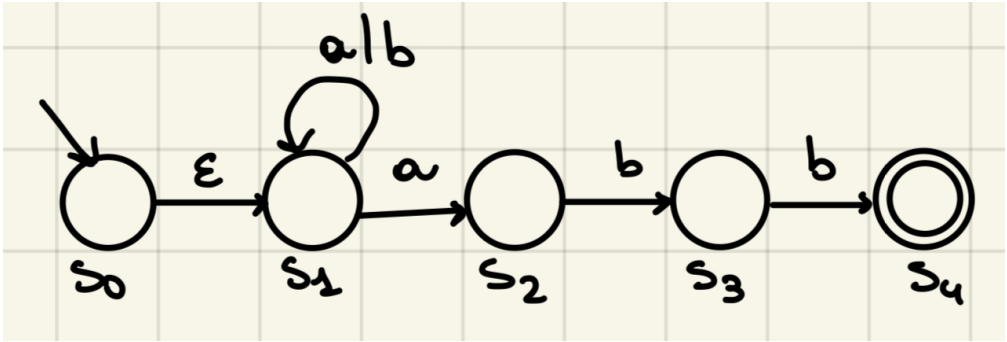
02-analisi-lessicale

RE - NFA - DFA

Nel seguente paragrafo costruiremo un automa a stati finiti deterministico (DFA) per riconoscere un'espressione regolare (RE).
Come prima cosa, partendo da un'espressione regolare, costruiremo un automa a stati finiti non deterministico (NFA). Successivamente dal NFA passeremo all'automa a stati finiti deterministico (DFA) per poi minimizzarlo (renderlo più semplice).

NFA

Costruiamo un NFA per la seguente RE: $(\mathrm{a}|\mathrm{b})^*\mathrm{abb}$



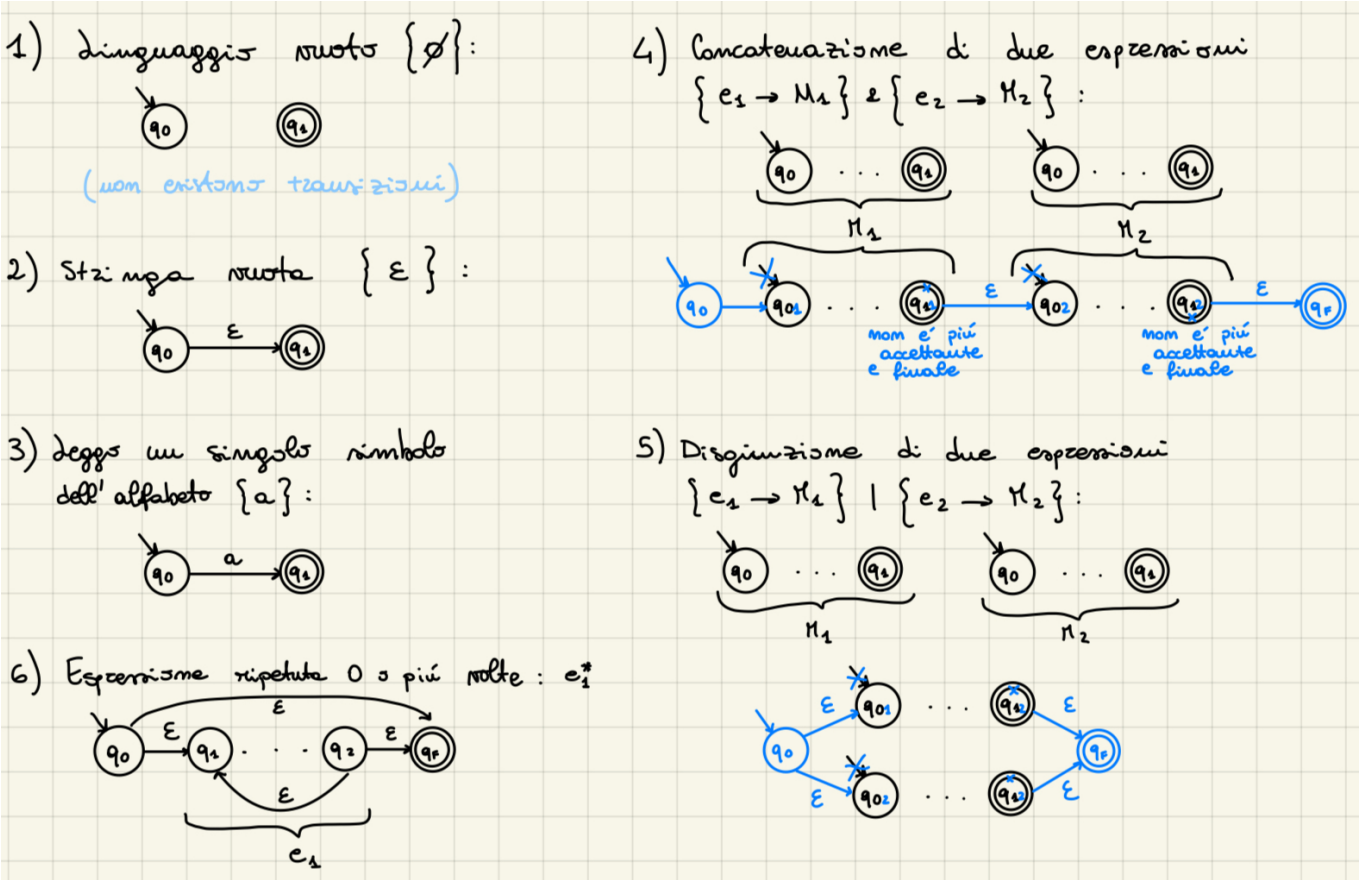
Partendo dallo stato iniziale S_0 e seguendo le frecce possiamo ricostruire la RE precedente, che ci diceva: "possiamo avere una sequenza di zero o più (*) a o b, l'importante è che la stringa si concluda con la sequenza abb".
Dunque se noi dovessimo avere una stringa del tipo "baababbabb" in questo caso il nostro automa sarebbe in grado di riconoscerla e accettarla, analizziamola:

Carattere letto	Stato corrente	Stato successivo
`leggo nulla`	S_0	S_1
b	S_1	S_1
a	S_1	S_1
a	S_1	S_1
b	S_1	S_1
a	S_1	S_1
b	S_1	S_1
a	S_1	S_2
b	S_2	S_3
b	S_3	S_4

Finiamo dunque in uno stato finale accettante S_4 quindi l'espressione è corretta e accettata dall'automa.

> le ε transizioni permettono all'automa di passare da uno stato a un altro **senza leggere alcun simbolo dell'input**. In altre parole, l'automa può "saltare" da uno stato a un altro senza consumare nessun carattere della stringa in input.

Vediamo più nella teoria come si costruisce un NFA, tramite la costruzione di **Thompson**.
Generalizziamo i casi base:

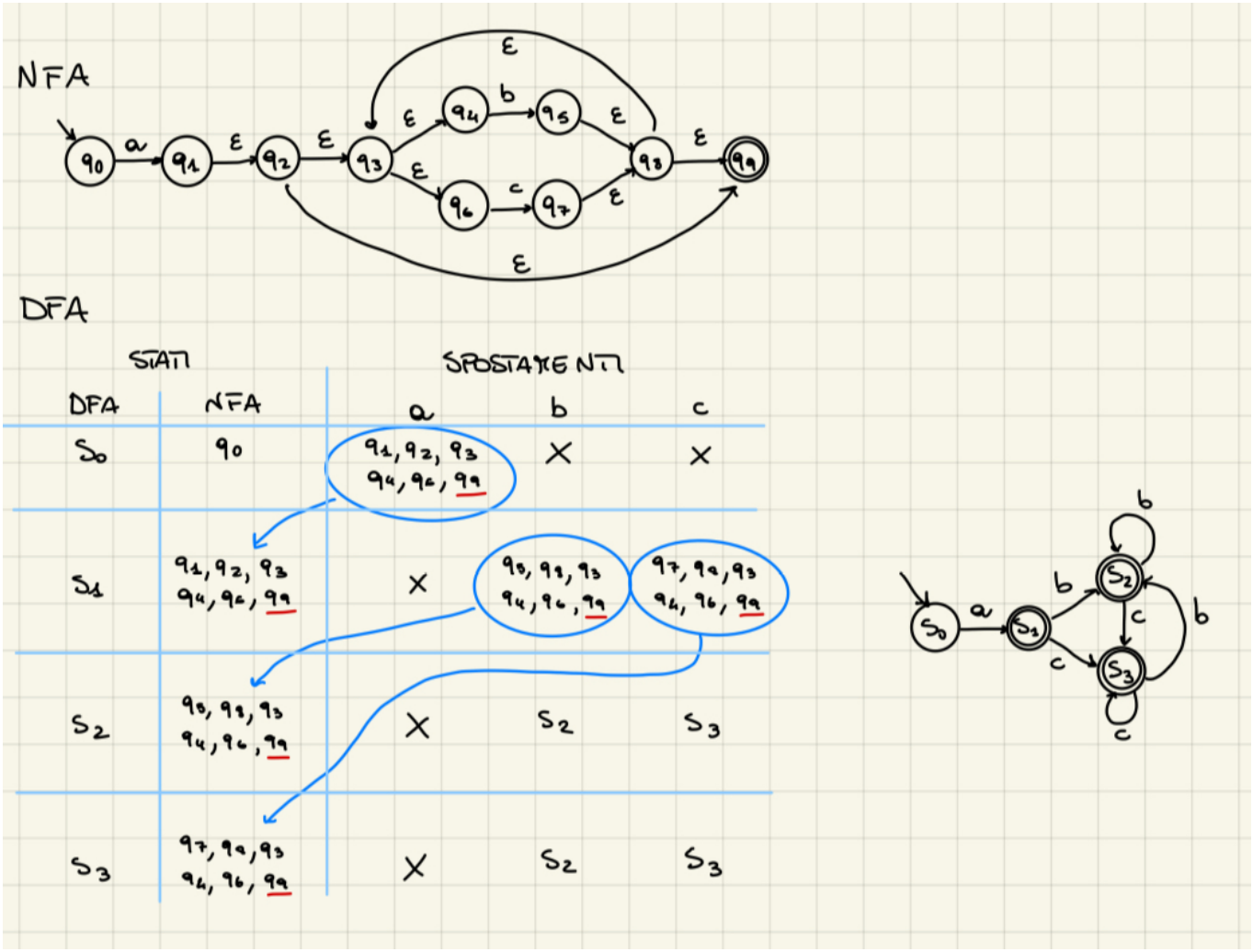


NFA vs DFA

Un automa a stati finiti deterministico è un caso speciale di automa a stati finiti non deterministico, questo perché un DFA non contiene le ε transizioni, di conseguenza questo porta anche al fatto che un DFA sia deterministico e quindi significa che per ogni stato e simbolo dell'alfabeto esiste una sola transizione verso un altro stato specifico (non ci sono ulteriori opzioni), cosa che nel NFA non accade in quanto non è deterministico, quindi possiamo avere più di una transizione o anche nessuna.

Vediamo nell'esempio precedente che quando mi trovo nello stato S_1 e leggo il carattere a, posso avere diverse transizioni in stato differenti, infatti posso continuare a rimanere nello stato S_1 oppure passare allo stato S_2.

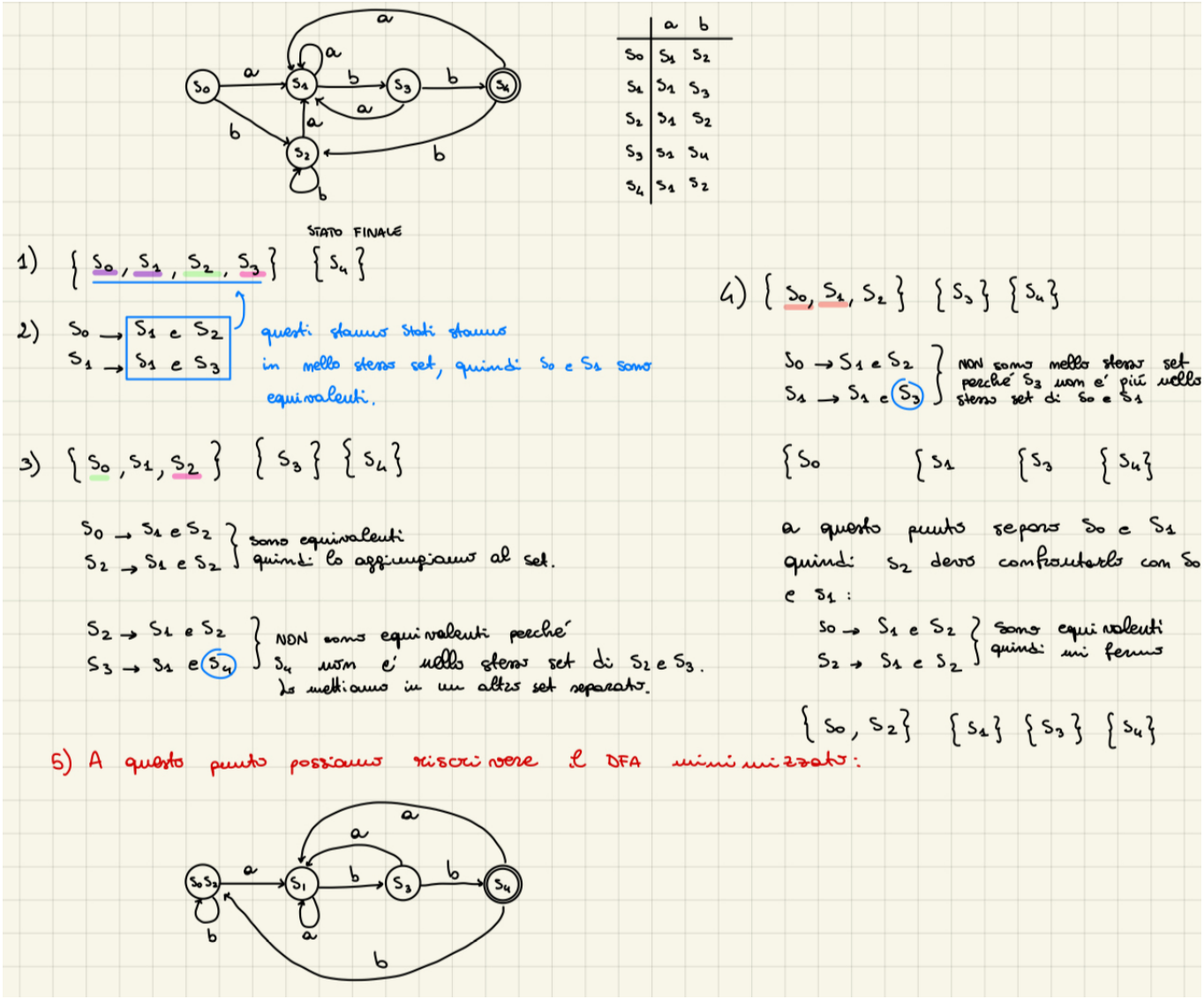
DFA



Vediamo come siamo passati da un NFA a un DFA:

1. il nostro stato S_0 del DFA corrisponde allo stato q_0 dell'NFA, a questo punto, per ogni lettera presente nel nell'NFA guardo quali stati attraverso partendo da q_0 (contando anche le ε), nel nostro caso con la lettera a attraverso gli stati $\{q_1, q_2, q_3, q_4, q_6, q_9\}$, mentre per le lettere b e c non faccio spostamenti in quanto partendo da q_0 devo per forza avere una a per passare agli stati successivi.
2. Il nuovo stato S_1 dunque corrisponderà all'insieme di stati trovati prima, ora da un qualsiasi stato dichiarato con la lettera a non mi sposto in un nessun nuovo stato, mentre con b posso raggiungere nuovi stati $\{q_5, q_8, q_3, q_4, q_6, q_9\}$, mentre con il carattere c raggiungo gli stati $\{q_7, q_8, q_3, q_4, q_6, q_9\}$.
3. Questi stati trovati precedentemente diventano nuovi stati del DFA S_2 e S_3, a questo punto da S_2 con il carattere b incontro nuovamente ancora tutti gli stati di S_2, stessa cosa per c che incontro nuovamente tutti gli stati di S_3.
4. Infine nello stato S_3, si ripete ancora la stessa cosa precedente.
   Da qui costruiamo il DFA.

Ora vediamo come **minimizzare** il DFA:



Un DFA - Automa a Stati Finiti Deterministico è una quintupla M = $\langle \Sigma, Q, \delta, q_0, F \rangle$, dove:

- Σ: alfabeto finito (Σ^* = insieme di tutte le stringhe finite sull'alfabeto Σ)
- Q: insieme finito degli stati dell'automa
- δ: $Q \times \Sigma \rightarrow Q$: funzione di transizione
- $q_0 \in Q$ lo stato iniziale
- F $\subseteq$ Q: sottoinsieme degli stati finali (accettanti)
  Il linguaggio riconosciuto da un DFA è l'insieme delle stringhe che sono **accettate** dall'automa, partendo dallo stato iniziale q_0, sono quelle per le quali la transizione estesa termina in una configurazione finale accettante.

Linguaggi formali:

- Σ: alfabeto finito
- Σ^*: insieme di tutte le stringhe finite sull'alfabeto Σ

- $\epsilon \in \Sigma^*$: indica la stringa vuota
- L $\subseteq \Sigma^*$: linguaggio
  - $\emptyset = \{\}$ è un linguaggio
  - $\{\epsilon\}$ è un linguaggio
  - Σ è un linguaggio
  - Σ^* è un linguaggio

Token

Come vengono classificati i token:

- parole chiavi
- identificatori
- costanti letterali (interi, floating point, stringa, ...)
- operatori (matematici, logici, ...)
- "punteggiatura" (parentesi, virgola, punto e virgola, ...)
- commenti (singola linea, multi linea)

Esempi:

- **Keyword**: if, then, else, while, ... (attenzione al case sensitive).
- **Identificatori**:
  - [a-zA-Z*][0-9a-zA-Z*]⋆
  - [a-zA-Z*]([0-9]* | *[a-zA-Z*])⋆
  - oppure
    - DIGIT = [0-9]
    - LETTER = [a-zA-Z] | [_]
    - LETTER (LETTER | DIGIT)⋆
- **Costanti**:
  - intere: DIGIT+ (accetta 000000 non accetta -1)
  - floating poit: [+-]?[0-9]+.[0-9]
  - carattere: '[^']' (^' = qualsiasi carattere che non sia l'apostrofo)

- **Operatori e punteggiatura**, ogni lessema ha la sua categoria lessicale:

lessema	categoria	lessema	categoria
(	OPEN_PAREN	)	CLOSE_PAREN
[	OPEN_BRACKET	]	CLOSE_BRACKET
{	OPEN_BRACE	}	CLOSE_BRACE
+	PLUS	-	MINUS
+=	PLUS_ASSIGN	-=	MINUS_ASSIGN
:	COLON	::	SCOPE
<	LESS_THAN	<<	SHIFT_LEFT
>	GREATER_THAN	>>	SHIFT_RIGHT
.	DOT	...	ELLIPSIS
...	...		

- **Commenti**:
  - //[^\n]* \n (C++)
  - --[^\n]* \n (SQL)
  - commento multilinea:
    - /*([^*] | * + [^/*])**+/

Flex

Lo strumento flex è un generatore di analizzatori lessicali, che è quindi un compilatore. Viene diviso in 3 sezioni generali:

1 - Sezione delle definizioni

Come prima sezione in un flexer abbiamo quella parte delle definizioni che può contenere:

- Literal Block
- Definizioni di pattern con nome
- Opzioni per flex
- Start states

Literal block

Il literal block è un blocco di codice C racchiuso da `%{ ... %}` a inizio riga, questo viene copiato verbatim (letteralmente così come lo scriviamo) nella parte iniziale del sorgente generato da flex, solitamente contiene:

- definizioni di costanti per categorie lessicali
- dichiarazioni di variabili (usate nelle regole)
- dichiarazioni di funzioni (invocate nelle regole)
- definizioni di funzioni inline

```
enum P_LANGUAGE {
        KEY_W = 1,
        IDENT,
        BOOL,
        INTEGER,
        FLOAT,
        CHAR,
        STRING,
        COMMENT,
};
```

Nell'esempio soprastante ho definito delle costanti che rappresentano i tipi di di token che posso riconoscere e restituire, queste vengono utilizzate successivamente nella sezione delle regole per classificare i pattern.

Pattern con nome

Successivamente al litteral block vengono definiti i pattern con nome, cioè i pattern riutilizzabili che vengono associati a nomi specifici che verranno poi utilizzati nelle regole del lexer.
la sintassi di questa sezione è del tipo: NOME_PATTERN espressione_regolare.

```
DIGIT [0-9]
LETTER [a-zA-Z]
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*
NUMBER {DIGIT}+
WHITESPACE [ \t\n]+
```

Nell'esempio vediamo che il pattern DIGIT rappresenta una cifra da 0 a 9, il pattern LETTER rappresenta una lettera maiuscola o minuscola, il pattern IDENTIFIER rappresenta un identificatore, formato da una lettera iniziale seguita da lettere o cifre, il pattern NUMBER rappresenta una sequenza di una o più cifre consecutive, infine, il patter WHITESPACE rappresenta una sequenza di spazi bianchi (spazio, tabulazione, o nuova riga).

Come specifico i pattern in flex?

- Uso le **virgolette** per simboli non alfanumerici e le **parentesi*
  - ("/*")([^*] | (*"*) +[^*/]) * (("*") + "/") questa espressione viene utilizzata per riconoscere i commenti multi-linea in C.
- Usare nomi di pattern per evitare ripetizioni:
  - STARS ("*")+ `nella sezione delle definizioni`
  - ("/*<")([^*] | {STARS} +[^*/]) * ({STARS} "/")

Opzioni per flex

Sono due quelle da usare sempre:

- `%option noyywrap`: evita la generazione della funzione `yywrap()` e della sua chiamata fine input
- `%option nodefault`: evita la generazione della regola *catch-all (. ECHO*;) che causa la stampa dei token non riconosciuti.
  Due sono quelle da usare quando utile:
- `%option yylineno`: definisce la variabile intera yylineno che mantiene il numero di riga della posizione corrente (la fine del lessema, usarla causa una perdita di efficienza)
- `%option case-insensitive`: rende case-insensitive i pattern, non modifica il file di input (i lessemi riconosciuti rimangono case-sensitive)

Start States

Servono a limitare l'applicabilità di alcune regole, le regole che vediamo successivamente si applicano quando il lexer è nello stato/condition INITIAL, possiamo definire altri stati/condition nella sezione delle definizioni, con la sintassi: `%x NOME_STATO`.
`%x` indica che si tratta di uno stato esclusivo, cioè che il lexer quando entra in quello stato deve uscire dagli altri stati, `%s` definirebbe uno stato shared, consentendo al lexer di essere contemporaneamente in più stati (complicato).

2 - Sezione delle regole

Questa sezione inizia dopo il marker `%%`, serve a fornire la definizione della funzione **int yylex()**, questa deve leggere un lessema dall'input e restituire al chiamante il token corrispondente.
La sezione quindi contiene le regole lessicali per riconoscere i token.

Regole lessicali

Il formato di ogni regola è: *pattern codice*.
Il pattern è un'**espressione regolare** che identifica una specifica sequenza di caratteri (lessema) che il lexer deve riconoscere, questo rappresenta la struttura dei caratteri che corrispondono a un token specifico.
Il codice è un blocco di codice associato al pattern, che viene eseguito quando il pattern viene riconosciuto nell'input, questo specifica cosa fare quando si incontra il lessema: in genere, restituire il token corrispondente o gestire il lessema in modo particolare.

```
bool            { return BOOL; }
if              { return KEY_W; }
```

```
{NUMBER}       { return INTEGER; }
{LETTER}       { return IDENTIFIER; }
{WHITESPACE}   { }
```

Nell'esempio soprastante vediamo:

- il pattern `bool` che riconosce un booleano e il codice `{ return BOOL; }` restituisce il token `BOOL` quando viene riconosciuto il pattern (costante definita nel litteral block).
- il pattern `if` che riconosce un if e il codice `{ return KEY_W; }` restituisce il token `KEY_W` quando viene riconosciuto il pattern (costante definita nel litteral block).
- Pattern `NUMBER` riconosce un numero intero, il codice `{ return INTEGER; }` restituisce il token `INTEGER`.
- Pattern `LETTER` riconosce una stringa, il codice `{ return IDENTIFIER; }` restituisce il token `IDENTIFIER`.
- Infine, il pattern `WHITESPACE` riconosce uno o più spazi, tabulazioni o nuove righe, il codice associato `{ }` indica di ignorare quel tipo di lessema senza restituire alcun token.

Se volessi conoscere il lessema che è stato identificato dal lexer, esso viene contenuto nelle variabili globali **yytext** (puntatore al primo carattere) e la sua lunghezza tramite la variabile **yyleng**.

> Il pattern deve essere specificato ad inizio riga, il codice deve iniziare nella stessa riga del pattern, è possibile andare a capo nel codice se lo si racchiude in un blocco: {codice}.
> E' possibile andare a capo con pattern disgiuntivi usando | al posto del codice, **l'ordine delle regole ne stabilisce la priorità**.

Come specificare i pattern in Flex:

pattern	significato
c	carattere **non** speciale sta per se stesso
c	carattere di escape (per i caratteri speciali)
(*pattern*)	parentesi (per specificare precedenze)
$pattern_1 pattern_2$	concatenazione
$pattern_1 \mid pattern_2$	alternanza
*pattern**	iterazione di Kleene (zero o più occorrenze)
pattern+	iterazione positiva (una o più occorrenze)
pattern?	opzionalità (zero o una occorrenza)
pattern{*m*,*M*}	iterazione limitata
.	qualsiasi carattere *singolo* **tranne newline**
[*chars*]	classe di caratteri (match singolo)
[^*chars*]	complemento di classe di caratteri
"*string*"	match letterale di *string*
{*name*}	uso di pattern tramite nome
$pattern_1 / pattern_2$	trailing context: $pattern_1$ solo se seguito da $pattern_2$
^*pattern*	start-of-line context (se primo carattere del pattern)
pattern$	end-of-line context (se ultimo carattere del pattern)

3- Sezione del codice utente

La sezione del codice utente inizia dopo il secondo marker `%%`, può contenere codice utente arbitrario, inserito verbatim dopo la definizione yylex.
Tipicamente vengono definite delle funzioni ausiliarie precedentemente dichiarate nella sezione delle definizioni e la funzione **main**.

```
int main() {
        int token;
        while(1) {
                token = yylex();
                if (token == 0)
                        break;
                if (token == ERROR)
                        exit(1);
        }
        return 0;
}
```