Specifications for the GAMBA Library

Erik de Vink and Wieger Wesselink

2019–2024

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. $\delta : Q \times \Sigma \to Q$ *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* $w = w_1 w_2 \cdots w_n$ *be a string where each* w_i *is a member of the alphabet* Σ*. Then* D *accepts* w *if a sequence of states* $r_0, r_1, \ldots, r_n$ *in* Q *exists with three conditions:*

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ *, for* $i = 0, \ldots, n-1$*, 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. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ *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* $r_0, r_1, \ldots, r_m \in Q$ *and strings* $s_0, s_1, \ldots, s_m \in \Gamma^*$ *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* $s_0 = \varepsilon$
2. *For* $i = 0, \ldots, m-1$*, 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* $a, b \in \Gamma_\varepsilon$ *and* $t \in \Gamma^*$
3. $r_m \in F$

Definition 3 (PDA, Ullman) *A UPDA or pushdown automaton à lá 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. $\delta : Q \times \Sigma_\epsilon \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ *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 $i = 1, \ldots, n$. Then U accepts w if states $r_0, \ldots, r_n$ in Q and strings $s_0, \ldots, s_n \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \varepsilon$,
2. For $i = 0, \ldots, n{-}1$, 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.

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

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

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(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):

```
q := q₀
for a ∈ w do
    q := δ(q, a)
return q ∈ 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 := ∅
if q₀ ∈ F then
    words := words ∪ {ε}
W := {(q₀, ε)}
for i ∈ [0 ... n) do
    W′ := ∅
    for (q, word) ∈ W do
        for a ∈ Σ do
            q′ := δ(q, a)
            word′ := word ++ a
            W′ := W′ ∪ {(q′, word′)}
            if q′ ∈ F then
                words := words ∪ {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 $\delta : Q \backslash \{q_{accept}\} \times Q \backslash \{q_{start}\} \to RE_\Sigma$. 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):

$Q' := Q \cup \{q_{start}, q_{accept}\}$
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**
  $\delta'(q, q') = +\{\text{regexp}(a) \mid \delta(q, a) = q'\}$
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**
      $\delta(q_i, q_j) := \text{regexp-simplify}(\delta(q_i, q_{rip}) \cdot \delta(q_{rip}, q_{rip})^* \cdot \delta(q_{rip}, q_j) + \delta(q_i, q_j))$
$\delta' := \{:\}$
$\delta'(q_{start}, q_{accept}) := \delta(q_{start}, q_{accept})$
$\delta'(q_{start}, q_{start}) := \mathbf{0}$
$\delta'(q_{accept}, q_{accept}) := \mathbf{0}$
$\delta'(q_{accept}, q_{start}) := \mathbf{0}$
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

dfa-minimize(D):

```
▷ initialization of the lower triangle of table, a [0..n−1] × [0..n−1] matrix
for 0 ⩽ i ⩽ j < n do                                  ▷ Note i ⩽ j rather than i < j
    table[i,j] := (q_i ∈ F ⟺ q_j ∈ F)

▷ comparing pairs of states on (bounded) equivalence,
▷       until no change occurs
changed := true
while changed do
    changed := false
    for 0 ⩽ i < j < n do
        if table[i,j] then
            for a ∈ Σ do
                let q_k = δ(q_i, a)
                let q_ℓ = δ(q_j, a)
                if ¬table[min{k,ℓ}, max{k,ℓ}] then    ▷ k and ℓ may be equal
                    table[i,j] := false
                    changed := true
                    break                              ▷ skip remaining symbols in Σ
D' := dfa-from-table(D,table)
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 $Q = \{q_0, \ldots, q_{n-1}\}$ 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$(D, table)$:

```
▷ form sets $Q_i$ of states that can be identified according to the table
▷ $R$ 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
                $Q_i := Q_i \cup \{q_j\}$
                $R := R \cup \{q_j\}$

▷ 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 $D_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$

dfa-isomorphic(D_1, D_2):

$\triangleright$ maintain boolean matrix *matching*
$\triangleright$ and a set of pairs of states $to - inspect$
for $q_1 \in Q_1$ and $q_2 \in Q_2$ **do**
 $matching(q_1, q_2) := false$
if $q_0^1 \in F_1 \iff q_0^2 \in F_2$ **then**
 $matching(q_0^1, q_0^2) := true$
 $to - inspect := \{(q_0^1, q_0^2)\}$
else
 return *false*
$\triangleright$ explore DFA D_1
while $to - inspect \neq \emptyset$ **do**
 choose $(q_1, q_2) \in to - inspect$
 for $a \in \Sigma$ **do**
  $q_1' := \delta_1(q_1, a)$, $q_2' := \delta_2(q_2, a)$
  if $matching(q_1', q_2')$ **then**
   if $q_1' \in F_1 \iff q_2' \in F_2$ **then**
    $matching(q_1', q_2') := true$
    $to - inspect := \{(q_1', q_2')\}$
   else
    return *false*
$\triangleright$ check if each state of D_1 has a unique matching state
for $q_1 \in Q_1$ **do**
 $count := 1$
 for $q_2 \in Q_2$ **do**
  if $matching(q_1, q_2)$ **then**
   $count \mathrel{+}= 1$
 if $count \neq 1$ **then**
  return *false*
$\triangleright$ check if each state of D_2 has a unique matching state
for $q_2 \in Q_2$ **do**
 $count := 1$
 for $q_1 \in Q_1$ **do**
  if $matching(q_1, q_2)$ **then**
   $count \mathrel{+}= 1$
 if $count \neq 1$ **then**
  return *false*
$\triangleright$ DFAs are isomorphic is a proper matching exists
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 $Q = \{q_0, \ldots, q_{n-1}\}$
Output DFA D', equivalent to D and minimal in number of states

```
dfa-quotient(D):
𝒱 := {F, Q \ F}                         ▷ 𝒱 is the current set of equivalence classes
eq := {↦}            ▷ Maintain a mapping from states to equivalence classes in 𝒱
while true do
    for V ∈ 𝒱 do
        for v ∈ V do
            eq[v] := V
    𝒱′ := ∅                              ▷ 𝒱′ is the next set of equivalence classes
    for V ∈ 𝒱 do
        𝒲 := ∅                           ▷ V is split into equivalence classes 𝒲
        for v ∈ V do
            matched := false
            for W ∈ 𝒲 do
                let w ∈ W
                if ∀a ∈ Σ : eq[δ(v, a)] = eq[δ(w, a)] then
                    W := W ∪ {v}
                    matched := true
                    break
            if ¬matched then
                𝒲 := 𝒲 ∪ {{v}}           ▷ Introduce a new equivalence class for v
        𝒱′ := 𝒱′ ∪ 𝒲
    if 𝒱 ≠ 𝒱′ then
        𝒱 := 𝒱′
    else
        break
δ′ := {↦}
for V ∈ 𝒱 do
    let v ∈ V
    for a ∈ Σ do
        δ′(V, a) := eq[δ(v, a)]
F′ := {V ∈ 𝒱 | V ∩ F ≠ ∅}
q′₀ := eq[q₀]
return D′ = (𝒱, Σ, δ′, 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 $Q = \{q_0, \ldots, q_{n-1}\}$
Output DFA D', equivalent to D and minimal in number of states

HOPCROFT(D):

$\mathcal{P} := \{F, Q \setminus F\} \setminus \{\emptyset\}$ ▷ The initial partition
$\mathcal{W} := \emptyset$ ▷ The waiting set
for $a \in \Sigma$ **do**
    $\mathcal{W} := \mathcal{W} \cup \{(\min(F, Q \setminus F), a)\}$ ▷ Initialization of the waiting set
while $\mathcal{W} \neq \emptyset$ **do**
    let $(W, a) \in \mathcal{W}$ ▷ Take and remove some splitter
    $\mathcal{W} := \mathcal{W} \setminus \{(W, a)\}$
    $\mathcal{P}_{\text{copy}} := \mathcal{P}$ ▷ A copy is made, since $\mathcal{P}$ is modified in the loop
    for $P \in \mathcal{P}_{\text{copy}}$ **do**
        $P', P'' := (W, a)|P$ ▷ Compute the split
        if $P' = \emptyset \vee P'' = \emptyset$ **then**
            continue
        $\mathcal{P} := (\mathcal{P} \setminus \{P\}) \cup \{P', P''\}$ ▷ Refine the partition
        for $b \in \Sigma$ **do** ▷ Update the waiting set
            if $(P, b) \in \mathcal{W}$ **then**
                $\mathcal{W} := (\mathcal{W} \setminus \{(P, b)\}) \cup \{(P', b), (P'', b)\}$
            else
                $\mathcal{W} := \mathcal{W} \cup \{(\min(P', P''), b)\}$
$\mathcal{Q} := \mathcal{P}$ ▷ Construct the result
$\delta' := \{(Q', a) \mapsto Q'' \mid Q', Q'' \in \mathcal{Q} \wedge Q' \xrightarrow{a} Q''\}$
$Q_0 := Q \in \mathcal{Q}$ **such that** $q_0 \in Q$
$\mathcal{F} := \{Q \in \mathcal{Q} \mid Q \cap F \neq \emptyset\}$
return $D' = (\mathcal{Q}, \Sigma, \delta', Q_0, \mathcal{F})$,

with

$$\min(\{P, Q\}) = \begin{cases} P & \text{if } (Q = \emptyset) \vee (|P| \leq |Q| \wedge |P| \neq \emptyset) \\ Q & \text{otherwise} \end{cases}$$

$$p \xrightarrow{a} Q = \exists q \in Q : p \xrightarrow{a} q$$

$$(W, a) \mid P = (\{w \in W \mid w \xrightarrow{a} P\}, W \setminus \{w \in W \mid w \xrightarrow{a} P\})$$

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):
$result := \{q\}$
$todo := \{q\}$
while $todo \neq \emptyset$ **do**
    $q := todo.pop()$ $\triangleright$ *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

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

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):
$q :=$ nfa-epsilon-closure(N, q_0)
for $a \in w$ **do**
    $q := \bigcup\{\text{nfa-epsilon-closure}(N, \delta(q_i, a)) \mid q_i \in q\}$
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: $\{w \in \mathcal{L}(N) \mid |w| \leq n\}$

$\textsf{nfa-words-up-to-n}(N, n)$

$F_1 := \{q \in Q \mid \textsf{nfa-epsilon-closure}(N, q) \cap F \neq \emptyset\}$ ▷ states that can terminate
$result := \emptyset$
if $q_0 \in F_1$ **then**
    $result := result \cup \{\varepsilon\}$
$W := \{(q, \varepsilon) \mid q \in \textsf{nfa-epsilon-closure}(N, q_0)\}$
for $0 \leq i < n$ **do**
    $W' := \{\mapsto\}$
    for $(q, words) \in W$ **do**
        for $a \in \Sigma$ **do**
            for $q_1 \in \textsf{nfa-epsilon-closure}(N, \delta(q, a))$ **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 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):
F := ∅
Q_0 := nfa-epsilon-closure(N, q_N)
Q := {Q_0}
δ := {:}                                   ▷ {:} is the empty mapping
if Q_0 ∩ F_N ≠ ∅ then
    F := F ∪ {Q_0}
todo := [Q_0]
while todo ≠ ∅ do
    Q_1 := todo[0]
    todo := todo[1..]
    for a ∈ Σ do
        Q_2 := ∅
        for q_1 ∈ Q_1 do
            Q_2 := Q_2 ∪ nfa-epsilon-closure(N, δ_N(q_1, a))
        δ(Q_1, a) := Q_2
        if Q_2 ∩ F_N ≠ ∅ then
            F := F ∪ {Q_2}
        if Q_2 ∉ Q then
            Q := Q ∪ {Q_2}
            todo := todo ++ [Q_2]
return (Q, Σ, δ, 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):
q'_0 := fresh-state()
Q' := Q ∪ {q_0}
F' := F ∪ {q_0}
δ' := δ
for q ∈ F do
    δ'(q, ε) := δ'(q, ε) ∪ {q_0}
δ'(q'_0, ε) := {q_0}
return N' = (Q', Σ, δ', 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 $\mathcal{L}(N') = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$

$\mathsf{nfa\text{-}union}(N)$:

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

Algorithm 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 $\mathcal{L}(N') = \mathcal{L}(N_1) \circ \mathcal{L}(N_2)$

$\mathsf{nfa\text{-}concatenation}(N)$:

$q'_0 := q_1$
$Q' := Q_1 \cup Q_2 \cup \{q_0\}$
$F' := F_2$
$\delta' := \delta_1 \cup \delta_2$
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 $\mathsf{can\text{-}pop\text{-}push}$ and $\mathsf{pop\text{-}push}$ as follows:

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

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

$\mathsf{pda\text{-}epsilon\text{-}closure}(P, r)$:

$result := \{r\}$
$todo := \{r\}$
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 $\mathsf{can\text{-}pop\text{-}push}(stack, u, v)$ **then**
        $stack' = \mathsf{pop\text{-}push}(stack, u, v)$
        $r' := (q, stack')$
        if $r' \notin result$ **then**
          $todo := todo \cup \{r'\}$
          $result := result \cup \{r'\}$
return $result$

We generalize the functions $\mathsf{pda\text{-}epsilon\text{-}closure}$ and $\mathsf{pda\text{-}do\text{-}transition}$ to a set of PDA states R by

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

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: $\{r' \in Q \times \Gamma^* \mid r \xrightarrow{a}\!\!\rightarrow r'\}$

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 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):

$R := \{(q_0, [])\}$
$R :=$ 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 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)
result := ∅
W := {↦}
R := {(q0, [ ])}
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 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):

$V := \{A_{pq} \mid (p, q) \in Q \times Q\}$
$R := \emptyset$
let $F = \{q_{accept}\}$
$S := A_{q_0 q_{accept}}$
$T_{push} := \{\mapsto\}$ ▷ maps stack symbols to corresponding push transitions
$T_{pop} := \{\mapsto\}$ ▷ 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**
        $R := R \cup \{A_{pq} \to a A_{rs} b\}$
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 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

let q_s, q'_a, q_a be fresh for Q
$Q' = Q \cup \{q_s, q'_a, q_a\}$
$\triangleright$ q_s new start state, q_a new single final state, q'_a additional state
let \$ be fresh for Γ
$\Gamma' = \Gamma \cup \{\$\}$
$\triangleright$ introduce stack-bottom \$ file
$\triangleright$ by convention, if not specified $\delta'(q, \alpha, X) = \emptyset$
for $q \in Q$, $a \in \Sigma$, $X \in \Gamma$ **do**
    $\delta'(q, a, X) = \delta(q, a, X)$
$\delta'(q_s, \varepsilon, \varepsilon) = \{(q_0, \$)\}$
$\delta'(q, \varepsilon, \varepsilon) = \delta(q, \varepsilon, \varepsilon) \cup \{(q'_a, \varepsilon)\}$ for $q \in F$
$\delta'(q'_a, \alpha, X) = \emptyset$ for $\alpha \in \Sigma$, $X \in \Gamma'_\varepsilon$
$\delta'(q'_a, \varepsilon, X) = \{(q'_a, \varepsilon)\}$ for $X \in \Gamma$
$\delta'(q'_a, \varepsilon, \$) = \{(q_a, \varepsilon)\}$
$F' = \{q_a\}$
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

let ¢ be fresh for Γ
$\Gamma' = \Gamma \cup \{¢\}$
▷ remove push/pop transitions: replace $q \xrightarrow{\alpha, X/Y} q'$ by $q \xrightarrow{\alpha, X/\varepsilon} \hat{q} \xrightarrow{\varepsilon, \varepsilon/Y} q'$
for $q \in Q$, $\alpha \in \Sigma_\varepsilon$, $X \in \Gamma$ **do**
    for $q' \in Q$, $Y \in \Gamma$ such that $(q', Y) \in \delta(q, \alpha, X)$ **do**
        let $\hat{q}$ be fresh for Q
        $Q = Q \cup \{\hat{q}\}$
        $\delta(q, \alpha, X) = (\delta(q, \alpha, X) \setminus \{(q', Y)\}) \cup \{(\hat{q}, \varepsilon)\}$
        $\delta(\hat{q}, \varepsilon, \varepsilon) = \{(q', Y)\}$
        ▷ implicitly $\delta(\hat{q}, a, Z) = \emptyset$ for $a \in \Sigma$, $Z \in \Gamma_\varepsilon$
▷ remove ε/ε-transitions: replace $q \xrightarrow{\alpha, \varepsilon, \varepsilon} q'$ by $q \xrightarrow{\alpha, \varepsilon, ¢} \hat{q} \xrightarrow{\varepsilon, ¢/\varepsilon} q'$
for $q \in Q$, $\alpha \in \Sigma_\varepsilon$ **do**
    for $q' \in Q$ such that $(q', \varepsilon) \in \delta(q, \alpha, \varepsilon)$ **do**
        let $\hat{q}$ be fresh for Q
        $Q = Q \cup \{\hat{q}\}$
        $\delta(q, \alpha, \varepsilon) = (\delta(q, \alpha, \varepsilon) \setminus \{(q', \varepsilon)\}) \cup \{\hat{q}, ¢)\}$
        $\delta(\hat{q}, \varepsilon, ¢) = \{(q', \varepsilon)\}$
        ▷ implicitly $\delta(\hat{q}, a, Z) = \emptyset$ for $a \in \Sigma$, $Z \in \Gamma_\varepsilon$
$\Gamma = \Gamma'$
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

$\mathsf{tm\text{-}do\text{-}transition}(T, p, tape, head)$:

$a := tape[head]$
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**
    $tape := tape \mathbin{++} [\sqcup]$ ▷ add a blank, so that tape[head] is defined
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):
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 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):
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 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)

```
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 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):

if $w = \varepsilon$ **then**
    return $S \to \varepsilon \in R$
$n := |w|$
$X := \{\mapsto\}$ ▷ initially $X(i, j) = \emptyset$ for all i, j
▷ $X(i, j)$ will contain $\{A \in V \mid A \overset{*}{\Longrightarrow} w_i w_{i+1} \ldots w_j\}$
for $0 \leq i < n$ **do**
    $X(i, i) := \{A \in V \mid A \to w_i \in R\}$
for $1 \leq m < n$ **do**
    for $0 \leq i < n - m$ **do**
        $j := i + m$
        for $i \leq k \leq j$ **do**
            for $(B, C) \in X(i, k) \times X(k + 1, j)$ **do**
                $X(i, j) := X(i, j) \cup \{A \in V \mid A \to BC \in R\}$
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 $R_n = \{A \to x \mid |x| = n\}$, 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 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)

```
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 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)

```
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 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)

$R' := R$
for $A \in V$ **do**
  $W :=$ cfg-derivable-variables(G, A)
  for $B \to \alpha \in R$ with $B \in W$ and $\alpha \notin V$ **do**
    $R' := R' \cup \{A \to \alpha\}$ ▷ Make sure not to add duplicate rules
$R' := \{A \to \alpha \in R' \mid \alpha \notin V\}$
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)

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

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

8 Regular expression algorithms

The functions $\mathsf{regexp\text{-}size}$ and $\mathsf{regexp\text{-}symbols}$ are inductively defined as

$$
\begin{array}{lcl}
\mathsf{regexp\text{-}size}(\mathbf{0}) & = & 0 \\
\mathsf{regexp\text{-}size}(\mathbf{1}) & = & 0 \\
\mathsf{regexp\text{-}size}(a) & = & 0 \\
\mathsf{regexp\text{-}size}(r^*) & = & \mathsf{regexp\text{-}size}(r) + 1 \\
\mathsf{regexp\text{-}size}(r_1 + r_2) & = & \mathsf{regexp\text{-}size}(r_1) + \mathsf{regexp\text{-}size}(r_2) + 2 \\
\mathsf{regexp\text{-}size}(r_1 \cdot r_2) & = & \mathsf{regexp\text{-}size}(r_1) + \mathsf{regexp\text{-}size}(r_2) + 2
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{regexp\text{-}symbols}(\mathbf{0}) & = & \emptyset \\
\mathsf{regexp\text{-}symbols}(\mathbf{1}) & = & \emptyset \\
\mathsf{regexp\text{-}symbols}(a) & = & \{a\} \\
\mathsf{regexp\text{-}symbols}(r^*) & = & \mathsf{regexp\text{-}symbols}(r) \\
\mathsf{regexp\text{-}symbols}(r_1 + r_2) & = & \mathsf{regexp\text{-}symbols}(r_1) \cup \mathsf{regexp\text{-}symbols}(r_2) \\
\mathsf{regexp\text{-}symbols}(r_1 \cdot r_2) & = & \mathsf{regexp\text{-}symbols}(r_1) \cup \mathsf{regexp\text{-}symbols}(r_2)
\end{array}
$$

The functions $\mathsf{regexp\text{-}simplify}$ is defined by means of the following rewrite rules

$$
\begin{array}{lcl}
\mathsf{regexp\text{-}simplify}(\mathbf{0}^*) & = & \mathbf{1} \\
\mathsf{regexp\text{-}simplify}(\mathbf{1}^*) & = & \mathbf{1} \\
\mathsf{regexp\text{-}simplify}(r^{**}) & = & r^* \\
\mathsf{regexp\text{-}simplify}(\mathbf{0} + r) & = & r \\
\mathsf{regexp\text{-}simplify}(r + \mathbf{0}) & = & r \\
\mathsf{regexp\text{-}simplify}(\mathbf{0} \cdot r) & = & \mathbf{0} \\
\mathsf{regexp\text{-}simplify}(\mathbf{1} \cdot r) & = & r \\
\mathsf{regexp\text{-}simplify}(r \cdot \mathbf{0}) & = & \mathbf{0} \\
\mathsf{regexp\text{-}simplify}(r \cdot \mathbf{1}) & = & r
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{regexp\text{-}accepts\text{-}word}(\mathbf{0}) & = & \emptyset \\
\mathsf{regexp\text{-}accepts\text{-}word}(\mathbf{1}) & = & \emptyset \\
\mathsf{regexp\text{-}accepts\text{-}word}(a) & = & \{a\} \\
\mathsf{regexp\text{-}accepts\text{-}word}(r^*) & = & \mathsf{regexp\text{-}accepts\text{-}word}(r) \\
\mathsf{regexp\text{-}accepts\text{-}word}(r_1 + r_2) & = & \mathsf{regexp\text{-}accepts\text{-}word}(r_1) \cup \mathsf{regexp\text{-}accepts\text{-}word}(r_2) \\
\mathsf{regexp\text{-}accepts\text{-}word}(r_1 \cdot r_2) & = & \mathsf{regexp\text{-}accepts\text{-}word}(r_1) \cup \mathsf{regexp\text{-}accepts\text{-}word}(r_2)
\end{array}
$$

The algorithms $\mathsf{regexp\text{-}accepts\text{-}word}$ and $\mathsf{regexp\text{-}words\text{-}up\text{-}to\text{-}n}$ 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: $w \in \mathcal{L}(r)$

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 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)

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 ⟨*symbol*⟩ is an arbitrary alpha-numeric or UTF-8 character, with the exception of those that are used as tokens in the grammars below.

⟨*blank*⟩ ::= '`blank`' ⟨*symbol*⟩

⟨*epsilon*⟩ ::= '`epsilon`' ⟨*symbol*⟩

⟨*direction*⟩ ::= '`L`' | '`R`'

⟨*input-symbols*⟩ ::= '`input_symbols`' ⟨*symbol*⟩*

⟨*stack-symbols*⟩ ::= '`stack_symbols`' ⟨*symbol*⟩*

⟨*tape-symbols*⟩ ::= '`tape_symbols`' ⟨*symbol*⟩*

⟨*identifier*⟩ ::= ⟨*symbol*⟩*

An ⟨*identifier*⟩ must be a single token, i.e. it may not contain spaces.

States

⟨*states*⟩ ::= '`states`' ⟨*state*⟩*

⟨*initial*⟩ ::= '`initial`' ⟨*state*⟩

⟨*accept*⟩ ::= '`accept`' ⟨*state*⟩

⟨*reject*⟩ ::= '`reject`' ⟨*state*⟩

⟨*final*⟩ ::= '`final`' ⟨*state*⟩*

⟨*state*⟩ ::= ⟨*identifier*⟩

DFA syntax

⟨*DFA*⟩ ::= ⟨*dfa-line*⟩*

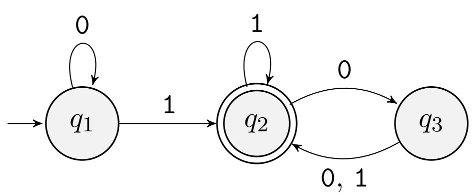
⟨*dfa-line*⟩ ::= ⟨*input-symbols*⟩
| ⟨*states*⟩
| ⟨*initial*⟩
| ⟨*final*⟩
| ⟨*dfa-transition*⟩

$\langle$*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*$\rangle$
| $\langle$*epsilon*$\rangle$
| $\langle$*states*$\rangle$
| $\langle$*initial*$\rangle$
| $\langle$*final*$\rangle$
| $\langle$*nfa-transition*$\rangle$

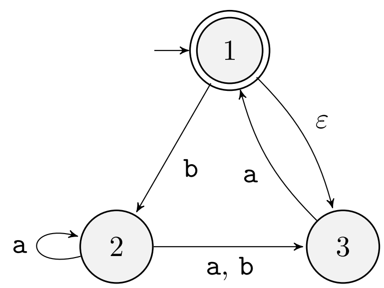
$\langle$*nfa-transition*$\rangle$ ::= $\langle$*state*$\rangle$ $\langle$*state*$\rangle$ $\langle$*nfa-update*$\rangle$*

$\langle$*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

⟨*PDA*⟩ ::= ⟨*pda-line*⟩*

⟨*pda-line*⟩ ::= ⟨*input-symbols*⟩
| ⟨*stack-symbols*⟩
| ⟨*epsilon*⟩
| ⟨*states*⟩
| ⟨*initial*⟩
| ⟨*final*⟩
| ⟨*pda-transition*⟩

⟨*pda-transition*⟩ ::= ⟨*state*⟩ ⟨*state*⟩ ⟨*pda-update*⟩*

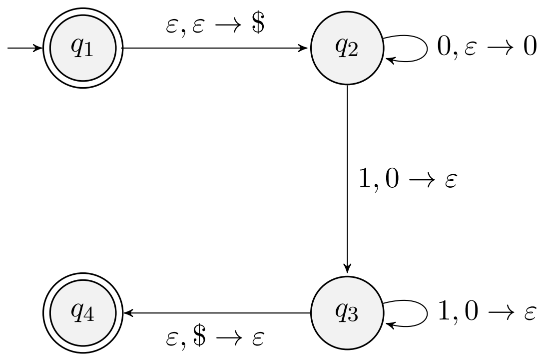
⟨*pda-update*⟩ ::= ⟨*symbol*⟩ ',' ⟨*symbol*⟩ ⟨*symbol*⟩

The elements ⟨*input-symbols*⟩, ⟨*stack-symbols*⟩ and ⟨*states*⟩ are optional. If they are omitted, they are derived from the transitions. If ⟨*epsilon*⟩ is omitted, the symbol '-' is treated as the empty string. A ⟨*pda-update*⟩ 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*$\rangle$

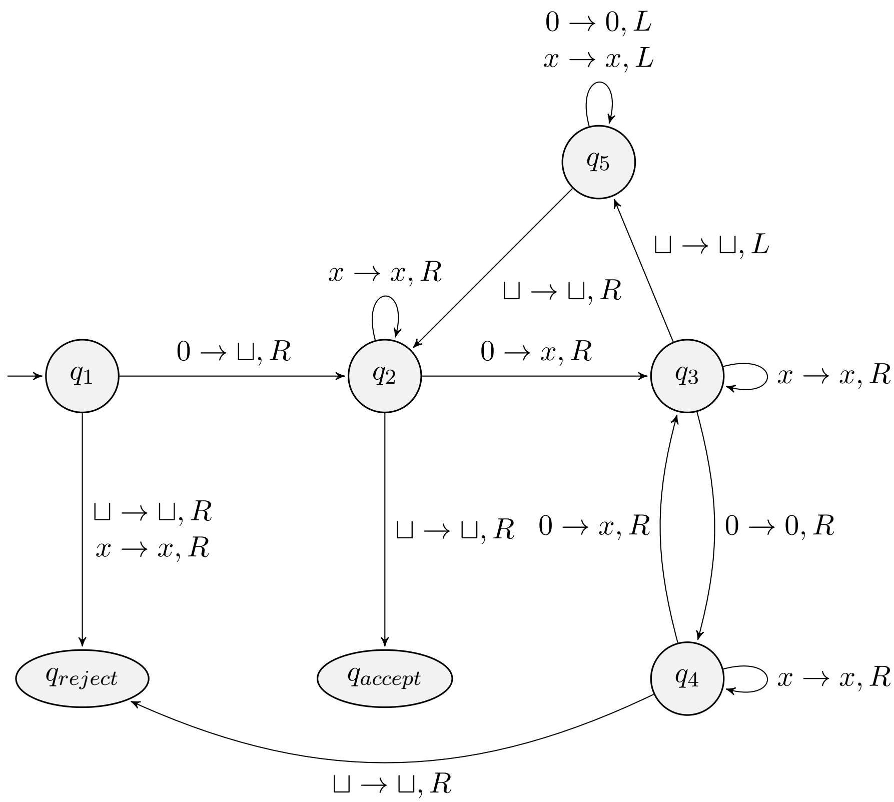
$\langle$*tm-transition*$\rangle$ ::= $\langle$*state*$\rangle$ $\langle$*state*$\rangle$ $\langle$*tm-update*$\rangle$*

$\langle$*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.

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.