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$

Entrambi gli stati B e C sono nuovi (non si trovano attualmente in DStates) e quindi sono stati inseriti non marcati in DStates. La tabella che rappresenta la parte di Dtran calcolata fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B			*No*
C			*No*

Procediamo scegliendo uno stato non marcato. Prendiamo B e calcoliamo:

- ϵ-*closure*$(move(B, a)) = \epsilon$-*closure*$(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$

- ϵ-*closure*$(move(B, b)) = \epsilon$-*closure*$(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\} = D$

Questa volta per il simbolo a non è stato generato nessuno stato nuovo, mentre per b è stato generato il nuovo stato D. La tabella parzialmente costruita fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B	*B*	*D*	*Si*
C			*No*
D			*No*

Continuando ad applicare l'algoritmo si arriva a generare un ulteriore stato $E = \{1, 2, 4, 5, 6, 7, 10\}$ *dopodiché non si generano più nuovi stati e l'algoritmo termina. La tabella finale è la seguente:*

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B	*B*	*D*	*Si*
C	*B*	*C*	*Si*
D	*B*	*E*	*Si*
E	*B*	*C*	*Si*

L'unico stato finale è E poiché è l'unico che contiene lo stato finale di N, cioè 10. L'automa è disegnato in Figura 2.9.

2.5.4 Minimizzazione di un DFA

La facilità di simulazione di un DFA rispetto a quella di un NFA è però compensata dal fatto che il DFA in genere ha un numero maggiore di stati. È lecito chiedersi, in quest'ottica, se è possibile, dato in certo DFA, trovarne uno equivalente, ma che abbia un numero minore di stati. La risposta è si ed esiste un algoritmo che trova un DFA equivalente ad un DFA dato e tale che l'automa risultato ha un numero *minimo* di stati. Qui minimo si riferisce agli stati necessari per poter accettare il linguaggio accettato dall'automa di partenza. Inoltre si ha che l'automa minimo è unico a meno di rinominare gli

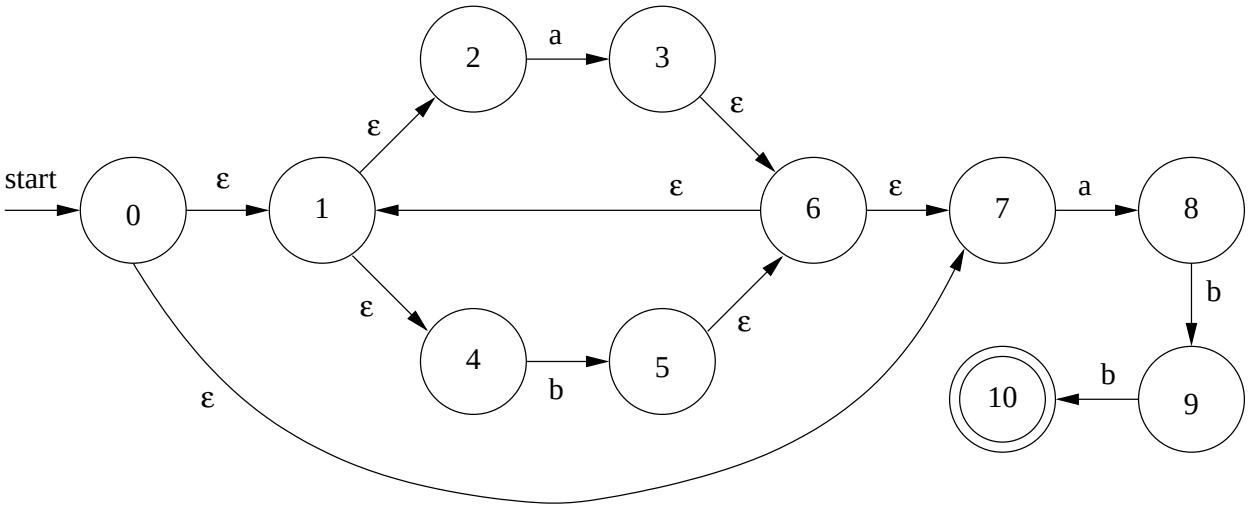


Figura 2.8: Un automa non deterministico per $(a|b)^*abb$.

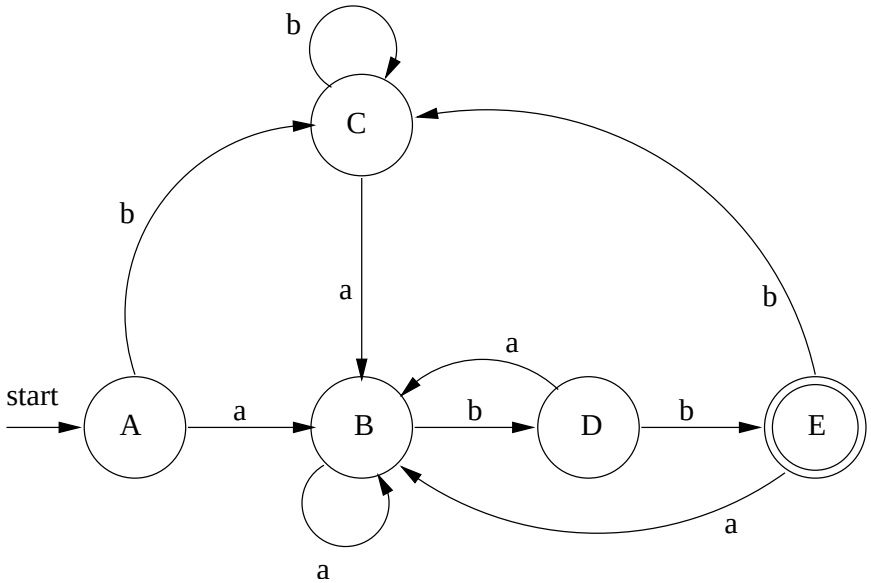


Figura 2.9: L'automa deterministico risultante dalla costruzione dei sottoinsiemi.

stati (in altre parole se trovo due automi minimi esiste sempre una funzione bigettiva fra gli stati dei due che rispetta tutte le transizioni).

L'algoritmo può essere applicato solo a DFA con una funzione di transizione che non restituisce mai l'insieme vuoto. Questa condizione non fa perdere di generalità all'algoritmo poiché è sempre possibile aggiungere un *dead state* ad un DFA nel modo seguente: si crea un nuovo stato d (il dead state appunto) che ha come transizioni uscenti dei self-loop etichettati con ognuno dei simboli di input. Questo significa che se l'automa entrerà nel dead state non ne uscirà più. Ovviamente il dead state *non* deve essere uno stato finale. Ogni volta che in un certo stato T e per un certo simbolo x si ha $move(T, x) = \{\}$ poniamo $move(T, x) := \{d\}$. In questo modo la funzione di transizione non restituisce mai l'insieme vuoto e l'automa ottenuto è equivalente a quello di partenza.

L'algoritmo di minimizzazione cerca di individuare gli stati dell'automa di partenza che si comportano nella stessa maniera e che quindi possono essere raggruppati insieme in un unico stato diminuendo così il numero totale di stati. Fare questo è la stessa cosa che individuare una partizione degli stati dell'automa tale che gli stati in ogni insieme della partizione siano indistinguibili.

Il criterio usato per distinguere gli stati è il seguente: diciamo che una stringa w distingue lo stato s dallo stato t se facendo partire il DFA di partenza dallo stato s e simulando il suo comportamento sulla stringa w arrivo in uno stato di accettazione, mentre facendo partire l'automa dallo stato t e simulandolo su w arrivo in uno stato di non accettazione.

Ad esempio la stringa ϵ distingue gli stati di accettazione da quelli di non accettazione. Ancora, la stringa bb distingue gli stati A e B del DFA in Figura 2.9 poichè partendo dallo stato A, con bb si arriva nello stato C, mentre partendo da B si arriva nello stato E.

Sia $M = \langle S, \Sigma, move, s_0, F \rangle$ un DFA con le caratteristiche richieste. L'algoritmo di minimizzazione è un algoritmo detto di *partition refining*. Esso parte dalla partizione iniziale di S formata dai due sottoinsiemi (chiamiamoli gruppi) F e $S - F$. Sicuramente gli stati di questi due gruppi sono distinti (dalla stringa ϵ). Tuttavia non possiamo ancora affermare che gli stati all'interno di uno stesso gruppo sono indistinguibili perché potrebbe esistere una certa stringa di input che li distingue. Ad ogni passo l'algoritmo raffina la partizione corrente cercando di individuare alcuni stati che si trovano nello stesso gruppo, ma che sono distinti da un simbolo x di Σ.

Sia $\Pi = \{G_1, G_2, \ldots, G_n\}$ la partizione corrente di S. Il passo di distinzione procede come descritto in Figura 2.10

Input: Π partizione di S.

Output: Π_{new} partizione di S.

Metodo:

$\Pi_{\text{new}} := \Pi$;
for each gruppo G_i $(i = 1, 2, \ldots, n)$ di Π **do begin**

partiziona G_i in sottogruppi $\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_k$ tali che:

due stati qualsiasi s e t in G_i sono nello stesso gruppo $\mathbb{G}_j$ $(j = 1, 2, \ldots, k)$ se e solo se:

per tutti i simboli x di Σ si ha che, in M,

$s \xrightarrow{x} s'$, $t \xrightarrow{x} t'$ e s' e t' sono in uno stesso gruppo G_h $(h = 1, 2, \ldots, n)$ di Π;

$\Pi_{\text{new}} := \Pi_{\text{new}}\,[G_i \backslash \mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_k]$;

end

return Π_{new};

Figura 2.10: Passo di raffinamento dei gruppi.

Algoritmo 2.2 (Minimizzazione degli stati di un DFA)

Input: Un DFA $M = \langle S, \Sigma, move, s_0, F \rangle$ con la funzione di transizione che non restituisce mai l'insieme vuoto.

Output: Un DFA M' equivalente che ha un numero minimo di stati.

Metodo:

1. Costruisci una partizione iniziale Π in cui ci sono solo due gruppi: $G_1 = F$ (gli stati finali) e $G_2 = S - F$.

2. Applica la procedura descritta in Figura 2.10 per calcolare la nuova partizione Π_{new}.

3. Se $\Pi_{\text{new}} = \Pi$ allora poni $\Pi_{\text{final}} := \Pi$ e continua con il passo (4). Altrimenti ripeti il passo (2) con $\Pi := \Pi_{\text{new}}$.

4. Costruisci M' seguendo le indicazioni che seguono. Scegli uno stato per ogni gruppo di Π_{final} come rappresentante del gruppo. Gli stati rappresentanti sono gli stati dell'automa minimo M'. Le transizioni di M' sono costruite seguendo il seguente criterio: sia s un rappresentante e supponiamo che in M è presente una transizione $s \xrightarrow{x} t$, sia r il rappresentante del gruppo a cui appartiene lo stato t (r potrebbe essere t stesso). Allora in M' c'è la transizione $s \xrightarrow{x} r$. Lo stato iniziale di M' è il rappresentante del gruppo in cui si trova lo stato iniziale di M. Gli stati finali di M' sono i rappresentanti che sono anche stati finali di M (si noti che ogni gruppo di Π_{final} o è formato da tutti stati finali si M oppure non ne contiene nemmeno uno).

5. Se M' ha un dead state d allora rimuovi d da M'. Rimuovi anche tutti gli stati che non sono raggiungibili da un cammino che parte dallo stato iniziale. Cancella ogni transizione che entrava in d. Restituisci il risultato come automa di output.

La dimostrazione che questo algoritmo produce un automa minimo non la vediamo.

Esempio 2.14 *Minimizziamo il DFA di Figura 2.9. L'algoritmo si può applicare all'automa così com'è poiché in ogni stato ci sono transizioni uscenti per ogni simbolo dell'alfabeto. La prima partizione è formata dal gruppo* (E) *(che è l'unico stato finale) e dal gruppo* (A, B, C, D, E)*. Facciamo il primo passo di raffinamento. Notiamo subito che il primo gruppo non può essere*

raffinato ancora di più poiché è composto da un solo stato e quindi (E) *rimane tale anche in* Π_{new}. *Proviamo quindi a dividere il secondo gruppo e cominciamo con il considerare il simbolo* a *di* $\Sigma = \{a, b\}$. *Si ha:*

- $move(A, a) = B$
- $move(B, a) = B$
- $move(C, a) = B$
- $move(D, a) = B$

Quindi, per quanto concerne il simbolo a *tutti e quattro gli stati possono rimanere nello stesso gruppo. Ma vediamo l'altro simbolo,* b. *Si ha:*

- $move(A, b) = C$
- $move(B, b) = D$
- $move(C, b) = C$
- $move(D, b) = E$

In questo caso si ha che gli stati A, B *e* C *vanno a finire tutti in uno stesso gruppo della partizione corrente* Π *(il gruppo* $(ABCD)$*) mentre lo stato* D *no. Pertanto dobbiamo isolare lo stato* D *e inserire in* Π_{new} *i due nuovi gruppi* (ABC) *e* (D) *al posto del gruppo* $(ABCD)$.

La partizione è cambiata e quindi dobbiamo fare un'altro passo di raffinamento, con la partizione corrente Π *formata dai tre gruppi* (ABC), (D) *ed* (E). *L'unico gruppo che potrebbe raffinarsi è il primo dei tre. Si ha:*

- $move(A, a) = B$
- $move(B, a) = B$
- $move(C, a) = B$

- $move(A, b) = C$
- $move(B, b) = D$
- $move(C, b) = C$

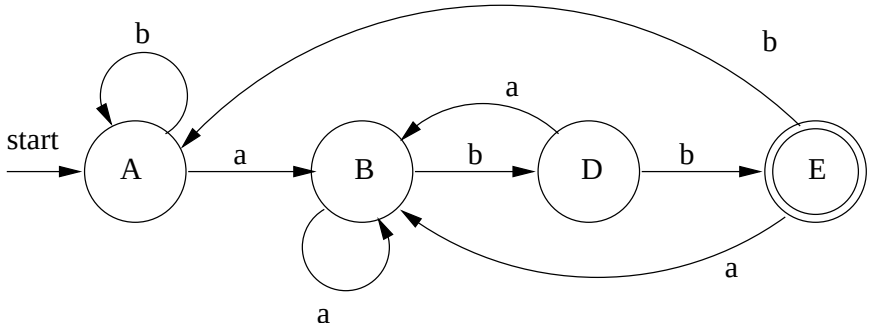


Figura 2.11: L'automa di Figura 2.9 minimizzato.

A questo stadio lo stato D non fa più parte dello stesso gruppo di C nella partizione corrente (D è stato isolato al passo precedente) e quindi siamo costretti ad isolare anche B che si differenzia dagli altri per il simbolo b. Π_{new} *diventa formata da 4 gruppi:* (AC), (B), (D) *ed* (E).

L'ultimo passo di raffinamento proviamo a farlo sul gruppo (AC). *Sappiamo già che il simbolo a non discrimina. Con il simbolo b vediamo che entrambi gli stati del gruppo finiscono nello stesso stato C. Pertanto possono rimanere nello stesso gruppo. In questo passo non sono state fatte modifiche ai gruppi e quindi finiamo l'iterazione.*

Dobbiamo ora scegliere i rappresentanti. L'unica scelta vera è per il gruppo (AC) *(per gli altri il rappresentante può essere solo l'unico stato che li forma). Scegliamo A. L'automa che viene fuori dopo la costruzione dei passi* (4) *e* (5) *dell'algoritmo è disegnato in Figura 2.11. Si noti che è lo stesso automa di Figura 2.6 con i nomi degli stati cambiati.*

2.6 Generatori automatici di analizzatori lessicali

Gli algoritmi che abbiamo studiato nelle sezioni precedenti possono essere combinati insieme in modo da ottenere, a partire dalla definizione di un certo numero di pattern, un riconoscitore dei pattern stessi. Questo processo può avvenire automaticamente ed esistono dei tool software, i generatori di analizzatori lessicali, che prendono una specifica dei pattern e restituiscono un programma che è l'analizzatore lessicale per i pattern dati. Un esempio di generatore di analizzatori lessicali è il programma `Lex`, di cui vedremo un esempio di specifica. `Lex` genera un programma in linguaggio C che riconosce i pattern dati.

In questa sezione schematizzeremo il funzionamento di un generatore di analizzatori lessicali usando gli algoritmi e le costruzioni visti nelle sezioni precedenti. Dopo ciò parleremo un po' più in dettaglio di `Lex`.

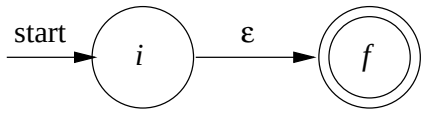
2.6.1 Dalle espressioni regolari agli automi deterministici

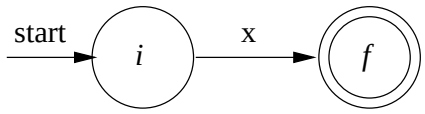
Le espressioni regolari sono il linguaggio più usato per specificare i pattern. Ciò è dovuto alla loro potenza espressiva, al fatto che sono un linguaggio formale, ma soprattutto al fatto che da un insieme di espressioni regolari si può automaticamente generare un automa che riconosce, seguendo un particolare procedimento, i pattern specificati.

Il primo passo della costruzione di un analizzatore lessicale consiste nel trasformare le espressioni regolari che definiscono i vari pattern in automi. Quella che vediamo adesso è una costruzione (detta di Thompson) che, data una qualunque espressione regolare r su un alfabeto Σ, genera un NFA che riconosce il linguaggio denotato da r e ha certe caratteristiche strutturali.

Sia r una espressione regolare su un certo alfabeto Σ. Costruiamo induttivamente un NFA N che riconosca $L(r)$.

Per prima cosa analizziamo la struttura sintattica dell'espressione regolare che risulta dalla sua definizione induttiva (vedi Definizione 2.3). Una volta individuata la struttura (un albero di derivazione) cominciamo a costruire un NFA per ogni simbolo di base, cioè ϵ o un simbolo di Σ (parte 1 e 2 di Definizione 2.3) nel modo seguente:

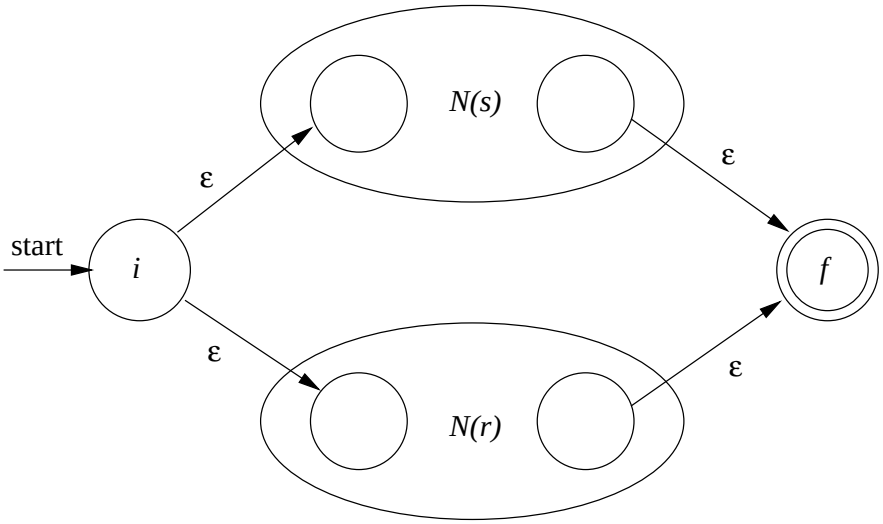




I precedenti sono i mattoni base della nostra costruzione induttiva. Ora, costruiamo induttivamente un NFA per un'espressione regolare qualsiasi partendo dagli NFA associati alle sue sottoespressioni che supponiamo di avere

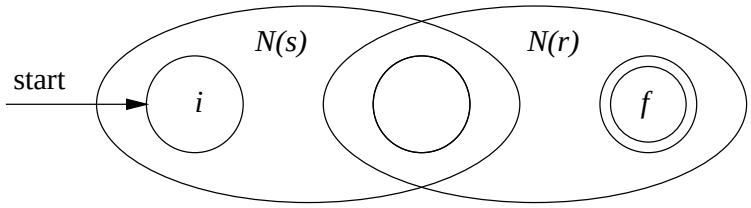
per ipotesi induttiva. Nel far questo usiamo l'accorgimento di costruire gli NFA intermedi in modo tale che: 1) abbiano un solo stato finale, 2) non abbiano nessuna transizione entrante nello stato iniziale e 3) non abbiano nessuna transizione uscente dallo stato finale (si noti che gli automi di base hanno tutte queste caratteristiche). Le seguenti sono le regole per i passi induttivi:

1. Supponiamo, per ipotesi induttiva, di aver ottenuto due NFA $N(s)$ ed $N(r)$ che riconoscono il linguaggio di s e r e con le caratteristiche volute. Un NFA che riconosce il linguaggio $L(s|r)$ e ha le caratteristiche richieste è il seguente



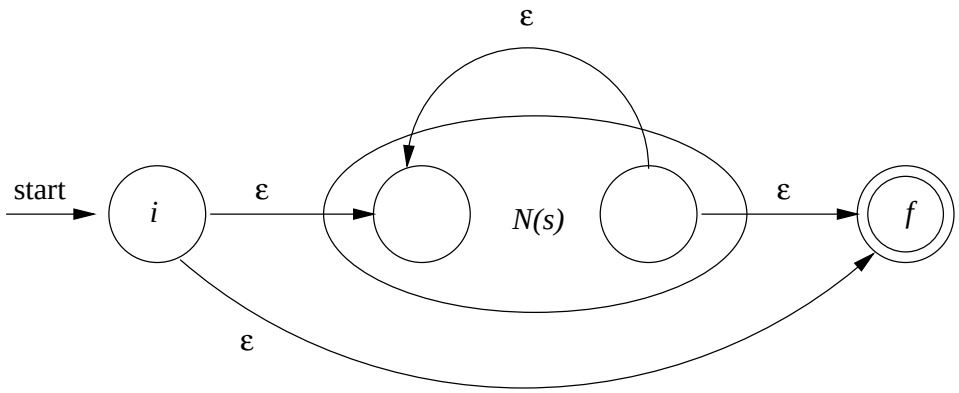
   Nella figura i è un nuovo stato iniziale ed f un nuovo stato finale. Gli ovali rappresentano gli automi $N(s)$ ed $N(r)$ che supponiamo di avere per ipotesi induttiva e i cerchi all'interno degli ovali rappresentano lo stato iniziale (quello a sinistra) e lo stato finale (quello a destra). Si noti che la doppia cerchiatura è stata eliminata dagli stati finali di $N(s)$ ed $N(r)$ e che l'unico stato finale di $N(s|r)$ è f. Stessa cosa è stata fatta per gli stati iniziali. In questo modo il nuovo stato iniziale non ha transizioni entranti e il nuovo stato finale non ha transizioni uscenti.

2. Supponiamo $N(s)$ ed $N(r)$ come sopra. Un NFA con le caratteristiche richieste che accetta $L(sr) = L(s)L(r)$ è il seguente:



Lo stato finale di $N(s)$ viene fatto coincidere con lo stato iniziale di $N(r)$ e lo stato risultante è un semplice stato intermedio, né iniziale né finale, di $N(sr)$. Lo stato iniziale di quest'ultimo è lo stesso di $N(s)$ e lo stato finale è lo stesso di $N(r)$.

3. Sia $N(s)$ un NFA che accetta $L(s)$. Costruiamo, per s^*, il seguente NFA che accetta $L(s)^*$



4. Infine, l'ultimo caso, cioè quello di un'espressione regolare tra parentesi (s), viene trattato semplicemente prendendo lo stesso $N(s)$ come NFA che accetta anche il linguaggio di (s).

Esempio 2.15 *Applichiamo la costruzione appena vista partendo dalla espressione regolare* $r = (a|b)^*abb$. *Innanzitutto individuiamo la struttura sintattica di* r *seguendo le regole di precedenza che abbiamo stabilito in Sezione 2.4.3. L'albero che rappresenta tale struttura è rappresentato in Figura 2.12*

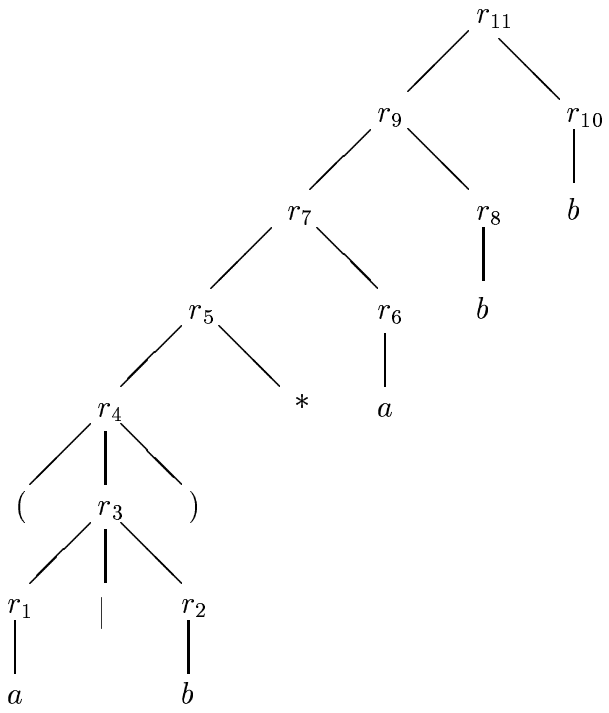
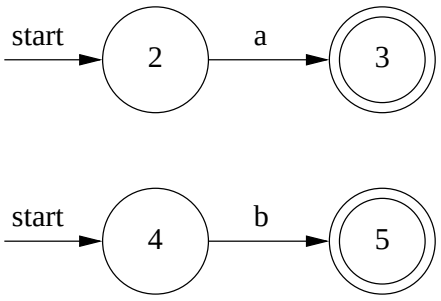


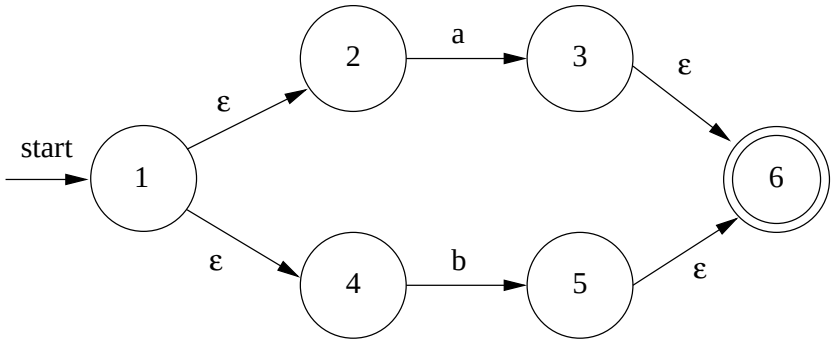
Figura 2.12: Struttura sintattica di $(a|b)^*abb$.

Abbiamo dato un nome ad ogni nodo intermedio e il nodo chiamato r_{11}, la radice dell'albero, corrisponde alla nostra r. Per ogni nodo interno, seguendo la definizione della costruzione di Thompson, costruiamo un NFA che riconosce il linguaggio denotato dalla corrispondente sottoespressione regolare. Per questo processo partiamo dalle foglie dell'albero e procediamo verso l'alto andando a costruire gli automi per le espressioni r_i utilizzando quelli precedentemente costruiti per le loro sottoespressioni (associate ai nodi figli).

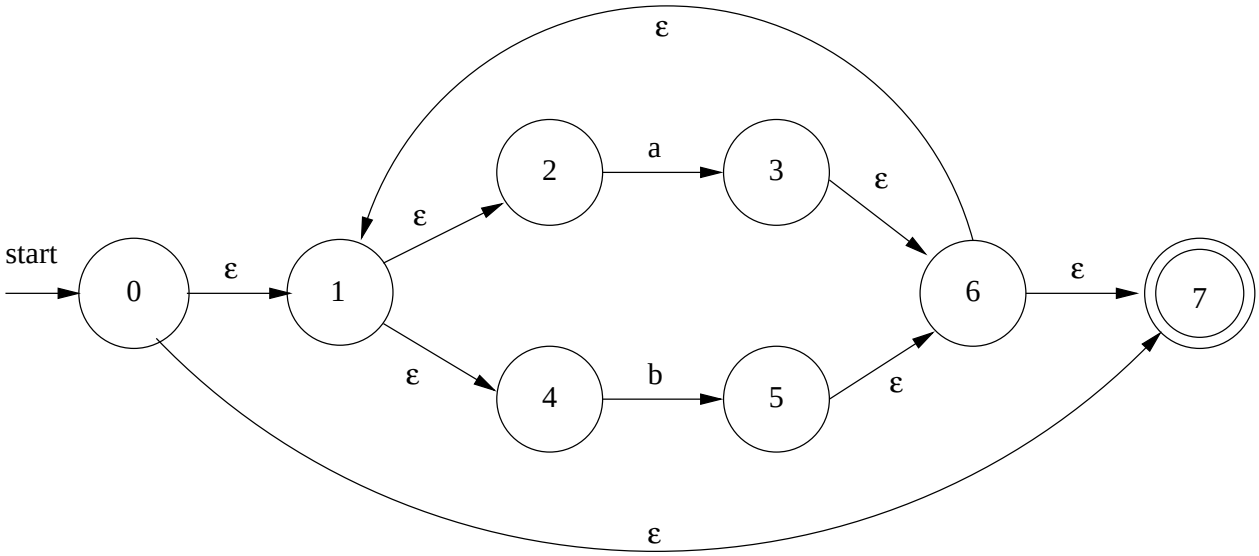
Cominciamo a considerare i nodi r_1 ed r_2. Per questi costruiamo i seguenti automi:



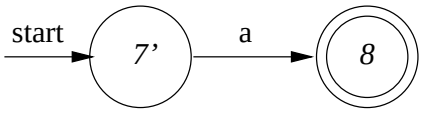
Adesso possiamo combinare $N(r_1)$ *ed* $N(r_2)$ *per ottenere* $N(r_3) = N(r_1|r_2) = N(a|b)$*:*



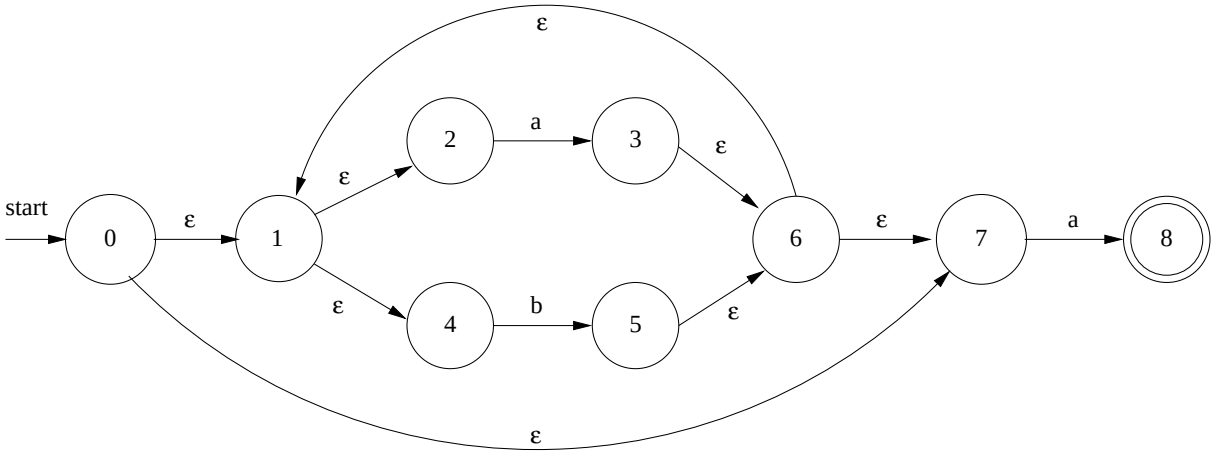
L'NFA per (r_3) *è lo stesso, abbiamo detto, di quello di* r_3*. Quindi andiamo avanti a costruire* $N(r_5) = N((r_3)^*)$*:*



L'NFA per r_6 *è semplicemente:*



L'automa risultante per r_7 è:



Proseguendo così fino alla fine otterremo l'automa in Figura 2.8

2.6.2 Dalle definizioni dei pattern al riconoscimento dei token

Un analizzatore lessicale deve riconoscere i lexeme che si susseguono nel programma sorgente individuando quelli che fanno match con i pattern che sono stati specificati per i token del linguaggio. Questa operazione si chiama *pattern matching* e può essere realizzata in maniera semplice utilizzando le nozioni che abbiamo visto finora.

Poniamoci quindi il seguente problema. Supponiamo data la definizione di un certo numero di pattern $p_1, p_2, \ldots p_n$ dove, per ogni pattern p_i, è specificata anche una sequenza di azioni a_i da effettuare:

$$\begin{array}{ll} p_1 & \{a_1\} \\ p_2 & \{a_2\} \\ \ldots & \\ p_n & \{a_n\} \end{array}$$

I pattern p_i, in generale, sono espressioni regolari. Ovviamente, per comodità, si permetterà di utilizzare all'interno dei pattern anche dei nomi definiti con una definizione regolare. Una sequenza di pattern e di azioni di questo genere è quello che tipicamente si dà in pasto ad un generatore automatico di analizzatori lessicali.

Quello che ci si aspetta è di ottenere un programma che legge una sequenza di caratteri e implementa un ciclo in cui, ad ogni passo, viene individuato nell'input il lexeme *più lungo* che fa match con uno dei pattern p_i.

Nel caso che una stessa sequenza di caratteri faccia match con due pattern contemporaneamente il programma sceglierà il pattern che sta più in alto nella definizione (ad esempio p_2 sta più in alto di p_3). Una volta deciso quale pattern p_i è stato riconosciuto il programma provvederà ad effettuare le operazioni a_i ad esso associate. Fatto questo il programma deve continuare il ciclo cercando un altro pattern a partire dal carattere di input successivo all'ultimo carattere dell'ultimo lexeme riconosciuto.

Per realizzare questo programma bisogna opportunamente utilizzare le costruzioni viste. Innanzitutto si noti che un NFA o un DFA, per definizione, riconosce una sola stringa per volta e non continua, una volta riconosciuta una stringa, a cercare la successiva. Inoltre un automa potrebbe fermarsi al primo stato finale che raggiunge non controllando se, andando avanti, potrebbe riconoscere una stringa più lunga. Per ovviare a questi problemi basterà utilizzare in maniera furba gli automi.

Il primo passo per la costruzione di un algoritmo di pattern matching con le caratteristiche che vogliamo è quello di convertire le espressioni regolari p_i (sostituendo opportunamente eventuali nomi definiti con una definizione regolare) in NFA utilizzando la costruzione di Thompson (Sezione 2.6.1). Otterremo così degli NFA $N(p_1), N(p_2), \ldots N(p_n)$ che riconoscono singolarmente il linguaggio formato da tutti i lexeme di ogni pattern. A questo punto costruiamo l'NFA mostrato in Figura 2.13. Questo NFA ha un nuovo stato iniziale ed esattamente n stati finali, uno per ogni pattern.

Per semplicità, descriviamo a parole la struttura dell'algoritmo di pattern matching basato su un NFA di questo tipo.

All'inizio di ogni iterazione del ciclo più esterno facciamo andare l'automa di Figura 2.13 sull'input attuale (il nondeterminismo può essere simulato semplicemente mantenendo ad ogni istante un insieme di stati attuali, come viene fatto nella costruzione dei sottoinsiemi) fino a quando non arriva ad uno stato finale.

Manteniamo in una variabile `p` un puntatore a questo punto dell'input e in una variabile `npattern` il numero corrispondente alla posizione nella lista, data all'inizio, del pattern che è stato riconosciuto. Per questo basta guardare lo stato finale che abbiamo incontrato (ricordiamo che esiste uno ed

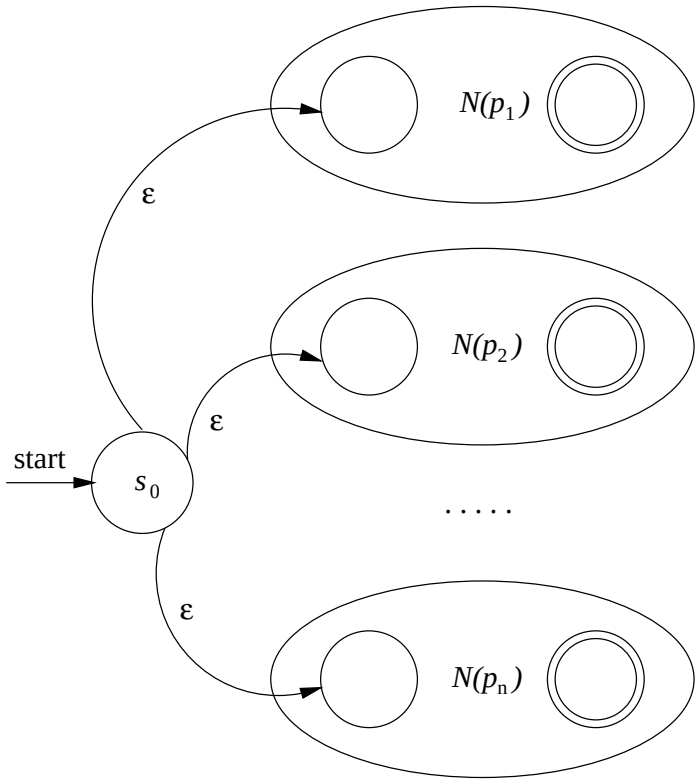


Figura 2.13: NFA usato per il pattern matching.

un solo stato finale per ogni pattern). Se abbiamo incontrato più stati finali scegliamo quello corrispondente al pattern che sta più in alto nella lista.

Facciamo ripartire l'automa dallo stesso insieme di stati in cui si trovava e dal carattere successivo a quello del lexeme individuato. Se l'automa incontra un nuovo pattern (va a finire in uno o più stati finali) aggiorniamo le variabili che mantenevano il puntatore all'input e il pattern riconosciuto con questi ultimi valori trovati.

L'automa deve continuare ad andare avanti fino a che non *termina*, cioè fino a che non può fare nessuna mossa (o perché è finito l'input o perché non può fare più nessuna transizione).

A questo punto il programma annuncia che è stato riconosciuto il pattern corrispondente all'ultimo pattern riconosciuto (che si è mantenuto in `npattern`) e che il lexeme corrispondente è quello che va dal primo carattere di input considerato nell'attuale passo del ciclo fino al carattere puntato da `p`.

Il programma passa poi ad eseguire le azioni associate al pattern riconosciuto (in genere sono delle sequenze di istruzioni scritte nello stesso

linguaggio di programmazione in cui viene generato l'analizzatore lessicale).

Il ciclo ricomincia facendo ripartire l'automa dallo stato iniziale e sull'input il cui primo carattere è quello successivo al lexeme riconosciuto (il carattere successivo a quello puntato da `p`).

Nel caso in cui l'automa termini senza avere riconosciuto nessun pattern, l'analizzatore lessicale deve segnalare, nei modi previsti, una situazione di errore.

Esempio 2.16 *Vediamo un esempio del procedimento appena descritto. Consideriamo la definizione dei seguenti tre pattern (per semplicità di presentazione non associamo azioni):*

$$\begin{array}{ll} a & \{\} \\ abb & \{\} \\ a^*b^+ & \{\} \end{array}$$

Innanzitutto dobbiamo costruire gli NFA relativi ad ognuno e poi metterli tutti insieme in un unico NFA come abbiamo mostrato in Figura 2.13. Per semplicità semplifichiamo l'NFA per il terzo pattern (quello risultante dalla costruzione di Thompson contiene più stati ed ϵ-transizioni) e otteniamo l'automa in Figura 2.14.

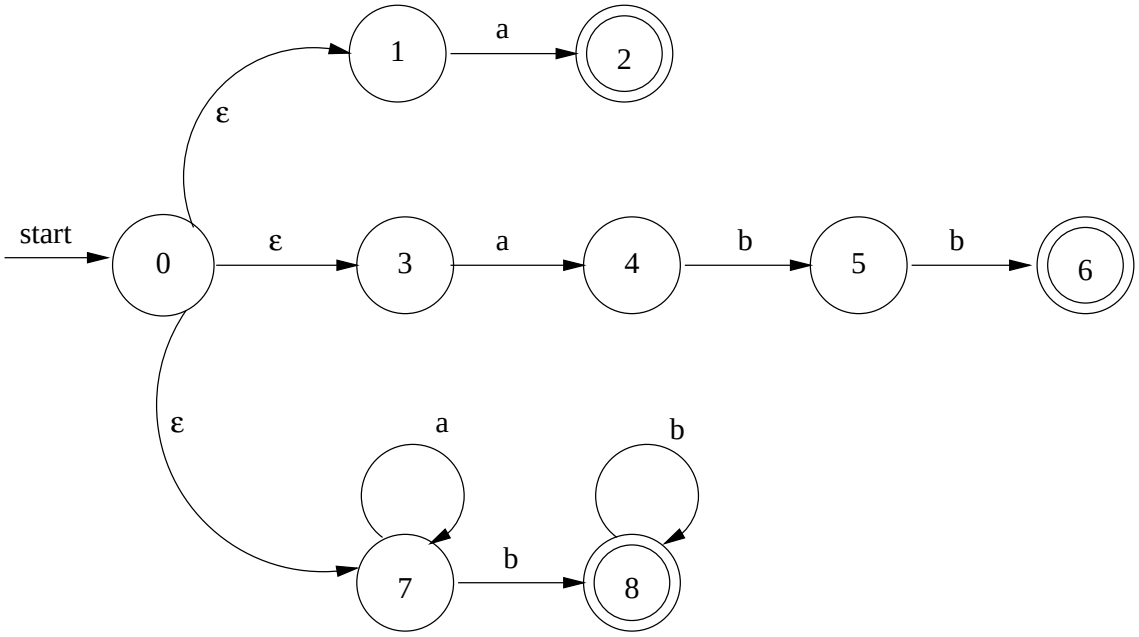


Figura 2.14: Un NFA che per il riconoscimento di tre pattern.

Proviamo ad applicare l'algoritmo di pattern matching descritto alla stringa di input aaba. In Figura 2.15 sono mostrati i momenti salienti della simulazione dell'automa e dell'algoritmo di riconoscimento dei pattern. Lo stato iniziale corrisponde all'insieme di stati ϵ*-closure*$(0) = \{0, 1, 3, 7\}$*. Leggendo una a (la prima dell'input) l'automa transisce nell'insieme di stati* ϵ*-closure*$(move(\{0, 1, 3, 7\}, a)) = \{2, 3, 7\}$ *come mostrato in figura. A questo punto ci accorgiamo che* 2 *è stato finale e che corrisponde al riconoscimento del primo pattern a. Come previsto dall'algoritmo andiamo avanti e memorizziamo questo risultato parziale, cioè assegniamo a* `npattern` *il valore 1 e facciamo puntare* `p` *alla prima a dell'input (in figura questo è indicato dal* p_1 *che si trova dopo la lettura della prima a).*

Dopo la lettura della seconda a dell'input l'automa si trova nell'insieme di stati $\{7\}$*, che non contiene stati finali. Andiamo quindi avanti e leggiamo il terzo simbolo dell'input: b. A questo punto l'automa si trova nell'insieme di stati* $\{8\}$ *che contiene lo stato finale relativo al terzo pattern. Come previsto dall'algoritmo aggiorniamo i nostri risultati parziali e memorizziamo che è stato riconosciuto il pattern* a^*b^+ *al terzo simbolo dell'input. Ma ancora dobbiamo andare avanti fino alla terminazione dell'automa.*

Leggendo il quarto simbolo a l'automa non può fare nessuna mossa perché dallo stato 8 *non ci sono transizioni uscenti etichettate con questo simbolo. Pertanto l'automa termina. L'algoritmo annuncia che il terzo pattern è stato riconosciuto e che il lexeme corrispondente è aab. Se ci fossero azioni associate al terzo pattern, questo sarebbe il momento di eseguirle.*

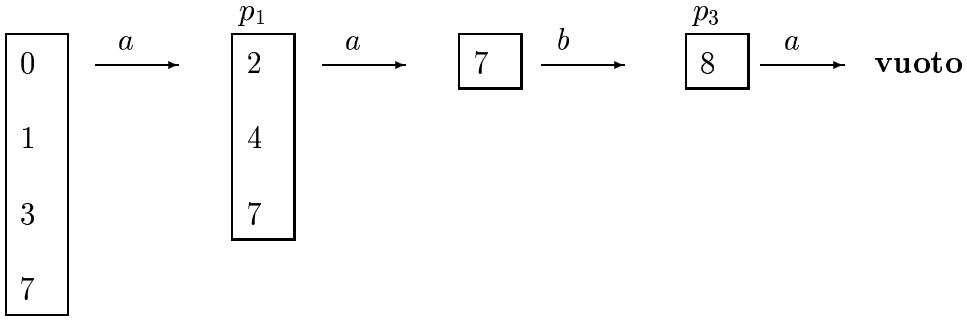


Figura 2.15: Sequenza di insiemi di stati attraversati leggendo l'input *aaba*.

Non essendo finito l'input, l'algoritmo riparte dallo stato iniziale per analizzare i caratteri che seguono il lexeme riconosciuto. In questo caso l'input ancora da analizzare è solo a (l'ultimo simbolo della stringa iniziale). Dallo stato iniziale arriviamo di nuovo all'insieme di stati $\{2, 4, 7\}$ *in cui* 2 *è lo stato finale associato al primo pattern. Memorizziamo questo risultato parziale ed andiamo avanti.*

L'automa termina perché non ci sono più simboli da analizzare e quindi l'algoritmo annuncia il riconoscimento del primo pattern con il lexeme a (che è anche l'unico possibile) dopodiché termina definitivamente.

Notiamo che, come volevamo, l'algoritmo riconosce i pattern che hanno il lexeme più lungo nell'input attuale.

In generale un analizzatore lessicale efficiente non utilizza un automa non deterministico per fare il pattern matching. Tramite l'algoritmo di costruzione dei sottoinsiemi si può ottenere un automa deterministico equivalente a quello usato per il pattern matching con l'algoritmo che abbiamo descritto. Inoltre questo automa può essere ancora migliorato, preservando il linguaggio accettato, con l'algoritmo di minimizzazione introdotto in Sezione 2.5.4.

Notiamo che queste trasformazioni non fanno altro che raggruppare gli stati dell'automa non deterministico di partenza. Pertanto è possibile eseguire lo stesso algoritmo per il pattern matching utilizzando l'automa deterministico minimo. In questo caso si avrà che ad ogni passo l'automa si troverà in un solo stato che però rappresenta un certo insieme di stati dell'automa di partenza. Per quanto riguarda quindi l'individuazione dei pattern attraverso gli stati finali raggiunti basta semplicemente controllare l'insieme di stati che l'unico stato di ogni passo rappresenta e agire come nel caso dell'automa non deterministico che abbiamo visto.

2.6.3 Il generatore automatico `Lex`

In questa sezione parliamo di un particolare generatore di analizzatori lessicali molto diffuso sui sistemi Unix che si chiama `Lex`. `Lex` riceve in input un file di testo la cui struttura analizzeremo in dettaglio e restituisce come output un programma sorgente scritto in linguaggio C che realizza il pattern matching (come lo abbiamo visto nella sezione precedente) dei pattern specificati nel file di input.

In Figura 2.16 sono sintetizzati i passi per creare un analizzatore lessicale scritto in C con `Lex`. Il file di input viene processato e viene generato un programma C che si chiama `lex.yy.c`. Una volta compilato, il programma riceve una sequenza di caratteri sullo standard input e restituisce sullo standard output la sequenza di token trovati.

Di seguito riportiamo un file di input che specifica i pattern di alcuni token del linguaggio Pascal.

```
%{ /* definizione delle costanti numeriche che
      rappresentano i vari token */
```

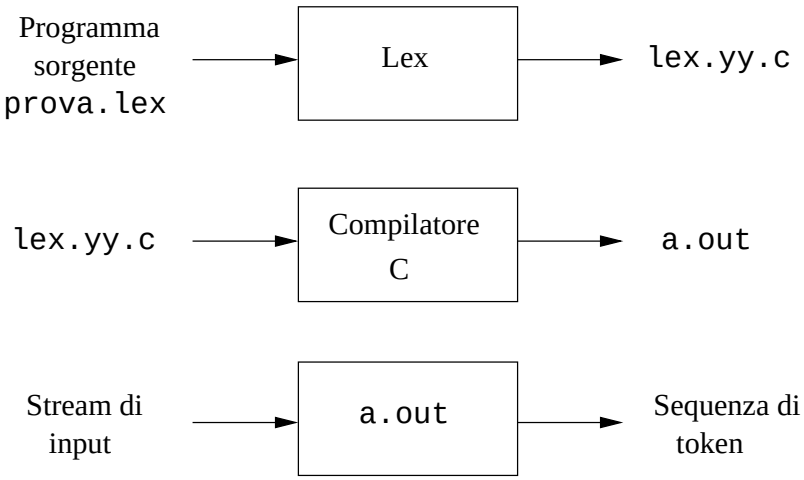


Figura 2.16: Passi per la creazione di un analizzatore lessicale con `Lex`.

```
#define IF 0
#define THEN 1
#define ELSE 2
#define ID 3
#define NUMBER 4

#define RELOP 5 // valore del token
#define LT 50   // valori dell'attributo per il token 5
#define LE 51   // ...
#define EQ 52
#define NE 53
#define GT 54
#define GE 55
%}

%%
/* Definizioni Regolari */
delim         [ \t\n]
ws            {delim}+    //white spaces
letter        [A-Za-z]
digit         [0-9]
id            {letter}({letter}|{digit})*
number        {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
```

```
/* Pattern e azioni associate */

{ws}          {/* Nessuna azione e nessun ritorno */}
if            {return(IF);}
then          {return(THEN);}
else          {return(ELSE);}
{id}          {yylval = install_id(); return(ID);}
{number}      {yylval = install_num(); return(NUMBER);}
''<''         {yylval = LT; return(RELOP);}
''<=''        {yylval = LE; return(RELOP);}
''=''         {yylval = EQ; return(RELOP);}
''<>''        {yylval = NE; return(RELOP);}
''>''         {yylval = GT; return(RELOP);}
''>=''        {yylval = GE; return(RELOP);}

%%
/* Funzioni da definire
   (qui va inserita anche l'implementazione) */
int install_id() {
     /* Installa il lexeme nella tabella dei simboli (se non
        c'e' gia'). Il primo carattere del lexeme e' puntato
        dalla variabile di tipo char* yytext, mentre la
        lunghezza del lexeme e' contenuta nella variabile
        yyleng. La funzione restituisce l'intero che e' il
        puntatore alla riga della tabella dei simboli che
        contiene le informazioni sul token appena installato
     */
}

int install_num () {
    /* Funzione simile, solo che il lexeme che viene installato
       nella tabella dei simboli e' un numero */
}
```

Come si vede, la prima parte di un file di input per `Lex` è racchiusa tra le due sequenze di caratteri %{ e %}. In questa sezione vanno inserite le definizioni di costanti e/o macro personalizzate del programma C che verrà generato. Questa parte di testo verrà copiata senza nessuna modifica all'inizio del programma C che verrà generato. Nell'esempio abbiamo inserito le costanti intere che rappresentano i vari token che vogliamo riconoscere. Si noti che per gli operatori binari di relazione fra interi abbiamo scelto

di creare un'unica categoria lessicale individuata dal token `RELOP`. Sarà poi l'attributo del token, anch'esso un intero, a specificare quale lexeme è stato effettivamente trovato.

Dopo la prima sezione può essere inserita una definizione regolare. `Lex` permette di specificare le espressioni regolari usando diversi operatori oltre a quelli che abbiamo già visto. Tutti questi operatori sono riportati nella seguente tabella[3]:

ESPRESSIONE	DESCRIZIONE	ESEMPIO
c	Un qualsiasi carattere non operatore c	`a`
$\backslash c$	Letteralmente il carattere c	`\*`
"s"	Letteralmente la stringa s	`"**"`
.	Un carattere qualsiasi tranne il *newline*	`a.*b`
^	Inizio della linea	`^ abc`
$	Fine della linea	`abc$`
$[s]$	Un qualsiasi carattere in s	`[abc]`
$[\hat{}\ s]$	Un qualsiasi carattere *non* in s	`[^ abc]`
r^*	Zero o più occorrenze di r	`a*`
r^+	Una o più occorrenze di r	`a+`
$r?$	Zero o una occorrenza di r	`a?`
$r\{m,n\}$	Da m a n occorrenze di r	`a{1,5}`
r_1r_2	Concatenazione	`ab`
$r_1\|r_2$	Or	`a|b`
(r)	Parentesi	`(a|b)`
r_1/r_2	r_1 se seguita da r_2	`abc/123`

Quando all'interno di una espressione regolare vogliamo utilizzare un nome definito precedentemente in una definizione regolare, il nome va messo tra parentesi graffe.

Nell'esempio abbiamo definito gli spazi bianchi, le costanti numeriche e gli identificatori.

Dopo la sequenza di caratteri `%%` inizia la sezione di definizione dei pattern veri e propri con le relative azioni da intraprendere una volta riconosciuti. L'ordine è importante in quanto a parità di lunghezza del lexeme prevale il pattern che viene prima nella lista di questa sezione. Notiamo che se viene riconosciuto il pattern degli spazi bianchi (sequenze di spazi, caratteri di tabulazione (`\t`) o caratteri *newline* (`\n`), non viene intrapresa nessuna

[3]In ogni caso si ha che tutti gli operatori sono riconducibili a quelli base, cioè concatenazione, or e stella di Kleene.

azione. In particolare il programma continuerà a cercare il prossimo pattern senza ritornare al chiamante.

Al pattern `if` abbiamo associato come azioni quella di ritornare al chiamante il token corrispondente alla parola chiave `if`. Notiamo che il chiamante è di solito il parser che richiede un token alla volta. Tuttavia nel caso degli spazi bianchi non è necessario restituire alcun token. Inoltre la parola chiave `if` è messa prima degli identificatori poiché alla visione del lexeme `if`, che fa match anche con il pattern degli identificatori, deve prevalere il pattern che identifica il lexeme come parola chiave.

Per il pattern {`id`} è specificata anche una chiamata di funzione che restituisce un intero assegnato alla variabile `yylval`. Questa variabile è una variabile intera globale del programma che contiene per convenzione, ad ogni nuovo riconoscimento di un token, il valore dell'attributo ad esso associato. Abbiamo visto che nel caso degli identificatori questo valore è spesso un intero che identifica l'entrata nella tabella dei simboli per il lexeme dell'identificatore riconosciuto. Dell'inserimento in questa tabella e della restituzione del valore si occupa la funzione `install_id()`. Tale funzione va scritta nella terza sezione del file di input di `Lex` (quella che segue le successive `%%`). In questo esempio l'implementazione viene lasciata vuota perché bisognerebbe definire anche una struttura dati per la tabella dei simboli con i relativi algoritmi di inserimento, ricerca, aggiornamento. In sede di una reale implementazione di un compilatore questa parte verrà organizzata in accordo con l'implementazione delle altre fasi.

Per il pattern {`number`} è prevista la chiamata ad una funzione analoga alla precedente ma che si dovrà occupare del riconoscimento del tipo di costante numerica (intera o floating point) e di altre cose strettamente relative ai numeri, oltre che all'inserimento del lexeme nella tabella dei simboli.

Per quanto riguarda l'implementazione di queste funzioni è utile sapere che `Lex` mette a disposizione nella variabile globale `yytext` un puntatore a carattere (che in C è lo stesso che dire una stringa) che punta all'inizio del lexeme del pattern corrente riconosciuto. Per completare l'informazione la variabile globale `yyleng` contiene la lunghezza del lexeme cosicchè quest'ultimo possa essere identificato univocamente e trattato adeguatamente.