Lexical analysis

Lexical analysis

- Lexical analysis checks the correctness of program words and transforms a program to the stream of tokens:
  - removes empty symbols and commentaries;
  - identifies keywords, indentifiers and literal constants;
  - constructs a symbol table;
  - finds line/column numbers of symbols;
  - informs about lexical errors when necessary.
- Lexical analysis is also called scanning and the corresponding analyser is called scanner.

Regular expressions

- Regular expressions over (finite) alphabet Σ

$$E ::= \emptyset \mid \varepsilon \mid a \mid (E\ E) \mid (E \mid E) \mid E^\star$$

  where $a \in \Sigma$.

- Regular expression E defines a language $L(E) \subseteq \Sigma^\star$

$$\begin{array}{lcl}
L(\emptyset) & = & \emptyset \\
L(\varepsilon) & = & \{\varepsilon\} \\
L(a) & = & \{a\}
\end{array}
\qquad
\begin{array}{lcl}
L(E_1\ E_2) & = & \{uv \mid u \in L(E_1),\ v \in L(E_2)\} \\
L(E_1 \mid E_2) & = & L(E_1) \cup L(E_2) \\
L(E^\star) & = & \{w^i \mid w \in L(E),\ i \geq 0\}
\end{array}$$

  where $w^0 = \varepsilon$ and $w^{n+1} = ww^n$.

Regular expressions

- Examples:

Regular expression	Defined language
$a \mid b$	$\{a,\ b\}$
$abba$	$\{abba\}$
$ab^\star a$	$\{aa,\ aba,\ abba,\ abbba,\ \ldots\}$
$(ab)^\star$	$\{\varepsilon,\ ab,\ abab,\ ababab,\ \ldots\}$

- To minimize a number of needed parentheses, operators have priorities:
  - the closure operator $(\cdot)^\star$ has highest priority;
  - the choice operator $(\cdot \mid \cdot)$ has lowest priority.

Regular expressions

- A regular description over alphabet Σ is the set of rules

$$\begin{array}{ccc} d_1 & \to & E_1 \\ d_2 & \to & E_2 \\ & \dots & \\ d_n & \to & E_n \end{array}$$

  where d_i is a (unique) name and E_i is a regular expression over alphabet $\Sigma \cup \{d_1, \dots, d_{i-1}\}$.

- Short-hand notation for regular expressions:
  - *nonempty closure*: $E^+ = EE^\star$;
  - *option*: $E? = \varepsilon \mid E$;
  - *character classes*: eg. $[a, b, c] = a \mid b \mid c$ or $[a - z] = a \mid \dots \mid z$.

Regular expressions

Examples of regular descriptions:

Identifiers:

$$\begin{array}{lcl} \text{Letter} & \rightarrow & [a - z,\ A - Z] \\ \text{Digit} & \rightarrow & [0 - 9] \\ \text{Identifier} & \rightarrow & \text{Letter}\,(\text{Letter} \mid \text{Digit})^{\star} \end{array}$$

Numeric constants:

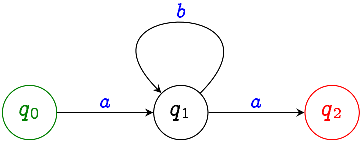
$$\begin{array}{lcl} \text{Sign} & \rightarrow & (+ \mid -)? \\ \text{Integer} & \rightarrow & 0 \mid \text{Sign}\,[1 - 9]\,\text{Digit}^{\star} \\ \text{Decimal} & \rightarrow & \text{Integer}\,.\,\text{Digit}^{+} \\ \text{Real} & \rightarrow & (\text{Integer} \mid \text{Decimal})\ E\ \text{Integer} \end{array}$$

Finite automata

- A finite automaton is the quintuple $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, where
  - Q is a finite set of states;
  - Σ is the finite alphabet;
  - $\delta \subseteq Q \times (\Sigma \cup \varepsilon) \times Q$ is the transition relation;
  - $q_0 \in Q$ is the initial state;
  - $F \subseteq Q$ is a set of final states.
- A finite automaton is deterministic (DFA), if the transition relation is a function $\delta : Q \times \Sigma \to Q$.
- Otherwise, the finite automaton is nondeterministic (NFA).

Finite automata

- Finite automata can be represented by state transition diagrams:



- The finite automaton $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ accepts the language

$$L(A) = \{w \in \Sigma^\star \mid (q_0, w, q_f) \in \delta^\star,\ q_f \in F\}$$

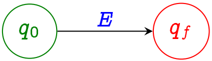
  where $\delta^\star \subseteq Q \times \Sigma^\star \times Q$ is a reflexive and transitive closure of the transition relation δ.

- Theorem: The class of languages accepted by finite automata is that of regular languages.

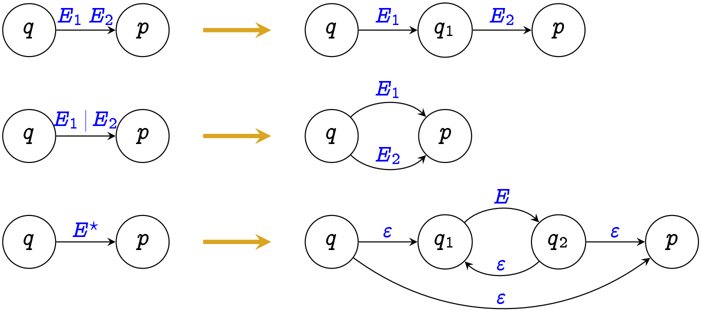
Converting a regular expression to an automaton

Thompson's construction for converting a regular expression to NFA:

- for a regular expression E construct the "automaton":

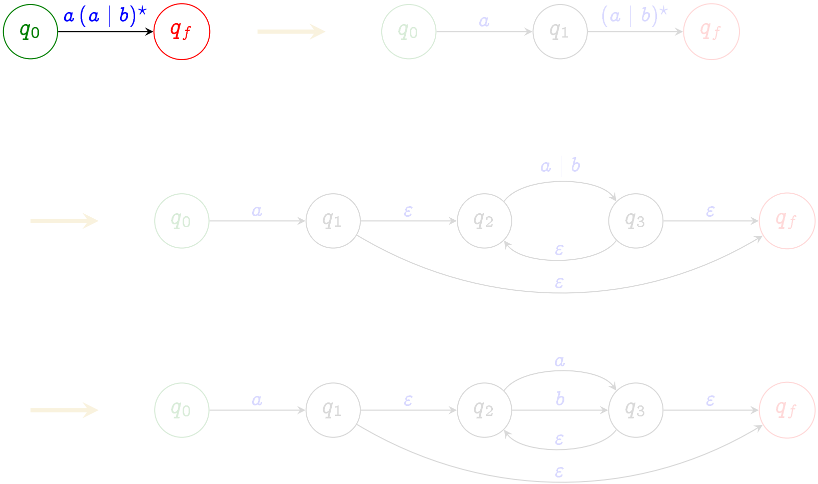


- transform the "automaton" using following rules until all transitions have only simple labels (ie. ε or a character):



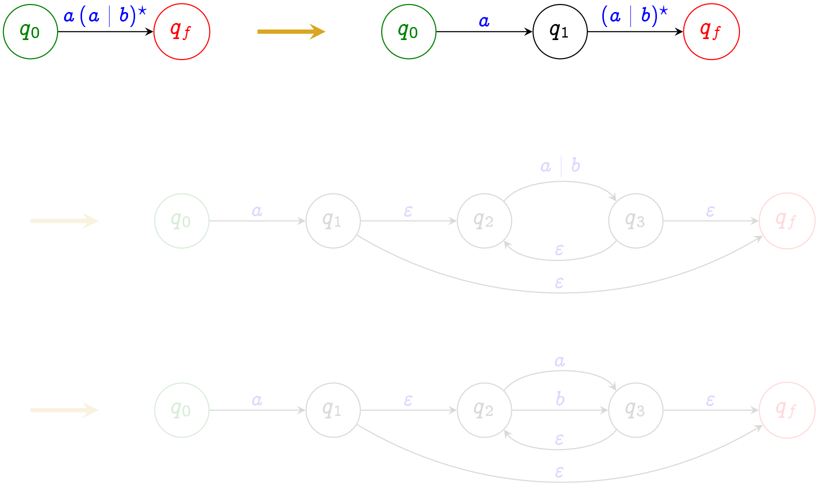
Converting a regular expression to an automaton

Example:



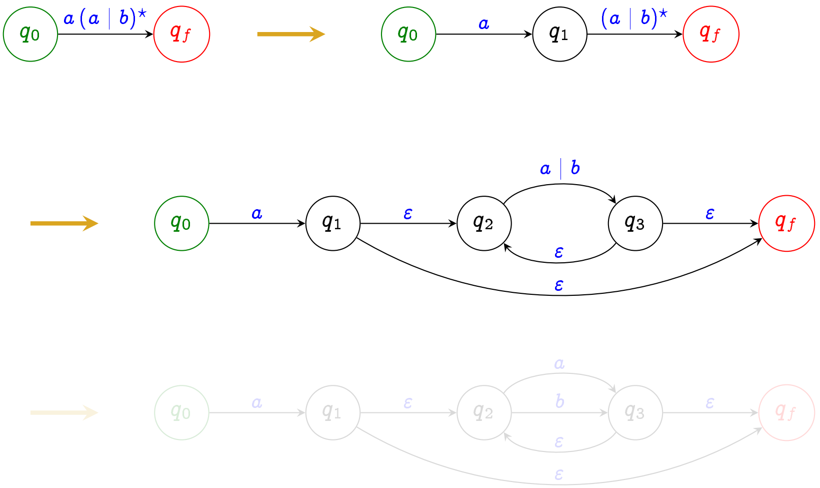
Converting a regular expression to an automaton

Example:



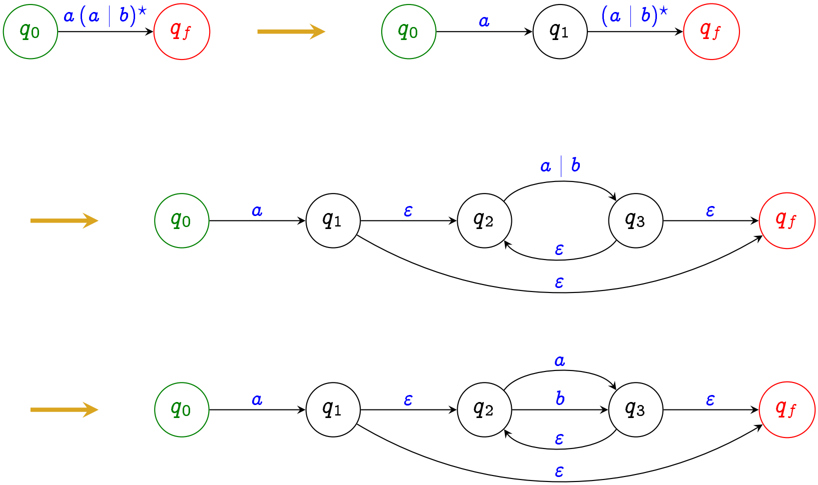
Converting a regular expression to an automaton

Example:



Converting a regular expression to an automaton

Example:



Constructing DFA

- Given NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ construct an equivalent DFA $A' = \langle Q', \Sigma, \delta', q_0', F' \rangle$ by subset construction.
- Auxiliary functions:
  - the ε-closure function $\varepsilon\text{-}closure : 2^Q \to 2^Q$

$$\varepsilon\text{-}closure(S) = \{p \mid q \in S,\ (q, \varepsilon, p) \in \delta^\star\}$$

  - the single step function $move : 2^Q \times \Sigma \to 2^Q$

$$move(S, a) = \{p \mid q \in S,\ (q, a, p) \in \delta\}$$

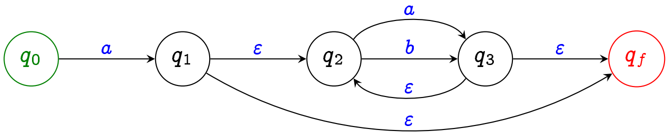
Constructing DFA

Algorithm:

$Q' := \emptyset;\ \ F' := \emptyset;\ \ \delta' := \emptyset;$
$q'_0 := \varepsilon\text{-}closure(\{q_0\});\ \ U := \{q'_0\};$
while $\exists S \in U$ **do**
  $U := U \setminus S;\ \ Q' := Q' \cup \{S\};$
  foreach $a \in \Sigma$ **do**
    $T := \varepsilon\text{-}closure(move(S, a));$
    if $T \notin U \cup Q'$ **then** $U := U \cup \{T\};$
    $\delta' := \delta' \cup \{(S, a) \mapsto T\};$
  end
end
$F' := \{S \in Q' \mid S \cap F \neq \emptyset\};$

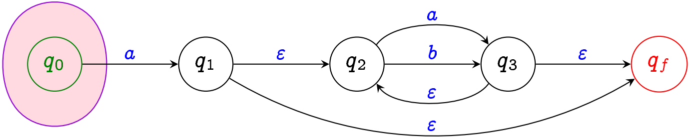
Constructing DFA

Example:



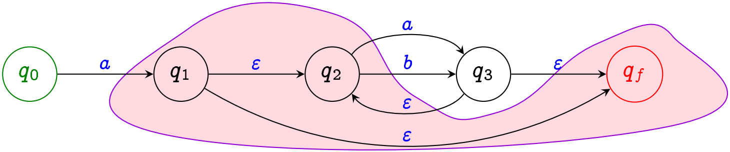
Constructing DFA

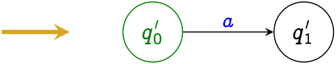
Example:



Constructing DFA

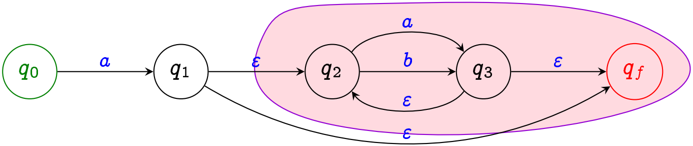
Example:

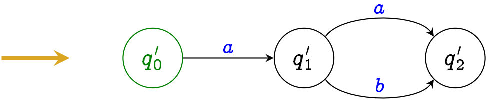




Constructing DFA

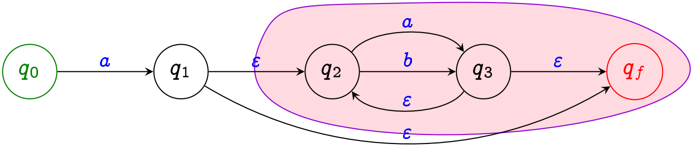
Example:

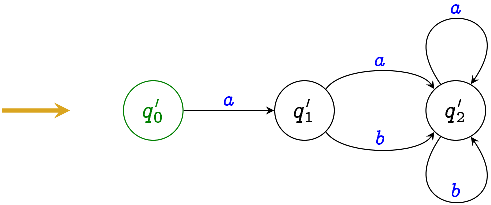




Constructing DFA

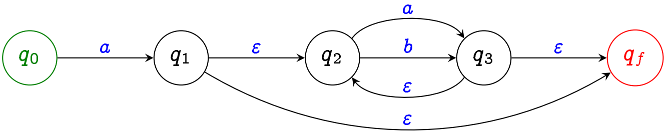
Example:

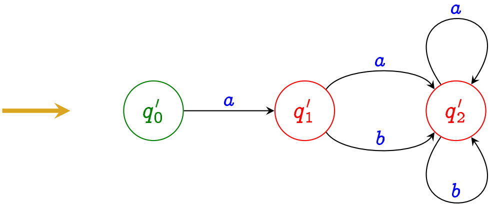




Constructing DFA

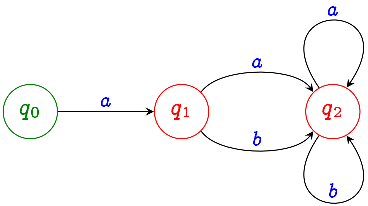
Example:



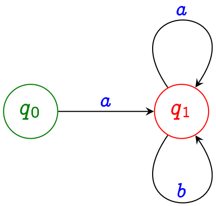


Minimizing DFA

- DFA constructed from the regular expression $a(a \mid b)^{\star}$:



- An equivalent smaller DFA:



Minimizing DFA

- DFA is minimal if there is no smaller DFA accepting the same language.
- For every DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ there exists an (unique) equivalent minimal DFA $A' = \langle Q', \Sigma, \delta', q_0', F' \rangle$.
- Idea: partition the set of states into equivalence classes.
  - States $p, q \in Q$ are equivalent or indistinguishable if automata having these as initial states accept the same language (ie. for any word $w \in \Sigma^\star$ if one succeeds (resp. fails), the other one does the same, and vice versa).
  - For every letter, the transition function transformes equivalent states to equivalent states.

Minimizing DFA

Minimization algorithm:

- Remove all states unreachable from the initial state q_0.
- On the remaining set of states find the biggest partition Π into equivalence classes.
- Construct the new automaton $A' = \langle Q', \Sigma, \delta', q_0', F' \rangle$, where
  - the set of states is $Q' = \Pi$;
  - the initial state is $q_0' = P_0$, where $P_0 \in \Pi$ and $q_0 \in P_0$;
  - the set of final states is $F' = \{P \in \Pi \mid P \cap F \neq \emptyset\}$;
  - the transition function is $\delta' = \{(P_i, a) \mapsto P_j \mid P_j \in move(P_i, a)\}$.

Minimizing DFA

Naive algorithm for finding partition:

$P := \{F,\ Q \setminus F\};$
do $\Pi := P;\ P := \emptyset;$
  foreach $S \in \Pi$ **do**
    foreach $a \in \Sigma$ **do**
      $U := \{T \in \Pi \mid T \cap \mathit{move}(S, a) \neq \emptyset\};$
      $V := \{S \cap \mathit{move}_a^{-1}(T) \mid T \in U\};$
      $P := P \cup V;$
    end
  end
until $\Pi = P;$

Minimizing DFA

- Naive algorithm tries to split all partition at every iteration.
  - In worst case has a quadradic complexity.
  - It is enough to consider only these partitions from which one can move to some split partition.
- Hopcroft's algorithm for finding the partition:
  - uses work-list for non-examined split partitions;
  - if a partition not in the work-list is split, then only one (smaller) subpartition is put to the work-list.

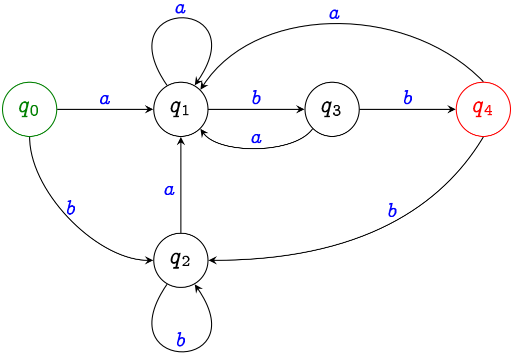
Minimizing DFA

Hopcroft's algorithm:

$\Pi := \{F,\ Q \setminus F\};\ W := \Pi;$
while $\exists S \in W$ **do**
  $W := W \setminus S;$
  foreach $a \in \Sigma$ **do**
    $P := move_a^{-1}(S);$
    foreach $R \in \{T \in \Pi \mid T \cap P \neq \emptyset,\ T \not\subseteq P\}$ **do**
      $R_1 := R \cap P;\ R_2 := R \setminus R_1;$
      $\Pi := (\Pi \setminus R) \cup \{R_1, R_2\};$
      if $R \in W$ **then** $W := (W \setminus R) \cup \{R_1, R_2\};$
      else if $|R_1| \leq |R_2|$ **then** $W := W \cup \{R_1\};$
                    else $W := W \cup \{R_2\};$
    end
  end
end

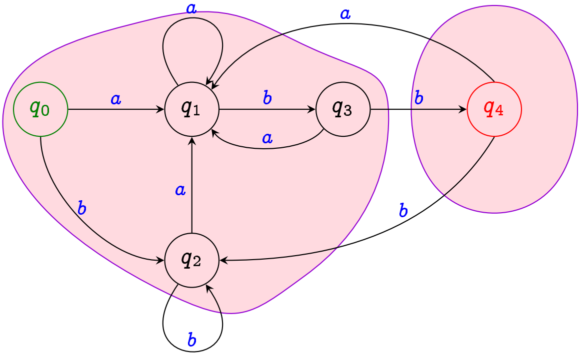
Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^{\star}abb$:



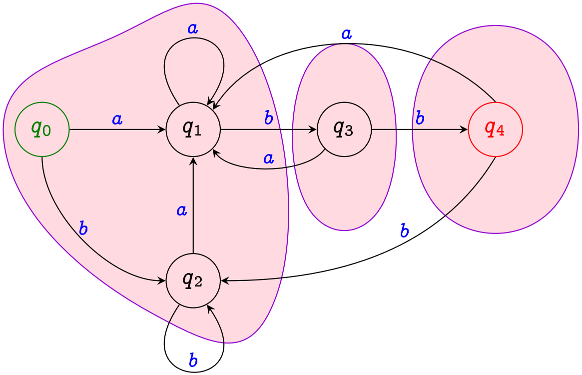
Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^{\star}abb$:



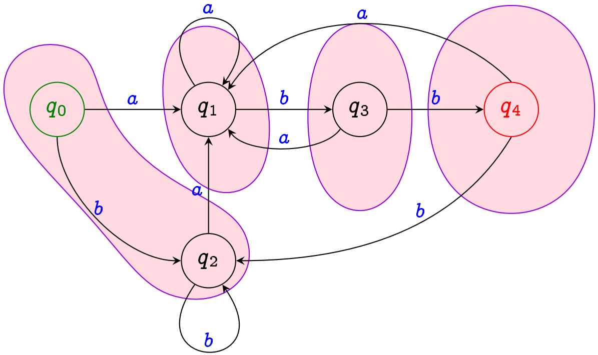
Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^{\star}abb$:



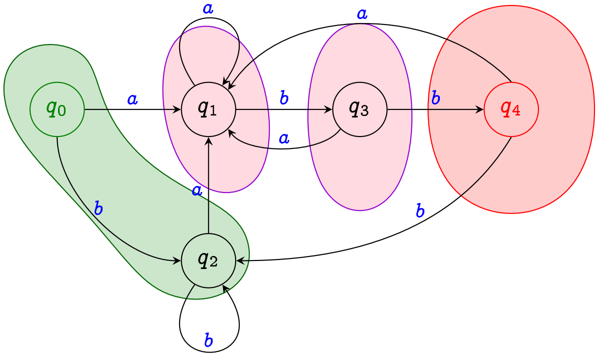
Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^\star abb$:



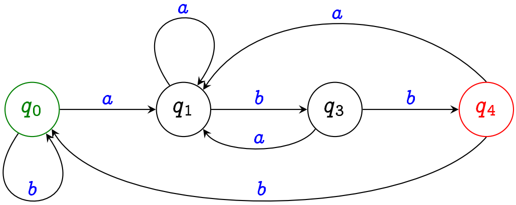
Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^{\star}abb$:

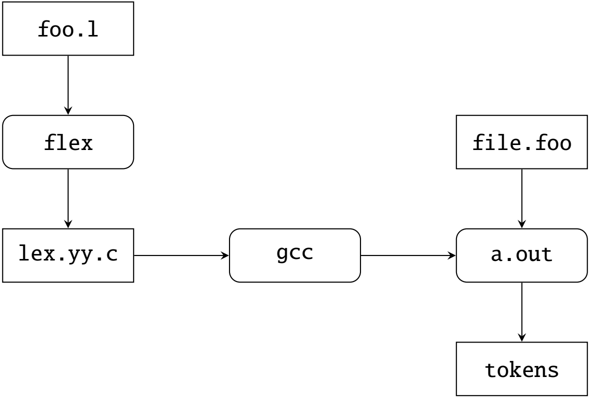


Minimizing DFA

Example – minimizing DFA corresponding to the regular expression $(a \mid b)^{\star}abb$:



Scanner generator Flex



Scanner generator Flex

Format of the input file:

- An input file of Flex has three parts:

```
definitions
%%
rules
%%
user code
```

- The definition part consits of:
  - C code (included header files, definitions of global variables);
  - regular descriptions;
  - definitions of start conditions.

Scanner generator Flex

- The rules part consits of a sequence of pairs:

  ```
  pattern  action
  ```

  where the pattern must start without indentation and ends with the first empty symbol; the action must start on the same line as is the pattern.
- A pattern is a (extended) regular expression; an action is an arbitrary C statement.
  - If action is empty, the input corresponding to the pattern is removed.
  - If input doesn't match with any pattern then it is copied to the output.
- The third part of the Flex input file is a C code which is copied to the generated file `lex.yy.c` in verbatim.
  - May be absent in which case the second separator is also not required.

Scanner generator Flex

Interface for a parser:

`int yylex(void)`	the main function; returns the class of the recognized word; and 0 at EOF
`char *yytext`	points to the last scanned word
`int yyleng`	the length of the last scanned word
`FILE *yyin`	the default input file
`FILE *yyout`	the default output file
`int yywrap(void)`	should be defined in the third part; if not then use '`-lfl`' when linking; usually returns simply 1
`YYSTYPE yylval`	the structure containing a value of the symbol; defined in the parser (in the included header fail `parser.tab.h`)