# Specifications for the GAMBA Library

Erik de Vink and Wieger Wesselink

2019–2020

## 1 Introduction

This document contains pseudocode specifications for the GAMBA Library, see https://github.com/wiegerw/gambatools. It contains support for DFAs, NFAs, PDAs, Turing machines (using the formalism of Sipser), context free grammars and regular expressions.

## 2 Algorithms overview

In this section a list of algorithms is given that can be used as building blocks for Gamba. We assume that $D$ is a DFA, $N$ is an NFA, $r$ is a regular expression, $P$ is a PDA, $T$ is a Turing Machine (TM), $G$ is a context free grammar, $w$ is a string in $\Sigma^*$, *text* is a string and $k$ and $n$ are natural numbers.

| | | |
|---|---|---|
| dfa_accepts_word$(D, w)$ | = | $w \in \mathcal{L}(D)$ |
| dfa_words_up_to_n$(D, n)$ | = | $\{w \in \mathcal{L}(D) \mid \lvert w\rvert \leq n\}$ |
| dfa_minimize$(D)$ | = | $D'$ a minimal DFA with $\mathcal{L}(D') = \mathcal{L}(D)$ |
| dfa_identifiable$(D_1, D_2)$ | = | reachable states of $D_1$ and $D_2$ constitute isomorphic DFAs |
| dfa_to_gnfa$(D)$ | = | $G$ a GNFA with $\mathcal{L}(G) = \mathcal{L}(D)$ |
| dfa_to_regexp$(D)$ | = | $r$ with $\mathcal{L}(r) = \mathcal{L}(D)$ |
| parse_dfa$(text)$ | = | $D$ the DFA corresponding to *text* |
| random_dfa$(\Sigma, n)$ | = | $D$ a random DFA with $n$ states and symbols in $\Sigma$ |

Table 1: DFA algorithms

| | | |
|---|---|---|
| nfa_epsilon_closure$(N, q)$ | $=$ | $\{q' \in Q \mid q \xrightarrow{\varepsilon} q'\}$ |
| nfa_accepts_word$(N, w)$ | $=$ | $w \in \mathcal{L}(N)$ |
| nfa_words_up_to_n$(N, n)$ | $=$ | $\{w \in \mathcal{L}(N) \mid \lvert w \rvert \leq n\}$ |
| nfa_repetition$(N)$ | $=$ | $N'$ with $\mathcal{L}(N') = \mathcal{L}(N)^*$ |
| nfa_concatenation$(N_1, N_2)$ | $=$ | $N$ with $\mathcal{L}(N) = \mathcal{L}(N_1) \circ \mathcal{L}(N_2)$ |
| nfa_union$(N_1, N_2)$ | $=$ | $N$ with $\mathcal{L}(N) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$ |
| nfa_to_dfa$(N)$ | $=$ | $D$ with $\mathcal{L}(D) = \mathcal{L}(N)$ |
| nfa_to_dot$(N)$ | $=$ | $text$ a graphical representation of $N$ in dot format |
| parse_nfa_simple$(text)$ | $=$ | $N$ the NFA corresponding to $text$ |
| random_nfa$(\Sigma, n)$ | $=$ | $N$ a random NFA with $n$ states and symbols in $\Sigma$ |

Table 2: NFA algorithms

| | | |
|---|---|---|
| pda_accepts_word$(P, w)$ | $=$ | $w \in \mathcal{L}(P)$ |
| pda_words_up_to_n$(P, n)$ | $=$ | $\{w \in \mathcal{L}(P) \mid \lvert w \rvert \leq n\}$ |
| pda_to_cfg$(P)$ | $=$ | $G$ with $\mathcal{L}(P) = \mathcal{L}(G)$ |
| pda_to_push_pop$(P)$ | $=$ | $P'$ a PDA in push/pop format with $\mathcal{L}(P') = \mathcal{L}(P)$ |
| parse_pda_simple$(text)$ | $=$ | $P$ the PDA corresponding to $text$ |

Table 3: PDA algorithms

| | | |
|---|---|---|
| tm_accepts_word$(T, w)$ | $=$ | $w \in \mathcal{L}(T)$ |
| tm_words_up_to_n$(T, n)$ | $=$ | $\{w \in \mathcal{L}(T) \mid \lvert w \rvert \leq n\}$ |
| parse_tm_simple$(text)$ | $=$ | $T$ the TM corresponding to $text$ |

Table 4: TM algorithms

| | | |
|---|---|---|
| gnfa_minimize$(G)$ | $=$ | $G'$ a minimal GNFA with $\mathcal{L}(G') = \mathcal{L}(G)$ |

Table 5: GNFA algorithms

| | | |
|---|---|---|
| cfg_accepts_word$(G, w)$ | $=$ | $w \in \mathcal{L}(G)$ |
| cfg_words_up_to_n$(G, n)$ | $=$ | $\{w \in \mathcal{L}(G) \mid \lvert w \rvert \leq n\}$ |
| cfg_eliminate_epsilon_rules$(G)$ | $=$ | $G'$ with epsilon rules eliminated and $\mathcal{L}(G') = \mathcal{L}(G)$ |
| cfg_eliminate_unit_rules$(G)$ | $=$ | $G'$ with unit rules eliminated and $\mathcal{L}(G') = \mathcal{L}(G)$ |
| cfg_to_dfa$(G)$ | $=$ | $D$ with $\mathcal{L}(D) = \mathcal{L}(G)$ |
| cfg_to_nfa$(G)$ | $=$ | $N$ with $\mathcal{L}(N) = \mathcal{L}(G)$ |
| cfg_to_chomsky$(G)$ | $=$ | $G'$ in Chomsky normal form with $\mathcal{L}(G') = \mathcal{L}(G)$ |
| parse_cfg$(text)$ | $=$ | $G$ the CFG corresponding to $text$ |
| parse_cfg_simple$(text)$ | $=$ | $G$ the CFG corresponding to $text$ |

Table 6: CFG algorithms

| | | |
|---|---|---|
| regexp_accepts_word$(r, w)$ | $=$ | $w \in \mathcal{L}(r)$ |
| regexp_words_up_to_n$(r, n)$ | $=$ | $\{w \in \mathcal{L}(r) \mid \lvert w \rvert \leq n\}$ |
| regexp_simplify$(r)$ | $=$ | $r'$ a simplified version of $r$ with $\mathcal{L}(r') = \mathcal{L}(r)$ |
| regexp_to_nfa$(r)$ | $=$ | $N$ with $\mathcal{L}(N) = \mathcal{L}(r)$ |
| parse_regexp$(text)$ | $=$ | $r$ the regular expression corresponding to $text$ |
| parse_regexp_simple$(text)$ | $=$ | $r$ the regular expression corresponding to $text$ |
| random_regexp$(\Sigma, n)$ | $=$ | $r$ a random regexp of size $n$ and symbols in $\Sigma$ |

Table 7: regexp algorithms

# 3 DFA algorithms

**Algorithm 1** Test if a DFA accepts a given word

**Input**: $D = (Q, \Sigma, \delta, q_0, F)$: a DFA; $w \in \Sigma^*$ a word
**Output:** $w \in \mathcal{L}(D)$

dfa_accepts_word$(D, w)$:

$q := q_0$
**for** $a \in w$ **do**
    $q := \delta(q, a)$
**return** $q \in F$

**Algorithm 2** Generate accepted words in a DFA up to a given length $n$

**Input**: $D = (Q, \Sigma, \delta, q_0, F)$: a DFA, $n$: a natural number
**Output:** $\{w \in \mathcal{L}(D) \mid |w| \leq n\}$

dfa_words_up_to_n$(D, n)$

$words := \emptyset$
**if** $q_0 \in F$ **then**
    $words := words \cup \{\varepsilon\}$
$W := \{(q_0, \varepsilon)\}$
**for** $i \in [0 \ldots n)$ **do**
    $W' := \emptyset$
    **for** $(q, word) \in W$ **do**
        **for** $a \in \Sigma$ **do**
            $q' := \delta(q, a)$
            $word' := word +\!\!+\, a$
            $W' := W' \cup \{(q', word')\}$
            **if** $q' \in F$ **then**
                $words := words \cup \{word'\}$
    $W := W'$
**return** $words$

### 3.1 Converting a DFA into a language equivalent regular expression

To convert a DFA into a language equivalent regular expression we apply the method proposed in the book of Sipser, that makes use of the concept of a generalized NFA, also referred to as a GNFA. See Lemma 1.60, pages 69 to 76, in the book of Sipser.

The notion of a GNFA is a generalization of that of a DFA. In the approach of Sipser a GNFA $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ has a set of states $Q$ with two different designated states $q_{start}$ and $q_{accept}$, and a transition function $\delta : Q \backslash \{q_{accept}\} \times Q \backslash \{q_{start}\} \rightarrow RE_\Sigma$. Thus, $q_{start}$ has no incoming transitions, $q_{accept}$ has no outgoing transitions, while transitions are labelled with regular expressions over the alphabet $\Sigma$ of the GNFA.

As usual, the class $RE_\Sigma$ of regular expressions over an alphabet $\Sigma$ is the least set containing $\mathbf{0}$, $\mathbf{1}$, $\mathbf{a}$ for each $a \in \Sigma$ and that is closed under sum $R_1 + R_2$, concatenation $R_1 \cdot R_2$, and the Kleene star operation, also called iteration, $R^*$, for $R_1, R_2, R \in RE_\Sigma$. Thus,

$$R ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{a} \mid R + R \mid R \cdot R \mid R^*$$

for $a \in \Sigma$.

**Algorithm 3** Representing a DFA as a GNFA

**Input** $D = (Q, \Sigma, \delta, q_0, F)$ a DFA
**Output** $G = (Q', \Sigma, \delta', q_{start}, q_{accept})$ a GNFA that is language equivalent to $D$.

dfa_to_gnfa($D$):

$Q' := Q \cup \{q_{start}, q_{accept}\}$
**for** $q \in Q'$ **do** ▷ only connect $q_{start}$ with $q_0$
  **if** $q = q_0$ **then**
    $\delta'(q_{start}, q) := \mathbf{1}$
  **else**
    $\delta'(q_{start}, q) := \mathbf{0}$
**for** $(q, q') \in Q \times Q$ **do**
  $\delta'(q, q') = +\{\mathsf{regexp}(a) \mid \delta(q, a) = q'\}$
**for** $q \in Q'$ **do** ▷ connect each final state of $D$ to $q_{accept}$
  **if** $q \in F$ **then**
    $\delta'(q, q_{accept}) := \mathbf{1}$
  **else**
    $\delta'(q, q_{accept}) := \mathbf{0}$
**return** $G = (Q', \Sigma, \delta', q_{start}, q_{accept})$

**Algorithm 4** Reducing a GNFA to a 2-state GNFA

**Input** $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ a GNFA
**Output** 2-state GNFA $G' = (\{q_{start}, q_{accept}\}, \Sigma, \delta', q_{start}, q_{accept})$
where $G'$, hence the regular expression $\delta'(q_{start}, q_{accept})$, language equivalent to $G$.

gnfa_minimize($G$):

**for** $q_{rip} \in Q - \{q_{start}, q_{accept}\}$ **do**
  $Q := Q \setminus \{q_{rip}\}$
  **for** $q_i \in Q - \{q_{accept}\}$ **do**
    **for** $q_j \in Q - \{q_{start}\}$ **do**
      $\delta(q_i, q_j) := \mathsf{regexp\_simplify}(\delta(q_i, q_{rip}) \cdot \delta(q_{rip}, q_{rip})^* \cdot \delta(q_{rip}, q_j) + \delta(q_i, q_j))$
$\delta' := \{:\}$
$\delta'(q_{start}, q_{accept}) := \delta(q_{start}, q_{accept})$
$\delta'(q_{start}, q_{start}) := \mathbf{0}$
$\delta'(q_{accept}, q_{accept}) := \mathbf{0}$
$\delta'(q_{accept}, q_{start}) := \mathbf{0}$
**return** $G' = (Q, \Sigma, \delta', q_{start}, q_{accept})$

**Algorithm 5** Converts a DFA to a language equivalent regular expression

**Input** $D = (Q, \Sigma, \delta, q_0, F)$: a DFA
**Output** $r$: a regular expression with $\mathcal{L}(r) = \mathcal{L}(D)$

dfa_to_regexp($D$):
1: $G :=$ dfa_to_gnfa($D$)
2: gnfa_minimize($G$)
3: **return** $G.\delta(G.q_{start}, G.q_{accept})$

### 3.2 Minimization of a DFA using the table filling method

**Algorithm 6** Minimizing a DFA

**Input** DFA $D = (Q, \Sigma, \delta, q_0, F)$: with $Q = \{q_0, \ldots, q_{n-1}\}$
**Output** DFA $D'$, equivalent to $D$ and minimal in number of states

$\mathsf{dfa\_minimize}(D)$:

▷ initialization of the lower triangle of table, a $[0..n-1] \times [0..n-1]$ matrix
**for** $0 \leqslant i \leqslant j < n$ **do** ▷ Note $i \leqslant j$ rather than $i < j$
    table[i,j] := $(q_i \in F \iff q_j \in F)$

▷ comparing pairs of states on (bounded) equivalence,
▷ until no change occurs
changed := true
**while** changed **do**
    changed := false
    **for** $0 \leqslant i < j < n$ **do**
        **if** table[i,j] **then**
            **for** $a \in \Sigma$ **do**
                **let** $q_k = \delta(q_i, a)$
                **let** $q_\ell = \delta(q_j, a)$
                **if** $\neg$table[$\min\{k, \ell\}, \max\{k, \ell\}$] **then** ▷ $k$ and $\ell$ may be equal
                    table[i,j] := false
                    changed := true
                    **break** ▷ skip remaining symbols in $\Sigma$
$D'$ := $\mathsf{dfa\_from\_table}$(D,table)
**return** $D'$

$\mathsf{dfa\_identifiable}(D_1, D_2)$:

**Algorithm 7** Constructing a DFA from a minimization table

**Input** DFA $D = (Q, \Sigma, \delta, q_0, F)$ with $Q = \{q_0, \ldots, q_{n-1}\}$ and Boolean matrix table
If table[i,j] = true, for any $0 < i < j < n$, then states $q_i$ and $q_j$ can be identified

dfa_from_table($D, table$):

$\triangleright$ form sets $Q_i$ of states that can be identified according to the table
$\triangleright$ $R$ is used to see if a state is already included
$R := \emptyset$
**for** $0 \leqslant i < n$ **do**
    **if** $q_i \notin R$ **then**
        $Q_i := \{q_i\}$
        $R := R \cup \{q_i\}$
        **for** $i < j < n$ **do**
            **if** table[i,j] **then**
                $Q_i := Q_i \cup \{q_j\}$
                $R := R \cup \{q_j\}$

$\triangleright$ constructing minimal DFA $D'$
$Q' = \{\, Q_i \mid 0 \leqslant i < n \,\}$
**let** $Q_0$ **such that** $q_0 \in Q_0$
**for** $Q_i \in Q'$, $a \in \Sigma$ **do**
    $q' := \delta(q_i, a)$
    **let** $j$ **such that** $q' \in Q_j$
    $\delta'(Q_i, a) := Q_j$
$F' = \{\, Q_i \mid q_i \in F \,\}$

**return** $D' = (Q', \Sigma, \delta', Q_0, F')$

**Algorithm 8** Decide if DFAs $D_1$ and $D_2$ constitute isomorphic DFAs

**Input** DFA $D_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$, DFA $D_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$

$\mathsf{dfa\_isomorphic}(D_1, D_2)$:

▷ maintain boolean matrix *matching*<br>
▷ and a set of pairs of states *to_inspect*<br>
**for** $q_1 \in Q_1$ and $q_2 \in Q_2$ **do**<br>
    $\mathit{matching}(q_1, q_2) := \mathit{false}$<br>
**if** $q_0^1 \in F_1 \iff q_0^2 \in F_2$ **then**<br>
    $\mathit{matching}(q_0^1, q_0^2) := \mathit{true}$<br>
    $\mathit{to\_inspect} := \{(q_0^1, q_0^2)\}$<br>
**else**<br>
    **return** *false*<br>
▷ explore DFA $D_1$<br>
**while** $\mathit{to\_inspect} \neq \emptyset$ **do**<br>
    **choose** $(q_1, q_2) \in \mathit{to\_inspect}$<br>
    **for** $a \in \Sigma$ **do**<br>
        $q_1' := \delta_1(q_1, a)$, $q_2' := \delta_2(q_2, a)$<br>
        **if** $\mathit{matching}(q_1', q_2')$ **then**<br>
            **if** $q_1' \in F_1 \iff q_2' \in F_2$ **then**<br>
                $\mathit{matching}(q_1', q_2') := \mathit{true}$<br>
                $\mathit{to\_inspect} := \{(q_1', q_2')\}$<br>
            **else**<br>
                **return** *false*<br>
▷ check if each state of $D_1$ has a unique matching state<br>
**for** $q_1 \in Q_1$ **do**<br>
    $\mathit{count} := 1$<br>
    **for** $q_2 \in Q_2$ **do**<br>
        **if** $\mathit{matching}(q_1, q_2)$ **then**<br>
            $\mathit{count} \mathrel{+}= 1$<br>
    **if** $\mathit{count} \neq 1$ **then**<br>
        **return** *false*<br>
▷ check if each state of $D_2$ has a unique matching state<br>
**for** $q_2 \in Q_2$ **do**<br>
    $\mathit{count} := 1$<br>
    **for** $q_1 \in Q_1$ **do**<br>
        **if** $\mathit{matching}(q_1, q_2)$ **then**<br>
            $\mathit{count} \mathrel{+}= 1$<br>
    **if** $\mathit{count} \neq 1$ **then**<br>
        **return** *false*<br>
▷ DFAs are isomorphic is a proper matching exists<br>
**return** *true*

## 4 NFA algorithms

In the algorithms below we use the convention that for NFA $N = (Q, \Sigma, \delta, q_0, F)$ the function $\delta$ may be partially defined. For all inputs $(q, a)$ where $\delta$ is undefined we assume that $\delta(q, a) = \emptyset$.

**Algorithm 9** Epsilon closure

**Input**: $N = (Q, \Sigma, \delta, q_0, F)$: an NFA; $q \in Q$: a state
**Output**: $\{q' \in Q \mid q \xrightarrow{\varepsilon} q'\}$

$\mathsf{nfa\_epsilon\_closure}(N, q)$:

$result := \{q\}$
$todo := \{q\}$
**while** $todo \neq \emptyset$ **do**
    $q := todo.pop()$ ▷ *pop* removes and returns an arbitrary element of a set
    $Q_1 := \delta(q, \varepsilon) \setminus result$
    $result := result \cup Q_1$
    $todo := todo \cup Q_1$
**return** $result$

We generalize the epsilon closure to a set of states $Q$ by

$$\mathsf{nfa\_epsilon\_closure}(N, Q) = \bigcup \{\mathsf{nfa\_epsilon\_closure}(N, q) \mid q \in Q\}.$$

**Algorithm 10** Test if an NFA accepts a given word

**Input**: $N = (Q, \Sigma, \delta, q_0, F)$: an NFA; $w \in \Sigma^*$ a word
**Output:** $w \in \mathcal{L}(N)$

$\mathsf{nfa\_accepts\_word}(N, w)$:

$q := \mathsf{nfa\_epsilon\_closure}(N, q_0)$
**for** $a \in w$ **do**
    $q := \bigcup \{\mathsf{nfa\_epsilon\_closure}(N, \delta(q_i, a)) \mid q_i \in q\}$
**return** $q \cap F \neq \emptyset$

**Algorithm 11** Generate accepted words in an NFA up to a given length $n$

**Input**: $N = (Q, \Sigma, \delta, q_0, F)$: an NFA, $n$: a natural number
**Output:** $\{w \in \mathcal{L}(N) \mid |w| \leq n\}$

nfa_words_up_to_n$(N, n)$

$F_1 := \{q \in Q \mid \mathsf{nfa\_epsilon\_closure}(N, q) \cap F \neq \emptyset\}$ ▷ states that can terminate
$result := \emptyset$
**if** $q_0 \in F_1$ **then**
  $result := result \cup \{\varepsilon\}$
$W := \{(q, \varepsilon) \mid q \in \mathsf{nfa\_epsilon\_closure}(N, q_0)\}$
**for** $0 \leq i < n$ **do**
  $W' := \{:\}$
  **for** $(q, words) \in W$ **do**
    **for** $a \in \Sigma$ **do**
      **for** $q_1 \in \mathsf{nfa\_epsilon\_closure}(N, \delta(q, a))$ **do**
        $words' := \{wa \mid w \in words\}$
        $W'(q_1) := W'(q_1) \cup words'$
        **if** $q_1 \in F_1$ **then**
          $result := result \cup words'$
  $W := W'$
**return** $result$

**Algorithm 12** Convert an NFA into a language equivalent DFA

**Input**: $N = (Q_N, \Sigma, \delta_N, q_N, F_N)$: An NFA
**Output**: $D$: a DFA with $\mathcal{L}(D) = \mathcal{L}(N)$

nfa_to_dfa($N$):

$F := \emptyset$
$Q_0 :=$ nfa_epsilon_closure($N, q_N$)
$Q := \{Q_0\}$
$\delta := \{:\}$ ▷ $\{:\}$ is the empty mapping
**if** $Q_0 \cap F_N \neq \emptyset$ **then**
    $F := F \cup \{Q_0\}$
$todo := [Q_0]$
**while** $todo \neq \emptyset$ **do**
    $Q_1 := todo[0]$
    $todo := todo[1..]$
    **for** $a \in \Sigma$ **do**
        $Q_2 := \emptyset$
        **for** $q_1 \in Q_1$ **do**
            $Q_2 := Q_2 \cup$ nfa_epsilon_closure($N, \delta_N(q_1, a)$)
        $\delta(Q_1, a) := Q_2$
        **if** $Q_2 \cap F_N \neq \emptyset$ **then**
            $F := F \cup \{Q_2\}$
        **if** $Q_2 \notin Q$ **then**
            $Q := Q \cup \{Q_2\}$
            $todo := todo +\!\!+ [Q_2]$
**return** $(Q, \Sigma, \delta, Q_0, F)$

**Algorithm 13** Computes the repetition of an NFA

**Input**: $N = (Q, \Sigma, \delta, q_0, F)$: an NFA
**Output:** $N'$: an NFA with $\mathcal{L}(N') = \mathcal{L}(N^*)$

nfa_repetition($N$):

$q_0' :=$ fresh_state()
$Q' := Q \cup \{q_0\}$
$F' := F \cup \{q_0\}$
$\delta' := \delta$
**for** $q \in F$ **do**
    $\delta'(q, \varepsilon) := \delta'(q, \varepsilon) \cup \{q_0\}$
$\delta'(q_0', \varepsilon) := \{q_0\}$
**return** $N' = (Q', \Sigma, \delta', q_0', F')$

**Algorithm 14** Computes the union of two NFAs

**Input**: $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$: an NFA; $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$: an NFA
**Output:** $N'$: an NFA with $\mathcal{L}(N') = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$

nfa_union$(N)$:

$q'_0 :=$ fresh_state$()$
$Q' := Q_1 \cup Q_2 \cup \{q_0\}$
$F' := F_1 \cup F_2$
$\delta' := \delta_1 \cup \delta_2$
$\delta'(q'_0, \varepsilon) := \{q_1, q_2\}$
**return** $N' = (Q', \Sigma, \delta', q'_0, F')$

**Algorithm 15** Computes the concatenation of two NFAs

**Input**: $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$: an NFA; $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$: an NFA
**Output:** $N'$: an NFA with $\mathcal{L}(N') = \mathcal{L}(N_1) \circ \mathcal{L}(N_2)$

nfa_concatenation$(N)$:

$q'_0 := q_1$
$Q' := Q_1 \cup Q_2 \cup \{q_0\}$
$F' := F_2$
$\delta' := \delta_1 \cup \delta_2$
**for** $q \in F_1$ **do**
    $\delta'(q, \varepsilon) := \delta'(q, \varepsilon) \cup \{q_2\}$
**return** $N' = (Q', \Sigma, \delta', q'_0, F')$

# 5 PDA algorithms

We define a PDA state as a tuple $(q, s)$ with $q \in Q$ a state and $s \in \Gamma^*$ a stack content. We define the functions can_pop_push and pop_push as follows:

$$\begin{aligned}
\mathsf{can_pop_push}(stack, u, v) &= (u = \varepsilon) \vee (u \neq \varepsilon \wedge stack = stack' \mathbin{+\!\!+} [u]) \\
\mathsf{pop_push}(stack, u, v) &= \begin{cases} stack & \text{if } u = \varepsilon \text{ and } v = \varepsilon \\ stack \mathbin{+\!\!+} [v] & \text{if } u = \varepsilon \text{ and } v \neq \varepsilon \\ stack[:|stack|-1] & \text{if } u \neq \varepsilon \text{ and } v = \varepsilon \\ stack[:|stack|-1] \mathbin{+\!\!+} [v] & \text{if } u \neq \varepsilon \text{ and } v \neq \varepsilon \end{cases}
\end{aligned}$$

**Algorithm 16** PDA epsilon closure

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA; $r \in Q \times \Gamma^*$: a PDA state
**Output**: $\{r' \in Q \times \Gamma^* \mid r \xrightarrow{\varepsilon}\!\!\!\rightarrow r'\}$

pda_epsilon_closure$(P, r)$:

$result := \{r\}$
$todo := \{r\}$
**while** $todo \neq \emptyset$ **do**
    $r := todo.pop()$ ▷ *pop* removes and returns an arbitrary element of a set
    **let** $r = (p, stack)$
    **for** $u \in \Gamma_\varepsilon$ **do**
        $Q_1 := \delta(p, \varepsilon, u)$
        **for** $(q, v) \in Q_1$ **do**
            **if** can_pop_push$(stack, u, v)$ **then**
                $stack' =$ pop_push$(stack, u, v)$
                $r' := (q, stack')$
                **if** $r' \notin result$ **then**
                    $todo := todo \cup \{r'\}$
                    $result := result \cup \{r'\}$
**return** $result$

We generalize the functions pda_epsilon_closure and pda_do_transition to a set of PDA states $R$ by

$$\begin{aligned}
\mathsf{pda_epsilon_closure}(P, R) &= \bigcup\{\mathsf{pda_epsilon_closure}(P, r) \mid r \in R\} \\
\mathsf{pda_do_transition}(P, a, R) &= \bigcup\{\mathsf{pda_do_transition}(P, a, r) \mid r \in R\}.
\end{aligned}$$

**Algorithm 17** Do a transition in a PDA

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA; $a \in \Sigma$: a symbol; $r \in Q \times \Gamma^*$: a PDA state
**Output**: $\{r' \in Q \times \Gamma^* \mid r \xrightarrow{a} r'\}$

pda_do_transition$(P, a, r)$:

**let** $r = (p, stack)$
$result := \emptyset$
**for** $u \in \Gamma_\varepsilon$ **do**
    $Q_1 := \delta(p, a, u)$
    **for** $(q, v) \in Q_1$ **do**
        **if** can_pop_push$(stack, u, v)$ **then**
            $stack' =$ pop_push$(stack, u, v)$
            $r' := (q, stack')$
            $result := result \cup \{r'\}$
**return** $result$

**Algorithm 18** Test if a PDA accepts a given word

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA; $w \in \Sigma^*$ a word
**Output:** $w \in \mathcal{L}(P)$

pda_accepts_word$(P, w)$:

$R := \{(q_0, [])\}$
$R :=$ pda_epsilon_closure$(P, R)$
**for** $a \in w$ **do**
    $R :=$ pda_do_transition$(P, a, R)$
    $R :=$ pda_epsilon_closure$(P, R)$
**return** $\exists (q, stack) \in R : q \in F$

**Algorithm 19** Generate accepted words in a PDA up to a given length $n$

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA, $n$: a natural number
**Output:** $\{w \in \mathcal{L}(P) \mid |w| \leq n\}$

```
pda_words_up_to_n(P, n)
result := ∅
W := {:}
R := {(q_0, [ ])}
R := pda_epsilon_closure(P, R)
for r ∈ R do
    W(r) := {ε}
    let r = (q, stack)
    if q ∈ F then
        result := result ∪ {ε}
for 0 ≤ i < n do
    W' := {:}
    for (r, words) ∈ W do
        for a ∈ Σ do
            R := pda_do_transition(P, a, r)
            R := pda_epsilon_closure(P, R)
            words' := {wa | w ∈ words}
            for r' ∈ R do
                W'(r') := W'(r') ∪ words'
                let r' = (q', stack')
                if q' ∈ F then
                    result := result ∪ words'
    W := W'
return result
```

**Algorithm 20** Converts a PDA into a language equivalent CFG

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA in push/pop format
**Output:** $G$: a CFG with $\mathcal{L}(G) = \mathcal{L}(P)$

pda_to_cfg($P$):

$V := \{A_{pq} \mid (p, q) \in Q \times Q\}$
$R := \emptyset$
**let** $F = \{q_{accept}\}$
$S := A_{q_0 q_{accept}}$
$T_{push} := \{:\}$ ▷ maps stack symbols to corresponding push transitions
$T_{pop} := \{:\}$ ▷ maps stack symbols to corresponding pop transitions
**for** $((p, a, u), Q_1) \in \delta$ **do**
    **for** $(q, v) \in Q_1$ **do**
        **if** $u = \varepsilon$ **then**
            $T_{push}[v] := T_{push}[v] \cup \{(p, a, \varepsilon) \to (q, v)\}$
        **else if** $v = \varepsilon$ **then**
            $T_{pop}[u] := T_{pop}[u] \cup \{(p, a, u) \to (q, \varepsilon)\}$
**for** $u \in \Gamma$ **do**
    **for** $((p, a, \varepsilon) \to (r, u), (s, b, u) \to (q, \varepsilon)) \in T_{push}[u] \times T_{pop}[u]$ **do**
        $R := R \cup \{A_{pq} \to aA_{rs}b\}$
**for** $(p, q, r) \in Q \times Q \times Q$ **do**
    $R := R \cup \{A_{pq} \to A_{pr}A_{rq}\}$
**for** $p \in Q$ **do**
    $R := R \cup \{A_{pp} \to \varepsilon\}$
**return** $G = (V, \Sigma, R, S)$

**Algorithm 21** Converts a PDA into a PDA accepting on empty stack

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA;
**Output**: $P = (Q', \Sigma, \Gamma', \delta', q_s, F')$: a PDA accepting on empty stack

```
let q_s, q'_a, q_a be fresh for Q
Q' = Q ∪ {q_s, q'_a, q_a}
▷ q_s new start state, q_a new single final state, q'_a additional state
let $ be fresh for Γ
Γ' = Γ ∪ {$}
▷ introduce stack-bottom $ file
▷ by convention, if not specified δ'(q, α, X) = ∅
for q ∈ Q, a ∈ Σ, X ∈ Γ do
    δ'(q, a, X) = δ(q, a, X)
δ'(q_s, ε, ε) = {(q_0, $)}
δ'(q, ε, ε) = δ(q, ε, ε) ∪ {(q'_a, ε)} for q ∈ F
δ'(q'_a, α, X) = ∅ for α ∈ Σ, X ∈ Γ'_ε
δ'(q'_a, ε, X) = {(q'_a, ε)} for X ∈ Γ
δ'(q'_a, ε, $) = {(q_a, ε)}
F' = {q_a}
return P'
```

**Algorithm 22** Converts –in place– a PDA into a PDA without push/pop or skip transitions

**Input**: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA;
**Output**: $P = (Q', \Sigma, \Gamma', \delta', q_0, F)$: a PDA with push and/or pop transitions only

```
let ¢ be fresh for Γ
Γ' = Γ ∪ {¢}
▷ remove push/pop transitions: replace q --α,X/Y--> q' by q --α,X/ε--> q̂ --ε,ε/Y--> q'
for q ∈ Q, α ∈ Σ_ε, X ∈ Γ do
    for q' ∈ Q, Y ∈ Γ such that (q', Y) ∈ δ(q, α, X) do
        let q̂ be fresh for Q
        Q = Q ∪ {q̂}
        δ(q, α, X) = (δ(q, α, X) \ {(q', Y)}) ∪ {(q̂, ε)}
        δ(q̂, ε, ε) = {(q', Y)}
        ▷ implicitly δ(q̂, a, Z) = ∅ for a ∈ Σ, Z ∈ Γ_ε
▷ remove ε/ε-transitions: replace q --α,ε,ε--> q' by q --α,ε,¢--> q̂ --ε,¢/ε--> q'
for q ∈ Q, α ∈ Σ_ε do
    for q' ∈ Q such that (q', ε) ∈ δ(q, α, ε) do
        let q̂ be fresh for Q
        Q = Q ∪ {q̂}
        δ(q, α, ε) = (δ(q, α, ε) \ {(q', ε)}) ∪ {q̂, ¢)}
        δ(q̂, ε, ¢) = {(q', ε)}
        ▷ implicitly δ(q̂, a, Z) = ∅ for a ∈ Σ, Z ∈ Γ_ε
Γ = Γ'
return P
```

# 6 TM algorithms

**Algorithm 23** Do a transition in a TM

**Input**: $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$: a TM; $p \in Q$ a state; $tape \in \Gamma^*$: a tape; $head \in \mathbb{N}$: the position of the tape head
**Output**: $q \in Q$ the new state; $head'$ the new position of the tape; N.B. $tape$ is modified

tm_do_transition($T, p, tape, head$):

$a := tape[head]$
**if** $(p, a) \in \mathsf{domain}(\delta)$ **then**
  $(q, b, d) := \delta(p, a)$
**else**
  $(q, b, d) := (q_{reject}, a, R)$
$tape[head] := b$
**if** $d = L$ **then**
  $head' := max(head - 1, 0)$
**else if** $d = R$ **then**
  $head' := head + 1$
**if** $|tape| = head'$ **then**
  $tape := tape ++ [\sqcup]$ ▷ add a blank, so that tape[head] is defined
**return** $(q, head')$

**Algorithm 24** Test if a TM accepts a given word

**Input**: $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$: a TM; $w \in \Sigma^*$ a word
**Output:** $w \in \mathcal{L}(T)$

```
tm_accepts_word(T, w):
q := q_0
tape := w
head := 0
if |tape| = 0 then
    tape := tape ++ [⊔]        ▷ add a blank, so that tape[head] is defined
while true do
    (q, head) := tm_do_transition(T, q, tape, head)
    if q == q_accept then
        return true
    if q == q_reject then
        return false
```

**Algorithm 25** Simulate the execution of a word on a TM

**Input**: $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$: a TM; $w \in \Sigma^*$ a word
**Output:** $result \in (Q \times \Gamma^* \times \mathbb{N})^*$ the intermediate states of the execution

```
tm_simulate_word(T, w):
q := q_0
tape := w
head := 0
if |tape| = 0 then
    tape := tape ++ [⊔]        ▷ add a blank, so that tape[head] is defined
result := [(q, tape, head)]
while true do
    (q, head) := tm_do_transition(T, q, tape, head)
    result := result ++ [(q, tape, head)]
    if q == q_accept then
        break
    if q == q_reject then
        break
return result
```

**Algorithm 26** Generate accepted words in a TM up to a given length $n$

**Input**: $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$: a TM; $n$: a natural number
**Output:** $\{w \in \mathcal{L}(T) \mid |w| \leq n\}$

```
tm_words_up_to_n(P, n)
result := ∅
for 0 ≤ i ≤ n do
    for w ∈ Σ^n do
        if tm_accepts_word(T, w) then
            words := words ∪ {w}
```

# 7 CFG algorithms

**Algorithm 27** Test if a CFG accepts a given word using the CYK algorithm

**Input**: $G = (V, \Sigma, R, S)$: a context free grammar in Chomsky normal form; $w \in \Sigma^*$ a word
**Output:** $w \in \mathcal{L}(G)$

cfg_accepts_word$(r, w)$:

```
if w = ε then
    return S → ε ∈ R
n := |w|
X := {:}                                        ▷ initially X(i, j) = ∅ for all i, j
                                 ▷ X(i, j) will contain {A ∈ V | A ⇒* w_i w_{i+1} ... w_j}
for 0 ≤ i < n do
    X(i, i) := {A ∈ V | A → w_i ∈ R}
for 1 ≤ m < n do
    for 0 ≤ i < n − m do
        j := i + m
        for i ≤ k ≤ j do
            for (B, C) ∈ X(i, k) × X(k + 1, j) do
                X(i, j) := X(i, j) ∪ {A ∈ V | A → BC ∈ R}
return S ∈ X(0, n − 1)
```

## 7.1 Generating the language corresponding to a CFG

Given the grammar $G = (V, \Sigma, R, S)$, we define $R_n = \{A \to x \mid |x| = n\}$, and we define $G_n = (V, \Sigma, R_n, S)$ The algorithm below generates all words accepted by $G$ up to a given length. N.B. This algorithm is not particularly efficient.

**Algorithm 28** Generate accepted words of a CFG up to a given length $n$

**Input**: $G = (V, \Sigma, R, S)$: a CFG in Chomsky normal form; $n$: a natural number
**Output:** $\{w \in \mathcal{L}(G) \mid |w| \leq n\}$

cfg_words_up_to_n$(G, n)$

```
words := ∅
if S → ε ∈ R then
    words := words ∪ {ε}
W := [S]
words := words ∪ {w ∈ Σ* | ∃x ∈ W : x ⇒*_{G_1} w}
for 2 ≤ i ≤ n do
    W := {w ∈ Σ* | ∃x ∈ W : x ⇒_{G_2} w}
    words := words ∪ {w ∈ Σ* | ∃x ∈ W : x ⇒*_{G_1} w}
return words
```

## 7.2 Eliminating unit rules

The following two algorithms were found on http://www.iitg.ac.in/gkd/ma513/oct/oct17/note.pdf, and have been slightly modified. A unit rule has the form $A \to B$, where $A$ and $B$ are variables.

**Algorithm 29** Find the set of derivable variables in a CFG

**Input**: $G = (V, \Sigma, R, S)$: a CFG; $A \in V$ a variable
**Output:** $\{B \in V \mid B \neq A \wedge A \Rightarrow^* B\}$

cfg_derivable_variables$(G, A)$

```
W := ∅
W' := ∅
for A → B ∈ R with B ∈ V do
    W := W ∪ {B}
while W' ≠ W do
    W' := W
    for C → B ∈ R with C ∈ W' and B ∈ V do
        W := W ∪ {B}
return W \ {A}
```

**Algorithm 30** Eliminate unit rules from a CFG

**Input**: $G = (V, \Sigma, R, S)$: a CFG
**Output:** $G'$ with $\mathcal{L}(G) = \mathcal{L}(G')$ such that $G'$ has no unit rules

```
cfg_eliminate_unit_rules(G, A)
R' := R
for A ∈ V do
    W := cfg_derivable_variables(G, A)
    for B → α ∈ R with B ∈ W and α ∉ V do
        R' := R' ∪ {A → α}                ▷ Make sure not to add duplicate rules
R' := {A → α ∈ R' | α ∉ V}
return G' = (V, Σ, R', S)
```

**Algorithm 31** Find the set of reachable variables in a CFG

**Input**: $G = (V, \Sigma, R, S)$: a CFG
**Output:** $\{A \in V \mid \exists x, y \in (V \cup \Sigma)^* : S \Rightarrow^* xAy\}$

cfg_reachable_variables($G$)

$W := \{S\}$
**while** true **do**
    $W' := \{A \in V \setminus W \mid \exists B \to xAy \in R \text{ with } B \in W \text{ and } x, y \in (V \cup \Sigma)^* \}$
    $W := W \cup W'$
    **if** $W' = \emptyset$ **then**
        **break**
**return** $W$

**Algorithm 32** Find the set of productive variables in a CFG

**Input**: $G = (V, \Sigma, R, S)$: a CFG
**Output:** $\{A \in V \mid \exists w \in \Sigma^* : A \Rightarrow^* w\}$

cfg_productive_variables($G$)

$W := \Sigma$
**while** true **do**
    $W' := \{A \in V \setminus W \mid \exists A \to x \in R \text{ with } x \in W^* \}$
    $W := W \cup W'$
    **if** $W' = \emptyset$ **then**
        **break**
**return** $W \setminus \Sigma$

## 8 Regular expression algorithms

The functions regexp_size and regexp_symbols are inductively defined as

$$\begin{aligned}
\mathsf{regexp_size}(\mathbf{0}) &= 0\\
\mathsf{regexp_size}(\mathbf{1}) &= 0\\
\mathsf{regexp_size}(a) &= 0\\
\mathsf{regexp_size}(r^*) &= \mathsf{regexp_size}(r) + 1\\
\mathsf{regexp_size}(r_1 + r_2) &= \mathsf{regexp_size}(r_1) + \mathsf{regexp_size}(r_2) + 2\\
\mathsf{regexp_size}(r_1 \cdot r_2) &= \mathsf{regexp_size}(r_1) + \mathsf{regexp_size}(r_2) + 2
\end{aligned}$$

$$\begin{aligned}
\mathsf{regexp_symbols}(\mathbf{0}) &= \emptyset\\
\mathsf{regexp_symbols}(\mathbf{1}) &= \emptyset\\
\mathsf{regexp_symbols}(a) &= \{a\}\\
\mathsf{regexp_symbols}(r^*) &= \mathsf{regexp_symbols}(r)\\
\mathsf{regexp_symbols}(r_1 + r_2) &= \mathsf{regexp_symbols}(r_1) \cup \mathsf{regexp_symbols}(r_2)\\
\mathsf{regexp_symbols}(r_1 \cdot r_2) &= \mathsf{regexp_symbols}(r_1) \cup \mathsf{regexp_symbols}(r_2)
\end{aligned}$$

The functions regexp_simplify is defined by means of the following rewrite rules

$$\begin{aligned}
\mathsf{regexp_simplify}(\mathbf{0}^*) &= \mathbf{1}\\
\mathsf{regexp_simplify}(\mathbf{1}^*) &= \mathbf{1}\\
\mathsf{regexp_simplify}(r^{**}) &= r^*\\
\mathsf{regexp_simplify}(\mathbf{0} + r) &= r\\
\mathsf{regexp_simplify}(r + \mathbf{0}) &= r\\
\mathsf{regexp_simplify}(\mathbf{0} \cdot r) &= \mathbf{0}\\
\mathsf{regexp_simplify}(\mathbf{1} \cdot r) &= r\\
\mathsf{regexp_simplify}(r \cdot \mathbf{0}) &= \mathbf{0}\\
\mathsf{regexp_simplify}(r \cdot \mathbf{1}) &= r
\end{aligned}$$

$$\begin{aligned}
\mathsf{regexp_accepts_word}(\mathbf{0}) &= \emptyset\\
\mathsf{regexp_accepts_word}(\mathbf{1}) &= \emptyset\\
\mathsf{regexp_accepts_word}(a) &= \{a\}\\
\mathsf{regexp_accepts_word}(r^*) &= \mathsf{regexp_accepts_word}(r)\\
\mathsf{regexp_accepts_word}(r_1 + r_2) &= \mathsf{regexp_accepts_word}(r_1) \cup \mathsf{regexp_accepts_word}(r_2)\\
\mathsf{regexp_accepts_word}(r_1 \cdot r_2) &= \mathsf{regexp_accepts_word}(r_1) \cup \mathsf{regexp_accepts_word}(r_2)
\end{aligned}$$

The algorithms regexp_accepts_word and regexp_words_up_to_n are used to detect if a word is in the language of a regular expression, and to generate words in that are in the language. These algorithms are not very efficient. We use $w[{:}k]$ as shorthand for $w_0 \dots w_{k-1}$ and $w[k{:}]$ as shorthand for $w_k \dots w_{|w|-1}$.

**Algorithm 33** Test if a regular expression matches a given word

**Input**: $r$: a regular expression; $w$ a word
**Output:** $w \in \mathcal{L}(r)$

regexp_accepts_word$(r, w)$:

**if** $r = \mathbf{0}$ **then**
    **return** $false$
**else if** $r = \mathbf{1}$ **then**
    **return** $|w| = 0$
**else if** $r = a$ **then**
    **return** $w = a$
**else if** $r = r_1 + r_2$ **then**
    **return** regexp_accepts_word$(r_1, w) \vee$ regexp_accepts_word$(r_2, w)$
**else if** $r = r_1 \cdot r_2$ **then**
    **return** $\exists 0 \leq k \leq |w|$ : regexp_accepts_word$(r_1, w[{:}k] \wedge$ regexp_accepts_word$(r_2, w[k{:}]$
**else if** $r = r_1^*$ **then**
    **if** $r_1 = \mathbf{0}$ **then**
        **return** $|w| = 0$
    **else**
        **return** $\exists 1 \leq k \leq |w|$ : regexp_accepts_word$(r_1, w[{:}k] \wedge$ regexp_accepts_word$(r, w[k{:}])$

**Algorithm 34** Generate words matching a regular expression up to a given length $n$

**Input**: $r$: a regular expression; $n$ a natural number
**Output:** $\{w \in \mathcal{L}(r) \mid |w| \leq n\}$

regexp_words_up_to_n$(r, n)$

**if** $r = \mathbf{0}$ **then**
  **return** $\emptyset$
**else if** $r = \mathbf{1}$ **then**
  **return** $\{\varepsilon\}$
**else if** $r = a$ **then**
  **if** $n > 0$ **then**
    **return** $\{a\}$
  **else**
    **return** $\emptyset$
**else if** $r = r_1 + r_2$ **then**
  **return** regexp_words_up_to_n$(r_1, n) \cup$ regexp_words_up_to_n$(r_2, n)$
**else if** $r = r_1 \cdot r_2$ **then**
  **return** $\cup\{$regexp_words_up_to_n$(r_1, k) \circ$ regexp_words_up_to_n$(r_2, n-k) \mid 0 \leq k \leq |w|\}$
**else if** $r = r_1^*$ **then**
  **if** $r_1 = \mathbf{0}$ **then**
    **return** $\{\varepsilon\}$
  **else**
    **return** $\cup\{$regexp_words_up_to_n$(r_1, k) \circ$ regexp_words_up_to_n$(r_1, n-k) \mid 1 \leq k \leq |w|\} \cup \{\varepsilon\}$

# 9 Syntax

**Symbols** A $\langle symbol \rangle$ is an arbitrary alpha-numeric or UTF-8 character, with the exception of those that are used as tokens in the grammars below.

$\langle blank \rangle$ ::= '`blank`' $\langle symbol \rangle$

$\langle epsilon \rangle$ ::= '`epsilon`' $\langle symbol \rangle$

$\langle direction \rangle$ ::= '`L`' | '`R`'

$\langle input\text{-}symbols \rangle$ ::= '`input_symbols`' $\langle symbol \rangle^*$

$\langle stack\text{-}symbols \rangle$ ::= '`stack_symbols`' $\langle symbol \rangle^*$

$\langle tape\text{-}symbols \rangle$ ::= '`tape_symbols`' $\langle symbol \rangle^*$

$\langle identifier \rangle$ ::= $\langle symbol \rangle^*$

An $\langle identifier \rangle$ must be a single token, i.e. it may not contain spaces.

### States

$\langle states \rangle$ ::= '`states`' $\langle state \rangle^*$

$\langle initial \rangle$ ::= '`initial`' $\langle state \rangle$

$\langle accept \rangle$ ::= '`accept`' $\langle state \rangle$

$\langle reject \rangle$ ::= '`reject`' $\langle state \rangle$

$\langle final \rangle$ ::= '`final`' $\langle state \rangle^*$

$\langle state \rangle$ ::= $\langle identifier \rangle$

### DFA syntax

$\langle DFA \rangle$ ::= $\langle dfa\text{-}line \rangle^*$

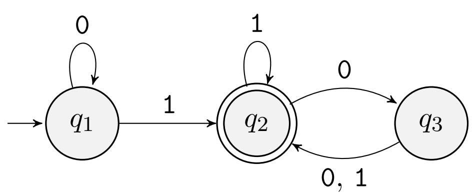
$\langle dfa\text{-}line \rangle$ ::= $\langle input\text{-}symbols \rangle$
| $\langle states \rangle$
| $\langle initial \rangle$
| $\langle final \rangle$
| $\langle dfa\text{-}transition \rangle$

$\langle \textit{dfa-transition} \rangle$ ::= $\langle \textit{state} \rangle$ $\langle \textit{state} \rangle$ $\langle \textit{symbol} \rangle$

The elements $\langle \textit{input-symbols} \rangle$ and $\langle \textit{states} \rangle$ are optional. If they are omitted, they are derived from the transitions.

For example, the DFA below is specified using

```
initial q1
final q2
q1 q1 0
q1 q2 1
q2 q2 1
q2 q3 0
q3 q2 0 1
```




**NFA syntax**

$\langle \textit{NFA} \rangle$ ::= $\langle \textit{nfa-line} \rangle$*

$\langle \textit{nfa-line} \rangle$ ::= $\langle \textit{input-symbols} \rangle$
| $\langle \textit{epsilon} \rangle$
| $\langle \textit{states} \rangle$
| $\langle \textit{initial} \rangle$
| $\langle \textit{final} \rangle$
| $\langle \textit{nfa-transition} \rangle$

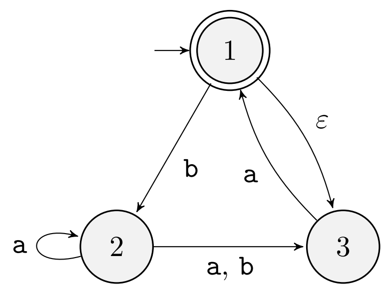
$\langle \textit{nfa-transition} \rangle$ ::= $\langle \textit{state} \rangle$ $\langle \textit{state} \rangle$ $\langle \textit{nfa-update} \rangle$*

$\langle \textit{nfa-update} \rangle$ ::= $\langle \textit{symbol} \rangle$ $\langle \textit{symbol} \rangle$ ',' $\langle \textit{symbol} \rangle$

The elements $\langle \textit{input-symbols} \rangle$ and $\langle \textit{states} \rangle$ are optional. If they are omitted, they are derived from the transitions. If $\langle \textit{epsilon} \rangle$ is omitted, the symbol '_' is treated as $\varepsilon$. An $\langle \textit{nfa-update} \rangle$ must be a single token, i.e. it may not contain spaces.

For example, the NFA below is specified using

```
initial 1
final 1
1 2 b
1 3 _
2 2 a
2 3 a b
3 1 a
```




**PDA syntax**

$\langle PDA \rangle$ ::= $\langle$*pda-line*$\rangle$*

$\langle$*pda-line*$\rangle$ ::= $\langle$*input-symbols*$\rangle$
| $\langle$*stack-symbols*$\rangle$
| $\langle$*epsilon*$\rangle$
| $\langle$*states*$\rangle$
| $\langle$*initial*$\rangle$
| $\langle$*final*$\rangle$
| $\langle$*pda-transition*$\rangle$

$\langle$*pda-transition*$\rangle$ ::= $\langle$*state*$\rangle$ $\langle$*state*$\rangle$ $\langle$*pda-update*$\rangle$*

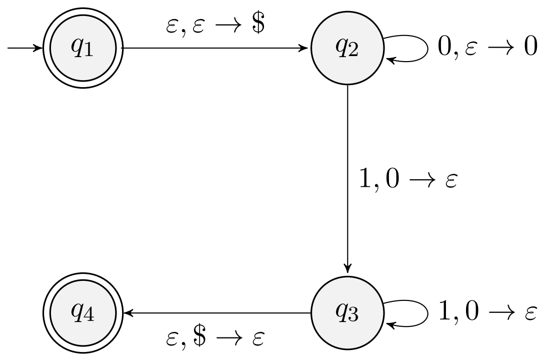
$\langle$*pda-update*$\rangle$ ::= $\langle$*symbol*$\rangle$ ',' $\langle$*symbol*$\rangle$ $\langle$*symbol*$\rangle$

The elements $\langle$*input-symbols*$\rangle$, $\langle$*stack-symbols*$\rangle$ and $\langle$*states*$\rangle$ are optional. If they are omitted, they are derived from the transitions. If $\langle$*epsilon*$\rangle$ is omitted, the symbol '_' is treated as the empty string. A $\langle$*pda-update*$\rangle$ must be a single token, i.e. it may not contain spaces.

For example, the PDA below is specified using

```
initial q1
final q1 q4
states q1 q2 q3 q4 q5
```

```
input_symbols 0 1
epsilon _
q1 q2 _,_$
q2 q2 0,_0
q2 q3 1,0_
q3 q3 1,0_
q3 q4 _,$_
```




**Turing machine syntax**

$\langle TM \rangle$ ::= $\langle$*tm-line*$\rangle$*

$\langle$*tm-line*$\rangle$ ::= $\langle$*input-symbols*$\rangle$
| $\langle$*tape-symbols*$\rangle$
| $\langle$*blank*$\rangle$
| $\langle$*states*$\rangle$
| $\langle$*initial*$\rangle$
| $\langle$*accept*$\rangle$
| $\langle$*reject*$\rangle$
| $\langle$*tm-transition*$\rangle$

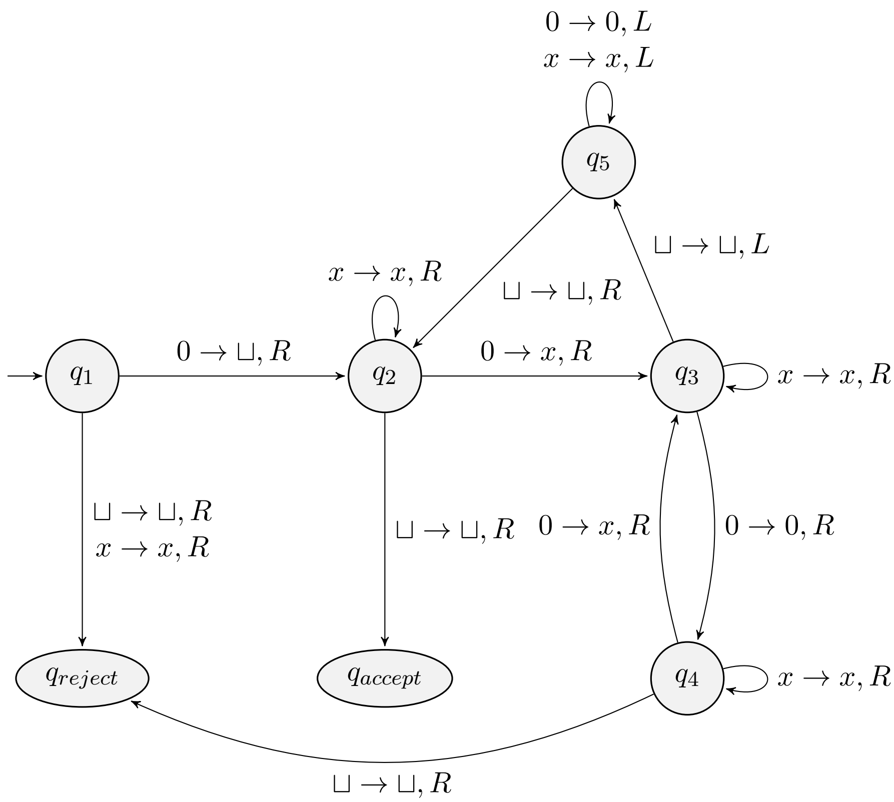
$\langle$*tm-transition*$\rangle$ ::= $\langle$*state*$\rangle$ $\langle$*state*$\rangle$ $\langle$*tm-update*$\rangle$*

$\langle$*tm-update*$\rangle$ ::= $\langle$*symbol*$\rangle$ $\langle$*symbol*$\rangle$ ',' $\langle$*direction*$\rangle$

The elements $\langle$*input-symbols*$\rangle$, $\langle$*tape-symbols*$\rangle$ and $\langle$*states*$\rangle$ are optional. If they are omitted, they are derived from the transitions. If $\langle$*blank*$\rangle$ is omitted, the symbol '_' is treated as the blank symbol. A $\langle$*tm-update*$\rangle$ must be a single token, i.e. it may not contain spaces.

For example, the Turing machine below is specified using

```
initial q1
accept q_accept
reject q_reject
input_symbols 0
tape_symbols 0 x _
blank _
q1 q2 0_,R
q1 q_reject __,R xx,R
q2 q2 xx,R
q2 q3 0x,R
q2 q_accept __,R
q3 q3 xx,R
q3 q4 00,R
q3 q5 __,L
q4 q3 0x,R
q4 q4 xx,R
q4 q_reject __,R
q5 q2 __,R
q5 q5 00,L xx,L
```




### CFG syntax

⟨*CFG*⟩ ::= ⟨*rule*⟩ (';' ⟨*rule*⟩)* ;

⟨*rule*⟩ ::= ⟨*variable*⟩ '`->`' ⟨*alternative*⟩ ('`|`' ⟨*alternative*⟩)* ;

⟨*alternative*⟩ ::= ⟨*cfg-symbol*⟩ ('`.`' ⟨*cfg-symbol*⟩)* ;

⟨*variable*⟩ ::= ⟨*cfg-identifier*⟩ ;

⟨*cfg-symbol*⟩ ::= ⟨*cfg-identifier*⟩ | '`_`' ;

A ⟨*cfg-identifier*⟩ is a token defined by the regular expression `[a-zA-Z_][a-zA-Z_0-9']*`. The left hand side of the first rule is the start variable.

### Regular expression syntax

$$
\begin{array}{lcl}
\langle \textit{regexp} \rangle & ::= & \text{‘}\mathtt{0}\text{’} \\
& | & \text{‘}\mathtt{1}\text{’} \\
& | & \langle \textit{identifier} \rangle \\
& | & \langle \textit{regexp} \rangle \ \text{‘}\mathtt{*}\text{’} \\
& | & \langle \textit{regexp} \rangle \ \text{‘}\mathtt{.}\text{’} \ \langle \textit{regexp} \rangle \\
& | & \langle \textit{regexp} \rangle \ \text{‘}\mathtt{+}\text{’} \ \langle \textit{regexp} \rangle \\
& | & \text{‘}\mathtt{(}\text{’} \ \langle \textit{regexp} \rangle \ \text{‘}\mathtt{)}\text{’}
\end{array}
$$

The productions are ordered by priority of the operators. There is a second version of the grammar in which the concatenation symbol ‘.’ is omitted.