

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.

Definition 1 *A DFA or finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with*

1. Q is a finite set called the states,
2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ . Then D accepts w if a sequence of states r_0, r_1, \dots, r_n in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n - 1$, and
3. $r_n \in F$.

Definition 2 (PDA, Sipser) A PDA or pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ with

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and let $w = w_1 w_2 \cdots w_m$ be a string where each w_i is a member of Σ_ϵ . Then P accepts w if sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that P has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \epsilon$
2. For $i = 0, \dots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$
3. $r_m \in F$

Definition 3 (PDA, Ullman) A UPDA or pushdown automaton à la Ullman is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ where

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$ is the transition function,
5. q_0 is the start state,
6. Z is the start symbol or stack bottom, and
7. F is the set of accepting states or final states

If $U = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a UPDA and let $w = w_1 \cdots w_n$ be a string with $w_i \in \Sigma_\varepsilon$ for $i = 1, \dots, n$. Then U accepts w if states r_0, \dots, r_n in Q and strings $s_0, \dots, s_n \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \varepsilon$,
2. For $i = 0, \dots, n-1$, we have $(r_{i+1}, \varrho) \in \delta(r_i, w_{i+1}, X)$, where $s_i = X\gamma$ and $s_{i+1} = \varrho\gamma$ for some $\varrho, \gamma \in \Gamma^*$, $X \in \Gamma$, and
3. $r_n \in F$.

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 Σ^* , $text$ is a string and k and n are natural numbers.

<code>dfa-accepts-word(D, w)</code>	=	$w \in \mathcal{L}(D)$
<code>dfa-words-up-to-n(D, n)</code>	=	$\{w \in \mathcal{L}(D) \mid w \leq n\}$
<code>dfa-minimize(D)</code>	=	D' a minimal DFA with $\mathcal{L}(D') = \mathcal{L}(D)$
<code>dfa-quotient(D)</code>	=	D' a minimal DFA with $\mathcal{L}(D') = \mathcal{L}(D)$
<code>dfa-identifiable(D_1, D_2)</code>	=	reachable states of D_1 and D_2 constitute isomorphic DFAs
<code>dfa-to-gnfa(D)</code>	=	G a GNFA with $\mathcal{L}(G) = \mathcal{L}(D)$
<code>dfa-to-regexp(D)</code>	=	r with $\mathcal{L}(r) = \mathcal{L}(D)$
<code>parse-dfa($text$)</code>	=	D the DFA corresponding to $text$
<code>random-dfa(Σ, n)</code>	=	D a random DFA with n states and symbols in Σ

Table 1: DFA algorithms

The function `dfa-quotient` is an implementation of the quotient algorithm as given by Lewis and Papadimitriou in [2].

<code>nfa-epsilon-closure(N, q)</code>	=	$\{q' \in Q \mid q \xrightarrow{\epsilon} q'\}$
<code>nfa-accepts-word(N, w)</code>	=	$w \in \mathcal{L}(N)$
<code>nfa-words-up-to-n(N, n)</code>	=	$\{w \in \mathcal{L}(N) \mid w \leq n\}$
<code>nfa-repetition(N)</code>	=	N' with $\mathcal{L}(N') = \mathcal{L}(N)^*$
<code>nfa-concatenation(N_1, N_2)</code>	=	N with $\mathcal{L}(N) = \mathcal{L}(N_1) \circ \mathcal{L}(N_2)$
<code>nfa-union(N_1, N_2)</code>	=	N with $\mathcal{L}(N) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$
<code>nfa-to-dfa(N)</code>	=	D with $\mathcal{L}(D) = \mathcal{L}(N)$
<code>nfa-to-dot(N)</code>	=	$text$ a graphical representation of N in dot format
<code>parse-nfa-simple($text$)</code>	=	N the NFA corresponding to $text$
<code>random-nfa(Σ, n)</code>	=	N a random NFA with n states and symbols in Σ

Table 2: NFA algorithms

<code>pda-accepts-word</code> (P, w)	=	$w \in \mathcal{L}(P)$
<code>pda-words-up-to-n</code> (P, n)	=	$\{w \in \mathcal{L}(P) \mid w \leq n\}$
<code>pda-to-cfg</code> (P)	=	G with $\mathcal{L}(P) = \mathcal{L}(G)$
<code>pda-to-push-pop</code> (P)	=	P' a PDA in push/pop format with $\mathcal{L}(P') = \mathcal{L}(P)$
<code>parse-pda-simple</code> ($text$)	=	P the PDA corresponding to $text$

Table 3: PDA algorithms

<code>tm-accepts-word</code> (T, w)	=	$w \in \mathcal{L}(T)$
<code>tm-words-up-to-n</code> (T, n)	=	$\{w \in \mathcal{L}(T) \mid w \leq n\}$
<code>parse-tm-simple</code> ($text$)	=	T the TM corresponding to $text$

Table 4: TM algorithms

<code>gnfa-minimize</code> (G)	=	G' a minimal GNFA with $\mathcal{L}(G') = \mathcal{L}(G)$
------------------------------------	---	---

Table 5: GNFA algorithms

<code>cfg-accepts-word</code> (G, w)	=	$w \in \mathcal{L}(G)$
<code>cfg-words-up-to-n</code> (G, n)	=	$\{w \in \mathcal{L}(G) \mid w \leq n\}$
<code>cfg-eliminate-epsilon-rules</code> (G)	=	G' with epsilon rules eliminated and $\mathcal{L}(G') = \mathcal{L}(G)$
<code>cfg-eliminate-unit-rules</code> (G)	=	G' with unit rules eliminated and $\mathcal{L}(G') = \mathcal{L}(G)$
<code>cfg-to-dfa</code> (G)	=	D with $\mathcal{L}(D) = \mathcal{L}(G)$
<code>cfg-to-nfa</code> (G)	=	N with $\mathcal{L}(N) = \mathcal{L}(G)$
<code>cfg-to-chomsky</code> (G)	=	G' in Chomsky normal form with $\mathcal{L}(G') = \mathcal{L}(G)$
<code>parse-cfg</code> ($text$)	=	G the CFG corresponding to $text$
<code>parse-cfg-simple</code> ($text$)	=	G the CFG corresponding to $text$

Table 6: CFG algorithms

<code>regexp-accepts-word</code> (r, w)	=	$w \in \mathcal{L}(r)$
<code>regexp-words-up-to-n</code> (r, n)	=	$\{w \in \mathcal{L}(r) \mid w \leq n\}$
<code>regexp-simplify</code> (r)	=	r' a simplified version of r with $\mathcal{L}(r') = \mathcal{L}(r)$
<code>regexp-to-nfa</code> (r)	=	N with $\mathcal{L}(N) = \mathcal{L}(r)$
<code>parse-regexp</code> ($text$)	=	r the regular expression corresponding to $text$
<code>parse-regexp-simple</code> ($text$)	=	r the regular expression corresponding to $text$
<code>random-regexp</code> (Σ, n)	=	r a random regexp of size n and symbols in Σ

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):

```
1:  $q := q_0$ 
2: for  $a \in w$  do
3:    $q := \delta(q, a)$ 
4: 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)

```
1:  $words := \emptyset$ 
2: if  $q_0 \in F$  then
3:    $words := words \cup \{\varepsilon\}$ 
4:  $W := \{(q_0, \varepsilon)\}$ 
5: for  $i \in [0 \dots n)$  do
6:    $W' := \emptyset$ 
7:   for  $(q, word) \in W$  do
8:     for  $a \in \Sigma$  do
9:        $q' := \delta(q, a)$ 
10:       $word' := word ++ a$ 
11:       $W' := W' \cup \{(q', word')\}$ 
12:      if  $q' \in F$  then
13:         $words := words \cup \{word'\}$ 
14:    $W := W'$ 
15: 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 $\delta : Q \setminus \{q_{accept}\} \times Q \setminus \{q_{start}\} \rightarrow RE_{\Sigma}$. Thus, q_{start} has no incoming transitions, q_{accept} has no outgoing transitions, while transitions are labelled with regular expressions over the alphabet Σ of the GNFA.

As usual, the class RE_{Σ} of regular expressions over an alphabet Σ 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):

```
1:  $Q' := Q \cup \{q_{start}, q_{accept}\}$ 
2: for  $q \in Q'$  do ▷ only connect  $q_{start}$  with  $q_0$ 
3:   if  $q = q_0$  then
4:      $\delta'(q_{start}, q) := 1$ 
5:   else
6:      $\delta'(q_{start}, q) := 0$ 
7: for  $(q, q') \in Q \times Q$  do
8:    $\delta'(q, q') = +\{\text{regex}(a) \mid \delta(q, a) = q'\}$ 
9: for  $q \in Q'$  do ▷ connect each final state of  $D$  to  $q_{accept}$ 
10:  if  $q \in F$  then
11:     $\delta'(q, q_{accept}) := 1$ 
12:  else
13:     $\delta'(q, q_{accept}) := 0$ 
14: 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):

```
1: for  $q_{rip} \in Q - \{q_{start}, q_{accept}\}$  do
2:    $Q := Q \setminus \{q_{rip}\}$ 
3:   for  $q_i \in Q - \{q_{accept}\}$  do
4:     for  $q_j \in Q - \{q_{start}\}$  do
5:        $\delta(q_i, q_j) := \text{regex-simplify}(\delta(q_i, q_{rip}) \cdot \delta(q_{rip}, q_{rip})^* \cdot \delta(q_{rip}, q_j) + \delta(q_i, q_j))$ 
6:    $\delta' := \{:\}$ 
7:    $\delta'(q_{start}, q_{accept}) := \delta(q_{start}, q_{accept})$ 
8:    $\delta'(q_{start}, q_{start}) := 0$ 
9:    $\delta'(q_{accept}, q_{accept}) := 0$ 
10:   $\delta'(q_{accept}, q_{start}) := 0$ 
11: 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: $G := \text{dfa-to-gnfa}(D)$

2: $\text{gnfa-minimize}(G)$

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 $Q = \{q_0, \dots, q_{n-1}\}$

Output DFA D' , equivalent to D and minimal in number of states

dfa-minimize(D):

```

1:  $\triangleright$  initialization of the lower triangle of table, a  $[0..n-1] \times [0..n-1]$  matrix
2: for  $0 \leq i \leq j < n$  do  $\triangleright$  Note  $i \leq j$  rather than  $i < j$ 
3:   table[i,j] := ( $q_i \in F \iff q_j \in F$ )
4:
5:  $\triangleright$  comparing pairs of states on (bounded) equivalence,
6:  $\triangleright$  until no change occurs
7: changed := true
8: while changed do
9:   changed := false
10:  for  $0 \leq i < j < n$  do
11:    if table[i,j] then
12:      for  $a \in \Sigma$  do
13:        let  $q_k = \delta(q_i, a)$ 
14:        let  $q_\ell = \delta(q_j, a)$ 
15:        if  $\neg \text{table}[\min\{k, \ell\}, \max\{k, \ell\}]$  then  $\triangleright k$  and  $\ell$  may be equal
16:          table[i,j] := false
17:          changed := true
18:          break  $\triangleright$  skip remaining symbols in  $\Sigma$ 
19:  $D' := \text{dfa-from-table}(D, \text{table})$ 
20: return  $D'$ 

```

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 $Q = \{q_0, \dots, q_{n-1}\}$ and Boolean matrix table
If $\text{table}[i,j] = \text{true}$, for any $0 < i < j < n$, then states q_i and q_j can be identified

dfa-from-table(D, table):

```
1:  $\triangleright$  form sets  $Q_i$  of states that can be identified according to the table
2:  $\triangleright R$  is used to see if a state is already included
3:  $R := \emptyset$ 
4: for  $0 \leq i < n$  do
5:   if  $q_i \notin R$  then
6:      $Q_i := \{q_i\}$ 
7:      $R := R \cup \{q_i\}$ 
8:   for  $i < j < n$  do
9:     if  $\text{table}[i,j]$  then
10:       $Q_i := Q_i \cup \{q_j\}$ 
11:       $R := R \cup \{q_j\}$ 
12:
13:  $\triangleright$  constructing minimal DFA  $D'$ 
14:  $Q' = \{Q_i \mid 0 \leq i < n\}$ 
15: let  $Q_0$  such that  $q_0 \in Q_0$ 
16: for  $Q_i \in Q', a \in \Sigma$  do
17:    $q' := \delta(q_i, a)$ 
18:   let  $j$  such that  $q' \in Q_j$ 
19:    $\delta'(Q_i, a) := Q_j$ 
20:  $F' = \{Q_i \mid q_i \in F\}$ 
21:
22: 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 $D_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$

dfa-isomorphic(D_1, D_2):

```
1:  $\triangleright$  maintain boolean matrix matching
2:  $\triangleright$  and a set of pairs of states to – inspect

3: for  $q_1 \in Q_1$  and  $q_2 \in Q_2$  do
4:   matching( $q_1, q_2$ ) := false

5: if  $q_0^1 \in F_1 \iff q_0^2 \in F_2$  then
6:   matching( $q_0^1, q_0^2$ ) := true
7:   to – inspect :=  $\{(q_0^1, q_0^2)\}$ 
8: else
9:   return false

10:  $\triangleright$  explore DFA  $D_1$ 
11: while to – inspect  $\neq \emptyset$  do
12:   choose  $(q_1, q_2) \in \textit{to – inspect}$ 
13:   for  $a \in \Sigma$  do
14:      $q'_1 := \delta_1(q_1, a)$ ,  $q'_2 := \delta_2(q_2, a)$ 
15:     if matching( $q'_1, q'_2$ ) then
16:       if  $q'_1 \in F_1 \iff q'_2 \in F_2$  then
17:         matching( $q'_1, q'_2$ ) := true
18:         to – inspect :=  $\{(q'_1, q'_2)\}$ 
19:       else
20:         return false

21:  $\triangleright$  check if each state of  $D_1$  has a unique matching state
22: for  $q_1 \in Q_1$  do
23:   count := 1
24:   for  $q_2 \in Q_2$  do
25:     if matching( $q_1, q_2$ ) then
26:       count += 1
27:   if count  $\neq 1$  then
28:     return false

29:  $\triangleright$  check if each state of  $D_2$  has a unique matching state
30: for  $q_2 \in Q_2$  do
31:   count := 1
32:   for  $q_1 \in Q_1$  do
33:     if matching( $q_1, q_2$ ) then
34:       count += 1
35:   if count  $\neq 1$  then
36:     return false

37:  $\triangleright$  DFAs are isomorphic if a proper matching exists
38: return true
```

3.3 Minimization of a DFA using the quotient relation

Algorithm 9 Minimizing a DFA using the quotient equivalence relation

Input DFA $D = (Q, \Sigma, \delta, q_0, F)$: with $Q = \{q_0, \dots, q_{n-1}\}$

Output DFA D' , equivalent to D and minimal in number of states

dfa-quotient(D):

```
1:  $\mathcal{V} := \{F, Q \setminus F\}$  ▷  $\mathcal{V}$  is the current set of equivalence classes
2:  $eq := \{\mapsto\}$  ▷ Maintain a mapping from states to equivalence classes in  $\mathcal{V}$ 
3: while true do
4:   for  $V \in \mathcal{V}$  do
5:     for  $v \in V$  do
6:        $eq[v] := V$ 
7:    $\mathcal{V}' := \emptyset$  ▷  $\mathcal{V}'$  is the next set of equivalence classes
8:   for  $V \in \mathcal{V}$  do
9:      $\mathcal{W} := \emptyset$  ▷  $V$  is split into equivalence classes  $\mathcal{W}$ 
10:    for  $v \in V$  do
11:      matched := false
12:      for  $W \in \mathcal{W}$  do
13:        let  $w \in W$ 
14:        if  $\forall a \in \Sigma : eq[\delta(v, a)] = eq[\delta(w, a)]$  then
15:           $W := W \cup \{v\}$ 
16:          matched := true
17:          break
18:        if  $\neg$ matched then
19:           $\mathcal{W} := \mathcal{W} \cup \{\{v\}\}$  ▷ Introduce a new equivalence class for  $v$ 
20:       $\mathcal{V}' := \mathcal{V}' \cup \mathcal{W}$ 
21:    if  $\mathcal{V} \neq \mathcal{V}'$  then
22:       $\mathcal{V} := \mathcal{V}'$ 
23:    else
24:      break
25:  $\delta' := \{\mapsto\}$ 
26: for  $V \in \mathcal{V}$  do
27:   let  $v \in V$ 
28:   for  $a \in \Sigma$  do
29:      $\delta'(V, a) := eq[\delta(v, a)]$ 
30:  $F' := \{V \in \mathcal{V} \mid V \cap F \neq \emptyset\}$ 
31:  $q'_0 := eq[q_0]$ 
32: return  $D' = (\mathcal{V}, \Sigma, \delta', q'_0, F')$ 
```

3.4 Minimization of a DFA using the Hopcroft algorithm

In [1] the following DFA minimization algorithm is given. Note that we have slightly modified the code, to get rid of the calls to unspecified functions.

Algorithm 10 Minimizing a DFA using the Hopcroft algorithm

Input DFA $D = (Q, \Sigma, \delta, q_0, F)$: with $Q = \{q_0, \dots, q_{n-1}\}$

Output DFA D' , equivalent to D and minimal in number of states

HOPCROFT(D):

```

1:  $\mathcal{P} := \{F, Q \setminus F\} \setminus \{\emptyset\}$  ▷ The initial partition
2:  $\mathcal{W} := \emptyset$  ▷ The waiting set
3: for  $a \in \Sigma$  do
4:    $\mathcal{W} := \mathcal{W} \cup \{(\min(F, Q \setminus F), a)\}$  ▷ Initialization of the waiting set
5: while  $\mathcal{W} \neq \emptyset$  do
6:   let  $(W, a) \in \mathcal{W}$  ▷ Take and remove some splitter
7:    $\mathcal{W} := \mathcal{W} \setminus \{(W, a)\}$ 
8:    $\mathcal{P}_{\text{copy}} := \mathcal{P}$  ▷ A copy is made, since  $\mathcal{P}$  is modified in the loop
9:   for  $P \in \mathcal{P}_{\text{copy}}$  do
10:     $P', P'' := (W, a) \mid P$  ▷ Compute the split
11:    if  $P' = \emptyset \vee P'' = \emptyset$  then
12:      continue
13:     $\mathcal{P} := (\mathcal{P} \setminus \{P\}) \cup \{P', P''\}$  ▷ Refine the partition
14:    for  $b \in \Sigma$  do ▷ Update the waiting set
15:      if  $(P, b) \in \mathcal{W}$  then
16:         $\mathcal{W} := (\mathcal{W} \setminus \{(P, b)\}) \cup \{(P', b), (P'', b)\}$ 
17:      else
18:         $\mathcal{W} := \mathcal{W} \cup \{(\min(P', P''), b)\}$ 
19:  $\mathcal{Q} := \mathcal{P}$  ▷ Construct the result
20:  $\delta' := \{(Q', a) \mapsto Q'' \mid Q', Q'' \in \mathcal{Q} \wedge Q' \xrightarrow{a} Q''\}$ 
21:  $Q_0 := Q \in \mathcal{Q}$  such that  $q_0 \in Q$ 
22:  $\mathcal{F} := \{Q \in \mathcal{Q} \mid Q \cap F \neq \emptyset\}$ 
23: return  $D' = (\mathcal{Q}, \Sigma, \delta', Q_0, \mathcal{F})$ 

```

with

$$\begin{aligned}
\min(\{P, Q\}) &= \begin{cases} P & \text{if } (Q = \emptyset) \vee (|P| \leq |Q| \wedge |P| \neq \emptyset) \\ Q & \text{otherwise} \end{cases} \\
p \xrightarrow{a} Q &= \exists q \in Q : p \xrightarrow{a} q \\
(W, a) \mid P &= (\{w \in W \mid w \xrightarrow{a} P\}, W \setminus \{w \in W \mid w \xrightarrow{a} P\})
\end{aligned}$$

4 NFA algorithms

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

Algorithm 11 Epsilon closure

Input: $N = (Q, \Sigma, \delta, q_0, F)$: an NFA; $q \in Q$: a state

Output: $\{q' \in Q \mid q \xrightarrow{\varepsilon} q'\}$

nfa-epsilon-closure(N, q):

```
1: result := {q}
2: todo := {q}
3: while todo  $\neq \emptyset$  do
4:   q := todo.pop()  $\triangleright$  pop removes and returns an arbitrary element of a set
5:    $Q_1 := \delta(q, \varepsilon) \setminus \textit{result}$ 
6:   result := result  $\cup Q_1$ 
7:   todo := todo  $\cup Q_1$ 
8: return result
```

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

$$\text{nfa-epsilon-closure}(N, Q) = \bigcup \{\text{nfa-epsilon-closure}(N, q) \mid q \in Q\}.$$

Algorithm 12 Test if an NFA accepts a given word

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

Output: $w \in \mathcal{L}(N)$

nfa-accepts-word(N, w):

```
1: q := nfa-epsilon-closure( $N, q_0$ )
2: for a  $\in w$  do
3:    $q := \bigcup \{\text{nfa-epsilon-closure}(N, \delta(q_i, a)) \mid q_i \in q\}$ 
4: return  $q \cap F \neq \emptyset$ 
```

Algorithm 13 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)

```

1:  $F_1 := \{q \in Q \mid \text{nfa-epsilon-closure}(N, q) \cap F \neq \emptyset\}$     ▷ states that can terminate
2:  $result := \emptyset$ 
3: if  $q_0 \in F_1$  then
4:    $result := result \cup \{\varepsilon\}$ 
5:  $W := \{(q, \varepsilon) \mid q \in \text{nfa-epsilon-closure}(N, q_0)\}$ 
6: for  $0 \leq i < n$  do
7:    $W' := \{\mapsto\}$ 
8:   for  $(q, words) \in W$  do
9:     for  $a \in \Sigma$  do
10:      for  $q_1 \in \text{nfa-epsilon-closure}(N, \delta(q, a))$  do
11:         $words' := \{wa \mid w \in words\}$ 
12:         $W'(q_1) := W'(q_1) \cup words'$ 
13:        if  $q_1 \in F_1$  then
14:           $result := result \cup words'$ 
15:    $W := W'$ 
16: return  $result$ 

```

Algorithm 14 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):

```
1:  $F := \emptyset$ 
2:  $Q_0 := \text{nfa-epsilon-closure}(N, q_N)$ 
3:  $Q := \{Q_0\}$ 
4:  $\delta := \{:\}$   $\triangleright \{:\}$  is the empty mapping
5: if  $Q_0 \cap F_N \neq \emptyset$  then
6:    $F := F \cup \{Q_0\}$ 
7:  $todo := [Q_0]$ 
8: while  $todo \neq \emptyset$  do
9:    $Q_1 := todo[0]$ 
10:   $todo := todo[1..]$ 
11:  for  $a \in \Sigma$  do
12:     $Q_2 := \emptyset$ 
13:    for  $q_1 \in Q_1$  do
14:       $Q_2 := Q_2 \cup \text{nfa-epsilon-closure}(N, \delta_N(q_1, a))$ 
15:       $\delta(Q_1, a) := Q_2$ 
16:      if  $Q_2 \cap F_N \neq \emptyset$  then
17:         $F := F \cup \{Q_2\}$ 
18:      if  $Q_2 \notin Q$  then
19:         $Q := Q \cup \{Q_2\}$ 
20:         $todo := todo ++ [Q_2]$ 
21: return  $(Q, \Sigma, \delta, Q_0, F)$ 
```

Algorithm 15 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):

```
1:  $q'_0 := \text{fresh-state}()$ 
2:  $Q' := Q \cup \{q_0\}$ 
3:  $F' := F \cup \{q_0\}$ 
4:  $\delta' := \delta$ 
5: for  $q \in F$  do
6:    $\delta'(q, \varepsilon) := \delta'(q, \varepsilon) \cup \{q_0\}$ 
7:  $\delta'(q'_0, \varepsilon) := \{q_0\}$ 
8: return  $N' = (Q', \Sigma, \delta', q'_0, F')$ 
```

Algorithm 16 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):

- 1: $q'_0 := \text{fresh-state}()$
 - 2: $Q' := Q_1 \cup Q_2 \cup \{q_0\}$
 - 3: $F' := F_1 \cup F_2$
 - 4: $\delta' := \delta_1 \cup \delta_2$
 - 5: $\delta'(q'_0, \varepsilon) := \{q_1, q_2\}$
 - 6: **return** $N' = (Q', \Sigma, \delta', q'_0, F')$
-

Algorithm 17 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):

- 1: $q'_0 := q_1$
 - 2: $Q' := Q_1 \cup Q_2 \cup \{q_0\}$
 - 3: $F' := F_2$
 - 4: $\delta' := \delta_1 \cup \delta_2$
 - 5: **for** $q \in F_1$ **do**
 - 6: $\delta'(q, \varepsilon) := \delta'(q, \varepsilon) \cup \{q_2\}$
 - 7: **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:

$$\begin{aligned} \text{can-pop-push}(stack, u, v) &= (u = \varepsilon) \vee (u \neq \varepsilon \wedge stack = stack' ++ [u]) \\ \text{pop-push}(stack, u, v) &= \begin{cases} stack & \text{if } u = \varepsilon \text{ and } v = \varepsilon \\ stack ++ [v] & \text{if } u = \varepsilon \text{ and } v \neq \varepsilon \\ stack[:|stack|-1] & \text{if } u \neq \varepsilon \text{ and } v = \varepsilon \\ stack[:|stack|-1] ++ [v] & \text{if } u \neq \varepsilon \text{ and } v \neq \varepsilon \end{cases} \end{aligned}$$

Algorithm 18 PDA epsilon closure

Input: $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$: a PDA; $r \in Q \times \Gamma^*$: a PDA state

Output: $\{r' \in Q \times \Gamma^* \mid r \xrightarrow{\varepsilon} r'\}$

pda-epsilon-closure(P, r):

```

1:  $result := \{r\}$ 
2:  $todo := \{r\}$ 
3: while  $todo \neq \emptyset$  do
4:    $r := todo.pop()$   $\triangleright pop$  removes and returns an arbitrary element of a set
5:   let  $r = (p, stack)$ 
6:   for  $u \in \Gamma_\varepsilon$  do
7:      $Q_1 := \delta(p, \varepsilon, u)$ 
8:     for  $(q, v) \in Q_1$  do
9:       if can-pop-push( $stack, u, v$ ) then
10:         $stack' = \text{pop-push}(stack, u, v)$ 
11:         $r' := (q, stack')$ 
12:        if  $r' \notin result$  then
13:           $todo := todo \cup \{r'\}$ 
14:           $result := result \cup \{r'\}$ 
15: return  $result$ 

```

We generalize the functions **pda-epsilon-closure** and **pda-do-transition** to a set of PDA states R by

$$\begin{aligned} \text{pda-epsilon-closure}(P, R) &= \bigcup \{ \text{pda-epsilon-closure}(P, r) \mid r \in R \} \\ \text{pda-do-transition}(P, a, R) &= \bigcup \{ \text{pda-do-transition}(P, a, r) \mid r \in R \}. \end{aligned}$$

Algorithm 19 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):

```
1: let  $r = (p, stack)$ 
2:  $result := \emptyset$ 
3: for  $u \in \Gamma_\varepsilon$  do
4:    $Q_1 := \delta(p, a, u)$ 
5:   for  $(q, v) \in Q_1$  do
6:     if can-pop-push( $stack, u, v$ ) then
7:        $stack' = \text{pop-push}(stack, u, v)$ 
8:        $r' := (q, stack')$ 
9:        $result := result \cup \{r'\}$ 
10: return  $result$ 
```

Algorithm 20 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):

```
1:  $R := \{(q_0, [])\}$ 
2:  $R := \text{pda-epsilon-closure}(P, R)$ 
3: for  $a \in w$  do
4:    $R := \text{pda-do-transition}(P, a, R)$ 
5:    $R := \text{pda-epsilon-closure}(P, R)$ 
6: return  $\exists (q, stack) \in R : q \in F$ 
```

Algorithm 21 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)

```
1:  $result := \emptyset$ 
2:  $W := \{\mapsto\}$ 
3:  $R := \{(q_0, [])\}$ 
4:  $R := \text{pda-epsilon-closure}(P, R)$ 
5: for  $r \in R$  do
6:    $W(r) := \{\varepsilon\}$ 
7:   let  $r = (q, stack)$ 
8:   if  $q \in F$  then
9:      $result := result \cup \{\varepsilon\}$ 
10: for  $0 \leq i < n$  do
11:    $W' := \{\mapsto\}$ 
12:   for  $(r, words) \in W$  do
13:     for  $a \in \Sigma$  do
14:        $R := \text{pda-do-transition}(P, a, r)$ 
15:        $R := \text{pda-epsilon-closure}(P, R)$ 
16:        $words' := \{wa \mid w \in words\}$ 
17:       for  $r' \in R$  do
18:          $W'(r') := W'(r') \cup words'$ 
19:         let  $r' = (q', stack')$ 
20:         if  $q' \in F$  then
21:            $result := result \cup words'$ 
22:    $W := W'$ 
23: return  $result$ 
```

Algorithm 22 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):

```

1:  $V := \{A_{pq} \mid (p, q) \in Q \times Q\}$ 
2:  $R := \emptyset$ 
3: let  $F = \{q_{accept}\}$ 
4:  $S := A_{q_0 q_{accept}}$ 
5:  $T_{push} := \{\mapsto\}$  ▷ maps stack symbols to corresponding push transitions
6:  $T_{pop} := \{\mapsto\}$  ▷ maps stack symbols to corresponding pop transitions
7: for  $((p, a, u), Q_1) \in \delta$  do
8:   for  $(q, v) \in Q_1$  do
9:     if  $u = \varepsilon$  then
10:        $T_{push}[v] := T_{push}[v] \cup \{(p, a, \varepsilon) \rightarrow (q, v)\}$ 
11:     else if  $v = \varepsilon$  then
12:        $T_{pop}[u] := T_{pop}[u] \cup \{(p, a, u) \rightarrow (q, \varepsilon)\}$ 
13: for  $u \in \Gamma$  do
14:   for  $((p, a, \varepsilon) \rightarrow (r, u), (s, b, u) \rightarrow (q, \varepsilon)) \in T_{push}[u] \times T_{pop}[u]$  do
15:      $R := R \cup \{A_{pq} \rightarrow aA_{rs}b\}$ 
16: for  $(p, q, r) \in Q \times Q \times Q$  do
17:    $R := R \cup \{A_{pq} \rightarrow A_{pr}A_{rq}\}$ 
18: for  $p \in Q$  do
19:    $R := R \cup \{A_{pp} \rightarrow \varepsilon\}$ 
20: return  $G = (V, \Sigma, R, S)$ 

```

Algorithm 23 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

- 1: **let** q_s, q'_a, q_a be fresh for Q
 - 2: $Q' = Q \cup \{q_s, q'_a, q_a\}$
 - 3: \triangleright q_s new start state, q_a new single final state, q'_a additional state
 - 4: **let** $\$$ be fresh for Γ
 - 5: $\Gamma' = \Gamma \cup \{\$\}$
 - 6: \triangleright introduce stack-bottom $\$$ file
 - 7: \triangleright by convention, if not specified $\delta'(q, \alpha, X) = \emptyset$
 - 8: **for** $q \in Q, a \in \Sigma, X \in \Gamma$ **do**
 - 9: $\delta'(q, a, X) = \delta(q, a, X)$
 - 10: $\delta'(q_s, \varepsilon, \varepsilon) = \{(q_0, \$)\}$
 - 11: $\delta'(q, \varepsilon, \varepsilon) = \delta(q, \varepsilon, \varepsilon) \cup \{(q'_a, \varepsilon)\}$ for $q \in F$
 - 12: $\delta'(q'_a, \alpha, X) = \emptyset$ for $\alpha \in \Sigma, X \in \Gamma'_\varepsilon$
 - 13: $\delta'(q'_a, \varepsilon, X) = \{(q'_a, \varepsilon)\}$ for $X \in \Gamma$
 - 14: $\delta'(q'_a, \varepsilon, \$) = \{(q_a, \varepsilon)\}$
 - 15: $F' = \{q_a\}$
 - 16: **return** P'
-

Algorithm 24 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

```

1:  let  $\mathfrak{c}$  be fresh for  $\Gamma$ 
2:   $\Gamma' = \Gamma \cup \{\mathfrak{c}\}$ 
3:   $\triangleright$  remove push/pop transitions: replace  $q \xrightarrow{\alpha, X/Y} q'$  by  $q \xrightarrow{\alpha, X/\varepsilon} \hat{q} \xrightarrow{\varepsilon, \varepsilon/Y} q'$ 
4:  for  $q \in Q, \alpha \in \Sigma_\varepsilon, X \in \Gamma$  do
5:    for  $q' \in Q, Y \in \Gamma$  such that  $(q', Y) \in \delta(q, \alpha, X)$  do
6:      let  $\hat{q}$  be fresh for  $Q$ 
7:       $Q = Q \cup \{\hat{q}\}$ 
8:       $\delta(q, \alpha, X) = (\delta(q, \alpha, X) \setminus \{(q', Y)\}) \cup \{(\hat{q}, \varepsilon)\}$ 
9:       $\delta(\hat{q}, \varepsilon, \varepsilon) = \{(q', Y)\}$ 
10:      $\triangleright$  implicitly  $\delta(\hat{q}, a, Z) = \emptyset$  for  $a \in \Sigma, Z \in \Gamma_\varepsilon$ 
11:   $\triangleright$  remove  $\varepsilon/\varepsilon$ -transitions: replace  $q \xrightarrow{\alpha, \varepsilon, \varepsilon} q'$  by  $q \xrightarrow{\alpha, \varepsilon, \mathfrak{c}} \hat{q} \xrightarrow{\varepsilon, \mathfrak{c}/\varepsilon} q'$ 
12:  for  $q \in Q, \alpha \in \Sigma_\varepsilon$  do
13:    for  $q' \in Q$  such that  $(q', \varepsilon) \in \delta(q, \alpha, \varepsilon)$  do
14:      let  $\hat{q}$  be fresh for  $Q$ 
15:       $Q = Q \cup \{\hat{q}\}$ 
16:       $\delta(q, \alpha, \varepsilon) = (\delta(q, \alpha, \varepsilon) \setminus \{(q', \varepsilon)\}) \cup \{(\hat{q}, \mathfrak{c})\}$ 
17:       $\delta(\hat{q}, \varepsilon, \mathfrak{c}) = \{(q', \varepsilon)\}$ 
18:      $\triangleright$  implicitly  $\delta(\hat{q}, a, Z) = \emptyset$  for  $a \in \Sigma, Z \in \Gamma_\varepsilon$ 
19:   $\Gamma = \Gamma'$ 
20:  return  $P$ 

```

6 TM algorithms

Algorithm 25 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($T, p, tape, head$):

```
1:  $a := tape[head]$ 
2: if  $(p, a) \in \text{domain}(\delta)$  then
3:    $(q, b, d) := \delta(p, a)$ 
4: else
5:    $(q, b, d) := (q_{reject}, a, R)$ 
6:  $tape[head] := b$ 
7: if  $d = L$  then
8:    $head' := \max(head - 1, 0)$ 
9: else if  $d = R$  then
10:   $head' := head + 1$ 
11: if  $|tape| = head'$  then
12:    $tape := tape ++ [\sqcup]$  ▷ add a blank, so that  $tape[head]$  is defined
13: return  $(q, head')$ 
```

Algorithm 26 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):

```
1:  $q := q_0$ 
2:  $tape := w$ 
3:  $head := 0$ 
4: if  $|tape| = 0$  then
5:    $tape := tape ++ [\sqcup]$  ▷ add a blank, so that  $tape[head]$  is defined
6: while true do
7:    $(q, head) := \text{tm-do-transition}(T, q, tape, head)$ 
8:   if  $q == q_{accept}$  then
9:     return true
10:  if  $q == q_{reject}$  then
11:    return false
```

Algorithm 27 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):

```
1:  $q := q_0$ 
2:  $tape := w$ 
3:  $head := 0$ 
4: if  $|tape| = 0$  then
5:    $tape := tape ++ [\sqcup]$  ▷ add a blank, so that  $tape[head]$  is defined
6:  $result := [(q, tape, head)]$ 
7: while true do
8:    $(q, head) := \text{tm-do-transition}(T, q, tape, head)$ 
9:    $result := result ++ [(q, tape, head)]$ 
10:  if  $q == q_{accept}$  then
11:    break
12:  if  $q == q_{reject}$  then
13:    break
14: return result
```

Algorithm 28 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)

```
1: result :=  $\emptyset$ 
2: for  $0 \leq i \leq n$  do
3:   for  $w \in \Sigma^n$  do
4:     if tm-accepts-word( $T, w$ ) then
5:       words := words  $\cup \{w\}$ 
```

7 CFG algorithms

Algorithm 29 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):

```

1: if  $w = \varepsilon$  then
2:   return  $S \rightarrow \varepsilon \in R$ 
3:  $n := |w|$ 
4:  $X := \{\mapsto\}$   $\triangleright$  initially  $X(i, j) = \emptyset$  for all  $i, j$ 
5:  $\triangleright X(i, j)$  will contain  $\{A \in V \mid A \xRightarrow{*} w_i w_{i+1} \dots w_j\}$ 
6: for  $0 \leq i < n$  do
7:    $X(i, i) := \{A \in V \mid A \rightarrow w_i \in R\}$ 
8: for  $1 \leq m < n$  do
9:   for  $0 \leq i < n - m$  do
10:     $j := i + m$ 
11:    for  $i \leq k \leq j$  do
12:      for  $(B, C) \in X(i, k) \times X(k + 1, j)$  do
13:         $X(i, j) := X(i, j) \cup \{A \in V \mid A \rightarrow BC \in R\}$ 
14: return  $S \in X(0, n - 1)$ 

```

7.1 Generating the language corresponding to a CFG

Given the grammar $G = (V, \Sigma, R, S)$, we define $R_n = \{A \rightarrow x \mid |x| = n\}$, 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 30 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)

```

1:  $words := \emptyset$ 
2: if  $S \rightarrow \varepsilon \in R$  then
3:    $words := words \cup \{\varepsilon\}$ 
4:  $W := [S]$ 
5:  $words := words \cup \{w \in \Sigma^* \mid \exists x \in W : x \Rightarrow_{G_1}^* w\}$ 
6: for  $2 \leq i \leq n$  do
7:    $W := \{w \in \Sigma^* \mid \exists x \in W : x \Rightarrow_{G_2} w\}$ 
8:    $words := words \cup \{w \in \Sigma^* \mid \exists x \in W : x \Rightarrow_{G_1}^* w\}$ 
9: 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 $A \rightarrow B$, where A and B are variables.

Algorithm 31 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)

```

1:  $W := \emptyset$ 
2:  $W' := \emptyset$ 
3: for  $A \rightarrow B \in R$  with  $B \in V$  do
4:    $W := W \cup \{B\}$ 
5: while  $W' \neq W$  do
6:    $W' := W$ 
7:   for  $C \rightarrow B \in R$  with  $C \in W'$  and  $B \in V$  do
8:      $W := W \cup \{B\}$ 
9: return  $W \setminus \{A\}$ 

```

Algorithm 32 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)

```
1:  $R' := R$ 
2: for  $A \in V$  do
3:    $W := \text{cfg-derivable-variables}(G, A)$ 
4:   for  $B \rightarrow \alpha \in R$  with  $B \in W$  and  $\alpha \notin V$  do
5:      $R' := R' \cup \{A \rightarrow \alpha\}$  ▷ Make sure not to add duplicate rules
6:  $R' := \{A \rightarrow \alpha \in R' \mid \alpha \notin V\}$ 
7: return  $G' = (V, \Sigma, R', S)$ 
```

Algorithm 33 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)

```
1:  $W := \{S\}$ 
2: while true do
3:    $W' := \{A \in V \setminus W \mid \exists B \rightarrow xAy \in R \text{ with } B \in W \text{ and } x, y \in (V \cup \Sigma)^*\}$ 
4:    $W := W \cup W'$ 
5:   if  $W' = \emptyset$  then
6:     break
7: return  $W$ 
```

Algorithm 34 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)

```
1:  $W := \Sigma$ 
2: while true do
3:    $W' := \{A \in V \setminus W \mid \exists A \rightarrow x \in R \text{ with } x \in W^*\}$ 
4:    $W := W \cup W'$ 
5:   if  $W' = \emptyset$  then
6:     break
7: return  $W \setminus \Sigma$ 
```

8 Regular expression algorithms

The functions `regex-size` and `regex-symbols` are inductively defined as

$$\begin{aligned}
 \text{regex-size}(\mathbf{0}) &= 0 \\
 \text{regex-size}(\mathbf{1}) &= 0 \\
 \text{regex-size}(a) &= 0 \\
 \text{regex-size}(r^*) &= \text{regex-size}(r) + 1 \\
 \text{regex-size}(r_1 + r_2) &= \text{regex-size}(r_1) + \text{regex-size}(r_2) + 2 \\
 \text{regex-size}(r_1 \cdot r_2) &= \text{regex-size}(r_1) + \text{regex-size}(r_2) + 2 \\
 \\
 \text{regex-symbols}(\mathbf{0}) &= \emptyset \\
 \text{regex-symbols}(\mathbf{1}) &= \emptyset \\
 \text{regex-symbols}(a) &= \{a\} \\
 \text{regex-symbols}(r^*) &= \text{regex-symbols}(r) \\
 \text{regex-symbols}(r_1 + r_2) &= \text{regex-symbols}(r_1) \cup \text{regex-symbols}(r_2) \\
 \text{regex-symbols}(r_1 \cdot r_2) &= \text{regex-symbols}(r_1) \cup \text{regex-symbols}(r_2)
 \end{aligned}$$

The functions `regex-simplify` is defined by means of the following rewrite rules

$$\begin{aligned}
 \text{regex-simplify}(\mathbf{0}^*) &= \mathbf{1} \\
 \text{regex-simplify}(\mathbf{1}^*) &= \mathbf{1} \\
 \text{regex-simplify}(r^{**}) &= r^* \\
 \text{regex-simplify}(\mathbf{0} + r) &= r \\
 \text{regex-simplify}(r + \mathbf{0}) &= r \\
 \text{regex-simplify}(\mathbf{0} \cdot r) &= \mathbf{0} \\
 \text{regex-simplify}(\mathbf{1} \cdot r) &= r \\
 \text{regex-simplify}(r \cdot \mathbf{0}) &= \mathbf{0} \\
 \text{regex-simplify}(r \cdot \mathbf{1}) &= r \\
 \\
 \text{regex-accepts-word}(\mathbf{0}) &= \emptyset \\
 \text{regex-accepts-word}(\mathbf{1}) &= \emptyset \\
 \text{regex-accepts-word}(a) &= \{a\} \\
 \text{regex-accepts-word}(r^*) &= \text{regex-accepts-word}(r) \\
 \text{regex-accepts-word}(r_1 + r_2) &= \text{regex-accepts-word}(r_1) \cup \text{regex-accepts-word}(r_2) \\
 \text{regex-accepts-word}(r_1 \cdot r_2) &= \text{regex-accepts-word}(r_1) \cup \text{regex-accepts-word}(r_2)
 \end{aligned}$$

The algorithms `regex-accepts-word` and `regex-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 35 Test if a regular expression matches a given word

Input: r : a regular expression; w a word

Output: $w \in \mathcal{L}(r)$

regex-accepts-word(r, w):

```

1: if  $r = \mathbf{0}$  then
2:   return false
3: else if  $r = \mathbf{1}$  then
4:   return  $|w| = 0$ 
5: else if  $r = a$  then
6:   return  $w = a$ 
7: else if  $r = r_1 + r_2$  then
8:   return regex-accepts-word( $r_1, w$ )  $\vee$  regex-accepts-word( $r_2, w$ )
9: else if  $r = r_1 \cdot r_2$  then
10:  return  $\exists 0 \leq k \leq |w| : \text{regex-accepts-word}(r_1, w[:k]) \wedge \text{regex-accepts-word}(r_2, w[k:])$ 
11: else if  $r = r_1^*$  then
12:   if  $r_1 = \mathbf{0}$  then
13:     return  $|w| = 0$ 
14:   else
15:     return  $\exists 1 \leq k \leq |w| : \text{regex-accepts-word}(r_1, w[:k]) \wedge \text{regex-accepts-word}(r, w[k:])$ 

```

Algorithm 36 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)

```

1: if  $r = 0$  then
2:   return  $\emptyset$ 
3: else if  $r = 1$  then
4:   return  $\{\varepsilon\}$ 
5: else if  $r = a$  then
6:   if  $n > 0$  then
7:     return  $\{a\}$ 
8:   else
9:     return  $\emptyset$ 
10: else if  $r = r_1 + r_2$  then
11:   return regexp-words-up-to-n( $r_1, n$ )  $\cup$  regexp-words-up-to-n( $r_2, n$ )
12: else if  $r = r_1 \cdot r_2$  then
13:   return  $\cup \{\text{regexp-words-up-to-n}(r_1, k) \circ \text{regexp-words-up-to-n}(r_2, n - k) \mid 0 \leq k \leq |w|\}$ 
14: else if  $r = r_1^*$  then
15:   if  $r_1 = 0$  then
16:     return  $\{\varepsilon\}$ 
17:   else
18:     return  $\cup \{\text{regexp-words-up-to-n}(r_1, k) \circ \text{regexp-words-up-to-n}(r_1, n - k) \mid 1 \leq k \leq |w|\} \cup \{\varepsilon\}$ 

```

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.

$\langle blank \rangle ::= \text{'blank'} \langle symbol \rangle$

$\langle epsilon \rangle ::= \text{'epsilon'} \langle symbol \rangle$

$\langle direction \rangle ::= \text{'L'} \mid \text{'R'}$

$\langle input_symbols \rangle ::= \text{'input_symbols'} \langle symbol \rangle^*$

$\langle stack_symbols \rangle ::= \text{'stack_symbols'} \langle symbol \rangle^*$

$\langle tape_symbols \rangle ::= \text{'tape_symbols'} \langle symbol \rangle^*$

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

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

States

$\langle states \rangle ::= \text{'states'} \langle state \rangle^*$

$\langle initial \rangle ::= \text{'initial'} \langle state \rangle$

$\langle accept \rangle ::= \text{'accept'} \langle state \rangle$

$\langle reject \rangle ::= \text{'reject'} \langle state \rangle$

$\langle final \rangle ::= \text{'final'} \langle state \rangle^*$

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

DFA syntax

$\langle DFA \rangle ::= \langle dfa_line \rangle^*$

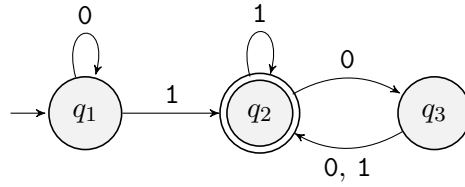
$\langle dfa_line \rangle ::= \langle input_symbols \rangle$
 $\mid \langle states \rangle$
 $\mid \langle initial \rangle$
 $\mid \langle final \rangle$
 $\mid \langle dfa_transition \rangle$

$\langle dfa-transition \rangle ::= \langle state \rangle \langle state \rangle \langle symbol \rangle$

The elements $\langle input-symbols \rangle$ and $\langle states \rangle$ 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

$\langle NFA \rangle ::= \langle nfa-line \rangle^*$

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

$\langle nfa-transition \rangle ::= \langle state \rangle \langle state \rangle \langle nfa-update \rangle^*$

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

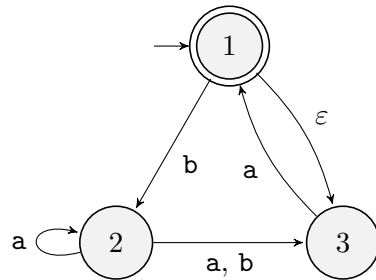
The elements $\langle input-symbols \rangle$ and $\langle states \rangle$ are optional. If they are omitted, they are derived from the transitions. If $\langle epsilon \rangle$ is omitted, the symbol '-' is treated as ϵ . An $\langle nfa-update \rangle$ must be a single token, i.e. it may not contain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

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

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

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

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

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

```

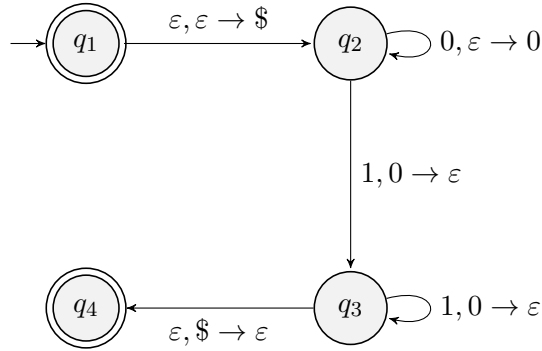
initial q1
final q1 q4
states q1 q2 q3 q4 q5

```

```

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

```



Turing machine syntax

$\langle TM \rangle ::= \langle tm\text{-}line \rangle^*$

$\langle tm\text{-}line \rangle ::= \langle input\text{-}symbols \rangle$
 $\quad \quad \quad | \langle tape\text{-}symbols \rangle$
 $\quad \quad \quad | \langle blank \rangle$
 $\quad \quad \quad | \langle states \rangle$
 $\quad \quad \quad | \langle initial \rangle$
 $\quad \quad \quad | \langle accept \rangle$
 $\quad \quad \quad | \langle reject \rangle$
 $\quad \quad \quad | \langle tm\text{-}transition \rangle$

$\langle tm\text{-}transition \rangle ::= \langle state \rangle \langle state \rangle \langle tm\text{-}update \rangle^*$

$\langle tm\text{-}update \rangle ::= \langle symbol \rangle \langle symbol \rangle ', ' \langle direction \rangle$

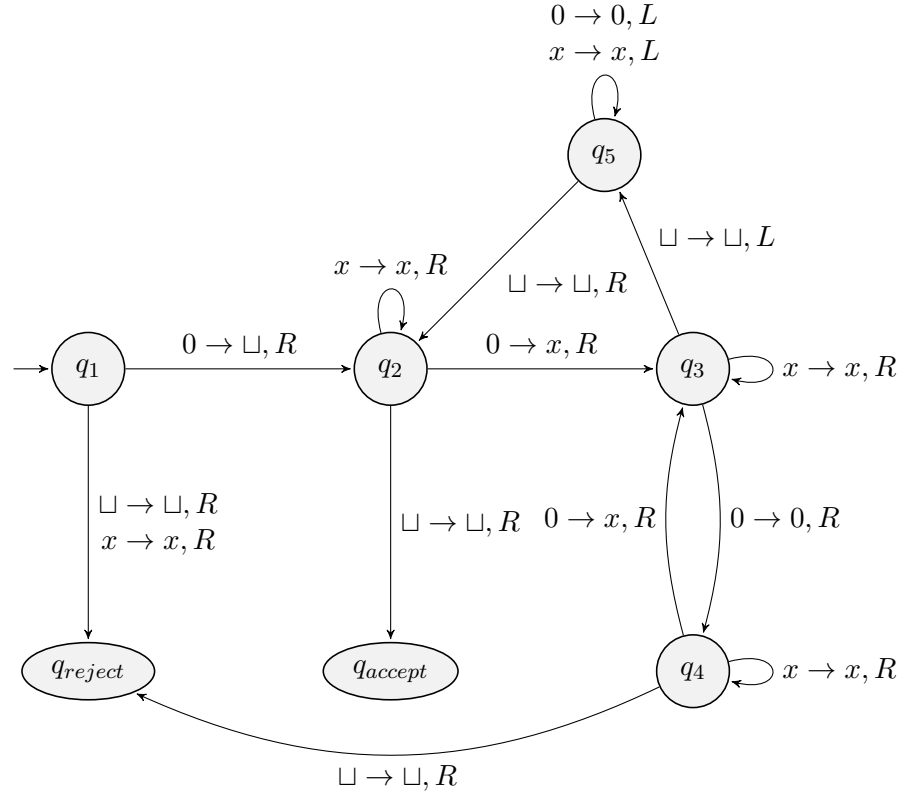
The elements $\langle input\text{-}symbols \rangle$, $\langle tape\text{-}symbols \rangle$ and $\langle states \rangle$ are optional. If they are omitted, they are derived from the transitions. If $\langle blank \rangle$ is omitted, the symbol '-' is treated as the blank symbol. A $\langle tm\text{-}update \rangle$ 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

$\langle CFG \rangle ::= \langle rule \rangle (; ' \langle rule \rangle)^*$;

$\langle rule \rangle ::= \langle variable \rangle ' \rightarrow ' \langle alternative \rangle (' | ' \langle alternative \rangle)^*$;

$\langle alternative \rangle ::= \langle cfg-symbol \rangle (' . ' \langle cfg-symbol \rangle)^*$;

$\langle variable \rangle ::= \langle cfg-identifier \rangle$;

$\langle cfg-symbol \rangle ::= \langle cfg-identifier \rangle | ' _ ' ;$

A $\langle cfg-identifier \rangle$ 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

$$\begin{array}{lcl}
\langle regexp \rangle & ::= & '0' \\
& | & '1' \\
& | & \langle identifier \rangle \\
& | & \langle regexp \rangle '*' \\
& | & \langle regexp \rangle '.' \langle regexp \rangle \\
& | & \langle regexp \rangle '+' \langle regexp \rangle \\
& | & '(' \langle regexp \rangle ')'
\end{array}$$

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

References

- [1] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.
- [2] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation, 2nd Edition*. Prentice Hall, 1998.