CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression
Student:	Bc. Tomáš Pecka
Supervisor:	Ing. Jan Trávníček
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Study regular tree expressions. Study the construction of the Glushkov's automaton for a given regular expression.
Propose a method of the construction of a pushdown automaton accepting a language given by a regular tree expression and describe it formally.
Implement the proposed method of the conversion of a regular tree expression to the pushdown automaton.
Test your implementation appropriately using regular tree expressions and trees of your choice.

References

Will be provided by the supervisor.

Jan Janoušek

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

České vysoké učení technické v Praze
Fakulta informačních technologií
L.S.
-5-

[illegible]

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 7, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression

Bc. Tomáš Pecka

Supervisor: Ing. Jan Trávniček

9th May 2016

Acknowledgements

I would like to thank my supervisor, Jan Trávníček, for the useful comments, remarks and help throughout my studies. Also, I would like to thank my family and friends for their support.

Declaration

In Prague on 9th May 2016 ....

Citation of this thesis

Pecka, Tomáš. *Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce studuje regulární stromové výrazy, formalismus pro popis regulárních stromových jazyků. Hlavním přínosem práce je nový algoritmus pro převod regulárního stromového výrazu na ekvivalentní zásobníkový automat, který přijímá stromy v lineárním postfixovém zápisu. Výsledný automat patří do kategorie real-time height-deterministic zásobníkových automatů, což znamená, že je vždy zdeterminizovatelný. Algoritmus vytvářející tento automat je modifikací Glushkovova algoritmu pro převod regulárních výrazů na nedeterministický konečný automat. Implementace převodu je v jazyce C++ jako rozšíření Automatové knihovny.

Klíčová slova Glushkovův automat, regulární stromový výraz, zásobníkový automat, real-time height-deterministic zásobníkový automat

Abstract

This thesis studies regular tree expressions, a formalism for describing regular tree languages. Main topic of this thesis is a new algorithm for converting a regular tree expression to a pushdown automaton recognizing trees in their linear postfix notation. Resulting pushdown automaton is real-time height-deterministic, i.e., it can always be determinised. The algorithm for conversion is an adaptation of Glushkov's algorithm for converting a regular expression to a nondeterministic finite automaton. C++ implementation of the algorithm is attached as an extension of Automata Library.

Keywords Glushkov's automaton, regular tree expression, pushdown automaton, real-time height-deterministic pushdown automaton

Contents

List of Figures

List of Algorithms

Introduction

This work contributes to a new algorithmic discipline called *Arbology*. Arbology research group was founded in 2008 at Czech Technical University in Prague[arb]. The name arbology comes from a Spanish word *arbol*, which means a tree. Arbology discipline focuses on algorithms on trees and uses a pushdown automaton as a model of computation. An idea of processing a tree by a pushdown automaton comes from a crucial observation that every tree can be represented as a string (in what is called a linear notation of a tree). This method of processing is similiar to what sequential algorithms do while traversing a tree in linear order.

The inspiration for building arbology algorithms comes from stringology. In stringology, finite automata are usually used as a model of computation. In arbology, trees are processed by pushdown automata.

Many algorithms analogous to the stringology ones were created. For example indexing trees, computing repeats in trees, searching in trees and last but not least converting regular tree expression to pushdown automaton by extension of Thompson's algorithm [JM09, PJM11].

Motivation

Regular tree expressions are a natural formalism for describing a set of trees. They are analogous to regular (string) expressions in the tree domain. Regular expression matching is an useful operation in computer science with number of applications. In string domain, one can describe sets of strings or search within the text with the use of regular expressions. In tree languages, it is similar. However, with regular tree expressions one can describe tree-like structures or search in them. Nice practical examples of tree-like structures are for instance XML documents or LISP programs. Unlike for strings, the most famous models of computation for trees are various kinds of tree automata.

There are some related works in tree domain for regular tree matching problem with tree automaton as a model of computation. This thesis presents a new systematic approach to the construction of a regular tree expression matcher by adaptation of Glushkov's algorithm from strings to trees. Created determinisable pushdown automaton reads subject trees described by a regular tree expression in their linear postfix notation. This pushdown automaton has the property of being real-time height-deterministic. The height-determinism means that for each run on the same input the height of automata pushdown store is the same in all nondeterministic paths and is predetermined by the input word. Real-time height-deterministic automata can be always determinised[NS07] and they accept a subclass of deterministic context-free languages. Deterministic pushdown automata can recognize the input string in linear time with respect to the size of input.

This is not the first work on this topic in Arbology research group. Different method for converting regular tree expressions to a determinisable pushdown automata was presented in [PJM11].

Organization of the text

This text is structured into multiple chapters.

Introduction provides an insight into the topic. It presents the Arbology research and the motivation for this thesis.

Chapter 1 defines basic notions in string and tree languages.

Chapter 2 introduces reader to regular tree expressions.

Chapter 3 presents related works in both string and tree domains as well as related algorithms for the conversion of regular tree expression to pushdown automaton.

Chapter 4 is the most important part of this work. It shows the process of creating the algorithm for conversion and presents the algorithm itself as well as some examples and formal analysis of the method.

Chapter 5 presents the Automata Library (ALIB), a framework for implementing algorithms in formal languages, describes the caveats in implementation and the process of testing.

Conclusion provides the summary and results of this thesis. It also points at some interesting problems for future work.

Chapter 1

Basic notions

This chapter presents basic notions needed for full definition of regular tree expressions and their conversion into pushdown automata. Basic definitions for strings and automata are provided similar to [Mel03, Hol15, ASU86]. Definitions of trees are similar to [CDG+07]. Regular tree expressions are defined exhaustively in the following chapter.

1.1 Alphabet, Language

Definition 1.1 (Alphabet)**.** *Alphabet* is a finite non-empty set of symbols usually denoted by Σ. △

Definition 1.2 (Word, string)**.** *Word* or *string* over *alphabet* Σ is a finite sequence of symbols of alphabet. An empty word is denoted by ε. Given a non-empty word w we can write $w = w_1 w_2 \dots w_n$ where $w_i \in \Sigma$. Length of such word is $|w| = n$. Length of the empty word is $|\varepsilon| = 0$. △

Definition 1.3 (Concatenation of words)**.** Let u and v be words over alphabet Σ. *Concatenation* operation (denoted by $u \cdot v$ or uv) is appending the word v to the word u in order to create word uv. △

Definition 1.4 (Language)**.** Set of all words over *alphabet* Σ is denoted by Σ^*. This set always contains the empty word ε. Set $L \subseteq \Sigma^*$ is called *language* L. △

Definition 1.5 (Language operations)**.** Let L_1 be a language over alphabet Σ_1 and let L_2 be a language over alphabet Σ_2. *Union* of L_1 and L_2 is a language L consisting of words in both languages, i.e., $L = L_1 \cup L_2$ over alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$. *Concatenation* of two languages L_1 and L_2 is language $L = L_1 \cdot L_2 = \{uv : u \in L_1,\ v \in L_2\}$ over alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$. △

Definition 1.6 (Closure of a language)**.** Let L be a language over alphabet Σ. *N-th power* of language L is defined as $L^n = L \cdot L^{n-1}$ with base case $L_0 = \{\varepsilon\}$. *Iteration (Kleene star)* of L is $L^* = \bigcup_{n=0}^{\infty} L^n$. △

Example 1.1. Let $\Sigma = \{a, b, c, d, \ldots, z\}$. The set of all words Σ^* over this alphabet is infinite and includes all possible combinations of elements of Σ of arbitrary length with repetitions.

Let $L \subseteq \Sigma^* = \{$ int, char, double, float $\}$. Then L is a language consisting of four words (primitive types of C language). ■

1.2 Grammars

Definition 1.7 (Grammar)**.** Grammar is a quadruple $G = (N, \Sigma, P, S)$, where N is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols ($\Sigma \cap N = \emptyset$), P is a set of production rules. It is a finite subset of $(N \cup \Sigma)^* \cdot N \cdot (N \cup \Sigma)^* \times (N \cup \Sigma)^*$. An element $(\alpha, \beta) \in P$ is written as $\alpha \to \beta$ and is called a rule. $S \in N$ is the start symbol of the grammar. △

Definition 1.8 (Derivation)**.** Let $G = (N, \Sigma, P, S)$, $x, y \in (N \cup \Sigma)^*$. We say that x derives y in one step ($x \Rightarrow y$), if there exists $\alpha \to \beta \in P$ and $\gamma, \delta \in (N \cup \Sigma)^*$ such that $x = \gamma\alpha\delta, y = \gamma\beta\delta$ (i.e., $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$). △

Definition 1.9 (Derivation)**.** $\alpha \Rightarrow^k \beta$ if there exists a sequence $\alpha_0, \alpha_1, \ldots, \alpha_k$ for $k \geq 0$, of $k+1$ strings such that $\alpha = \alpha_0 \alpha_{i-1} \Rightarrow \alpha i$ for $1 \leq i \leq k$, and $\alpha_k = \beta$. This sequence is called derivation of string β from string α that has length k in grammar G. △

Definition 1.10 (Derivation)**.** Transitive and reflexive closure of relation $\Rightarrow$: $\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow^i \beta$ for some $i \geq 0$. △

Definition 1.11 (Language of grammar)**.** $L(G) = \{w : w \in \Sigma^*, \exists S \Rightarrow^* w\}$ is the language generated by grammar $G = (N, \Sigma, P, S)$. △

Definition 1.12 (Classes of grammars)**.** Let $G = (N, \Sigma, P, S)$. We say that G is:

0. *Unrestricted* (type 0), if it satisfies the general grammar definition.
1. *Context-sensitive* (type 1), if every rule from P is of the form $\gamma A \delta \to \gamma\alpha\delta$, where $\gamma, \delta \in (N \cup \Sigma)^*$, $\alpha \in (N \cup \Sigma)^+$, $A \in N$, or the form $S \to \varepsilon$ in case that S is not present in the right-hand side of any rule.
2. *Context-free* (type 2), if every rule is of the form $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup \Sigma)^*$.
3. *Regular* (type 3), if every rule is of the form $A \to aB$ or $A \to a$, where $A, B \in N$, $a \in \Sigma$, or the form $S \to \varepsilon$ in case that S is not present in the right-hand side of any rule. △

1.3 Hierarchy of Languages

Definition 1.13 (Language type)**.** Language is

0. *recursively enumerable* (type 0), if $\exists$ unrestricted grammar which generates it,
1. *context-sensitive* (type 1), if $\exists$ context-sensitive grammar which generates it,
2. *context-free* (type 2), if $\exists$ context-free grammar which generates it,
3. *regular* (type 3), if $\exists$ regular grammar which generates it. $\triangle$

1.4 Regular Languages

Language is regular if there exists a regular grammar generating it. This type of language can be recognized by finite automata (both deterministic and nondeterministic) and can be described using regular expressions.

1.4.1 Finite Automaton

A finite automaton is an abstract computational model. Its purpose is to determine whether an input word belongs to a regular language described by the finite automaton.

1.4.1.1 Deterministic Finite Automaton

Definition 1.14 (Deterministic finite automaton (DFA))**.** *DFA* is a 5-tuple $A = (Q, T, \delta, q_0, F)$, where

- Q is a finite non-empty set of states,
- Σ is a finite set of input symbols (alphabet),
- δ is a state transition function, $\delta : Q \times \Sigma \to Q$,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states. $\triangle$

Definition 1.15 (Configuration of a finite automaton)**.** *Configuration of a DFA* is a pair $(q, w) \in Q \times \Sigma^*$. Configuration (q_0, w) is the *initial configuration* and finally (q, ε) with $q \in F$ is the *final configuration* of the DFA. $\triangle$

Definition 1.16 (Move in DFA)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. Let $\vdash_A$ be a relation over $Q \times \Sigma^*$ (i.e., subset of $(Q \times \Sigma^*) \times (Q \times \Sigma^*)$) such that $(q, w) \vdash_A (p, w')$ iff $w = aw'$ and $\delta(q, a) = p$ for some $a \in \Sigma$, $w \in \Sigma^*$. An element of relation $\vdash_A$ is called a move in automaton A.

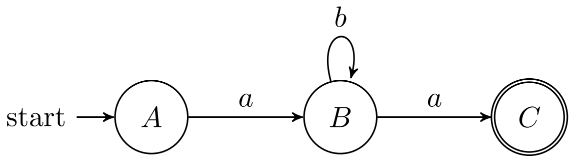


Figure 1.1: Transition diagram of a DFA.

- $\vdash_A^k$ is the k-th power of relation $\vdash_A$
- $(\alpha_0, \beta_0) \vdash_A^K (\alpha_k, \beta_k)$ if
  $\exists(\alpha_i, \beta_i),\ 0 < i < k : (\alpha_i, \beta_i) \vdash_A (\alpha_{i+1}, \beta_{i+1}), 0 \leq i < k$
- $\vdash_A^+$ is the transitive closure of relation $\vdash_A$
- $\vdash_A^*$ is the transitive and reflexive closure of relation $\vdash_A$
- $(q, aw') \vdash_M (p, w')$ means $((q, aw'), (p, w')) \in \vdash_M$. △

Definition 1.17 (Language of DFA)**.** String $w \in \Sigma^*$ is accepted by a DFA $A = (Q, \Sigma, \delta, q_0, F)$ if $\exists(q_0, w) \vdash_A^* (q, \varepsilon)$ for some $q \in F$. Language recognized by automaton A is denoted by $L(A)$ and $L(A) = \{w : w \in \Sigma^*, \exists q \in F : (q_0, w) \vdash_A^* (q, \varepsilon)\}$.

String $w \in L(A)$ if there exists a sequence of moves from the initial configuration (q_0, w) into an accepting configuration (q, ε). △

Finite automata can be depicted using transition diagrams. Transition diagram is an oriented graph where states of the automaton corresponds to nodes and transitions between states corresponds to the edges of the graph. Value of the edge is the input symbol of the transition. Initial state of the automaton is the target of the arrow with *start* label. Final states are drawn as double circle around the node. Example of this is in figure 1.1.

Example 1.2 (DFA)**.** Consider a language L over an alphabet $\Sigma = \{a, b\}$ and $L = \{aa, aba, abba, abbba, abbbba, \ldots\}$. Then DFA in figure 1.1 accepts words from this language. ■

1.4.1.2 Nondeterministic Finite Automaton

Definition 1.18 (Nondeterministic Finite Automaton)**.** *Nondeterministic finite automaton (NFA)* is a 5-tuple $A = (Q, T, \delta, q_0, F)$, where

- Q is a finite non-empty set of states,
- Σ is a finite set of input symbols (alphabet),
- δ is a state transition function, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (powerset of Q),

- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states. $\triangle$

Definition 1.19 (Move in NFA)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton. Let $\vdash_A$ be a relation over $Q \times \Sigma^*$ (i.e., subset of $(Q \times \Sigma^*) \times (Q \times \Sigma^*)$) such that $(q, w) \vdash_A (p, w')$ iff $w = aw'$ and $p \in \delta(q, a)$ for some $a \in \Sigma$, $w \in \Sigma^*$. An element of relation $\dashv_A$ is called a move in automaton A.

- $\dashv_A^k$ is the k-th power of relation $\dashv_A$.
- $\dashv_A^+$ is the transitive closure of relation $\dashv_A$.
- $\dashv_A^*$ is the transitive and reflexive closure of relation $\dashv_A$. $\triangle$

Definition 1.20 (Language of NFA)**.** Same as definition 1.17, but NFA is used. $\triangle$

Definition 1.21 (NFA with ε-transitions)**.** NFA with ε-transitions is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, F are the same as in the definition of NFA. Mapping δ is defined as follows: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$. $\triangle$

Definition 1.22 (Move in NFA with ε-transitions)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA with ε-transitions. An element of relation $\vdash_A \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ is called a move in automaton A. If $p \in \delta(q, a)$, $a \in \Sigma \cup \{\varepsilon\}$, then $(q, aw) \dashv_A (p, w)$ for every $w \in \Sigma^*$. $\triangle$

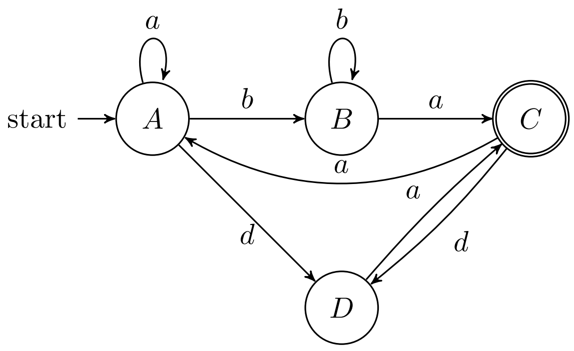
1.4.1.3 Homogeneous Finite Automaton

Definition 1.23 (Set of target states)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$. For any $a \in \Sigma$ set $Q(a) \subseteq Q$ of target states is defined: $Q(a) = \{q : q \in \delta(p, a), p, q \in Q\}$. $\triangle$

Definition 1.24 (Homogeneous finite automaton)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$ and $Q(a)$ be the sets of target states $\forall a \in \Sigma$. If for all pairs of symbols $a, b \in \Sigma$, $a \neq b$ holds that $Q(a) \cap Q(b) = \emptyset$, then the automaton A is called homogeneous. $\triangle$

Example 1.3. Automaton A from figure 1.2 is homogeneous because it has the following disjoint sets of target states:

- $Q(a) = \{A, C\}$,
- $Q(b) = \{B\}$,
- $Q(d) = \{D\}$. ■



$$\begin{tikzcd}
\text{start} \arrow[r] & A \arrow[loop above, "a"] \arrow[r, "b"] \arrow[rd, "d"'] & B \arrow[loop above, "b"] \arrow[r, "a"] & C \arrow[ll, bend left, "a"] \arrow[ld, shift left, "d"] \\
 & & D \arrow[ru, "a"] &
\end{tikzcd}$$

Figure 1.2: Homogeneous NFA.

1.4.2 Regular Expression

A regular expression is a formal model for description of regular languages. Every regular language can be described using regular expression and vice versa[Kle56].

Definition 1.25 (Regular expression)**.** *Regular expression* E over alphabet Σ is defined inductively:

1. $\emptyset$, ε and a $\forall a \in \Sigma$ are regular expressions.
2. If E_1 and E_2 are regular expressions over Σ, then
    a) union $(E_1 + E_2)$,
    b) concatenation $(E_1 \cdot E_2$ or $E_1E_2)$,
    c) iteration (E_1^*)

   are also regular expressions over Σ. △

Definition 1.26 (Language of a regular expression)**.** Language described by regular expression E (denoted by $L(E)$) is defined inductively:

1. a) $L(\emptyset) = \emptyset$,
   b) $L(\varepsilon) = \{\varepsilon\}$,
   c) $L(a) = \{a\}$ if $a \in \Sigma$.
2. a) $L(E_1 + E_2) = L(E_1) \cup L(E_2)$,
   b) $L(E_1 \cdot E_2) = L(E_1) \cdot L(L_2)$,
   c) $L(E_1^*) = L(E_1)^*$. △

Example 1.4. Automaton in example 1.2 accepts a language which can be expressed by regular expression $E = ab^*a$. ■

1.5 Context-free Languages

A language is regular if there exists a regular grammar generating it. This type of language can be recognized by finite automata (both deterministic and nondeterministic) and can be described using regular expressions.

1.5.1 Pushdown Automaton

Definition 1.27. Pushdown automaton (PDA) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ where

- Q is a finite set of states,
- Σ is a finite input alphabet,
- Γ is a finite set of pushdown store alphabet,
- δ is a transition function $(\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \mapsto Q \times \Gamma^*)$,
- $q_0 \in Q$ is the initial state,
- $\perp \in \Gamma$ is the initial pushdown store symbol,
- $F \subseteq Q$ is a set of final states. △

Definition 1.28 (Configuration of PDA)**.** Configuration of PDA is $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, where q is the current state, w is yet unprocessed part of the input string and α is the pushdown store content. The initial configuration is $(q_0, w, \perp)$, $w \in \Sigma^*$. △

Definition 1.29 (Move in PDA)**.** Let $R = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ be a PDA. Let $\vdash_R$ be a relation over $Q \times \Sigma^* \times \Gamma^*$ (i.e., subset of $(Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Sigma^* \times \Gamma^*)$) such that $(q, w, \beta\alpha) \vdash_R (p, w', \beta\gamma)$ iff $w = aw'$ and $(p, \gamma) \in \delta(q, a, \alpha)$ for some $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $\alpha, \beta\gamma \in \Gamma^*$. An element of relation $\dashv_R$ is called a move in automaton R.

- $\dashv_R^k$ is the k-th power of relation $\dashv_R$.
- $\dashv_R^+$ is the transitive closure of relation $\dashv_R$.
- $\dashv_R^*$ is the transitive and reflexive closure of relation $\dashv_R$. △

Note. As follows from previous definition, in this thesis, the top of the pushdown store is situated on the right.

Definition 1.30 (Language of PDA)**.** Language accepted by PDA $R = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$:

1. by transition into a final state: $L(R) = \{w : w \in \Sigma^*, \exists\gamma \in \Gamma^*, \exists q \in F, (q_0, w, \perp) \vdash^* (q, \varepsilon, \gamma)\}$,

2. by empty pushdown store: $L(R) = \{w : w \in \Sigma^*, \exists q \in Q, (q_0, w, \bot) \vdash^* (q, \varepsilon, \varepsilon)\}$. △

Definition 1.31 (Deterministic PDA)**.** PDA $(Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ is deterministic if:

1. $|\delta(q, a, \Gamma)| \leq 1, \forall q \in Q, \forall a \in (\Sigma \cup \{\varepsilon\}), \forall \gamma \in \Gamma^*$,

2. If $\delta(q, a, \alpha) \neq \emptyset, \gamma(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, then α is not a prefix of β and β is not a prefix of α (i.e., $\alpha\gamma \neq \beta, \alpha \neq \beta\gamma, \gamma \in \Gamma^*$),

3. If $\delta(q, a, \alpha) \neq \emptyset, \gamma(q, \varepsilon, \beta) \neq \emptyset$, then α is not a prefix of β and β is not a prefix of α (i.e., $\alpha\gamma \neq \beta, \alpha \neq \beta\gamma, \gamma \in \Gamma^*$). △

Pushdown automaton's transition diagram can be drawn too with same principle as for finite automata except for edge labels. Label is now in a form $a \mid \alpha \to \beta$ where a is an input symbol, α is a sequence of pushdown store symbols which will be removed from the top of the pushdown store and replaced with a sequence β.

1.5.1.1 Real-Time Height-deterministic Pushdown Automaton

Determinisable pushdown automata are classified into following tree groups:

- input-driven pushdown automata,

- visibly pushdown automata[AM04],

- height-deterministic pushdown automata[NS07].

Height-deterministic pushdown automata are a generalization of visibly pushdown automata. They have better properties than visibly pushdown automata and describe larger set of languages.

Definition 1.32 (Height-deterministic PDA)**.** Height-deterministic PDA is such PDA, which on all of its runs on a input $w \in (\Sigma \cup \{\varepsilon\})$ leads to same pushdown store height. △

Definition 1.33 (Real-time height-deterministic PDA)**.** Real-time height-deterministic PDA is always determinisable. It is such PDA that is height-deterministic and for every state holds that its outgoing transitions are either ε-transitions or non-ε-transitions. △

1.6 Trees

This section defines basic notions for trees similarly to [JM09, CDG$^+$07] based on the concepts from graph theory[AU72].

Definition 1.34 (Ranked alphabet)**.** A ranked alphabet $\mathcal{F}$ is finite nonempty set of symbols. Each symbol of this set has a unique *arity* (or *rank*) assigned. Arity is an integer number greater or equal than zero. Arity of a symbol a is denoted by $arity(a)$. The set of symbols of arity n from alphabet $\mathcal{F}$ is denoted by $\mathcal{F}_n$. Symbols with zero arity are called *constants.* △

Note. It is assumed that $\mathcal{F}$ contains at least one constant symbol. The arity of a symbol represents the number of children of the symbol. If there were no constant symbols in the alphabet $\mathcal{F}$, a tree over $\mathcal{F}$ would have no leaf.

Note. If the context is clear, a shorter notation of arity will be used. Symbol $a2$ will declare symbol a with arity 2.

Definition 1.35 (Directed graph)**.** A *directed graph* G is a pair (N, R) where N is a set of notes and R is a set of lists of edges such that each element R is of the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. This element indicates that for node f, there are n edges from f to $g_1, g_2, \ldots g_n$. △

Definition 1.36 (Path, cycle)**.** A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$ is a *path* of length n from node f_0 to node f_n. Between i-th and $(i+1)$-th element of a path, there must be an edge. A *cycle* is such path $(f_0, f_1, \ldots, f_n)$, that starts and ends in the same node, i.e., $f_0 = f_n$. △

Definition 1.37 (DAG)**.** An ordered *directed acyclic graph* (DAG) is an ordered directed graph with no cycles. △

Definition 1.38 (Node degree)**.** An *in-degree* of a node f is the number of distinct pairs $(g, f) \in R$, $g \in N$. An *out-degree* of f is the number of distinct pairs $(f, g) \in R$, $g \in N$. A node with out-degree 0 is called *leaf.* △

Definition 1.39 (Directed tree)**.** A *directed tree* is rooted and directed tree t is a DAG $t = (N, R)$ with special node $r \in N$ called the *root* such that

1. r has in-degree 0,
2. all other nodes of t have in-degree 1,
3. there is just one path from the root r to every $f \in N$, $f \neq r$. △

Definition 1.40 (Labeled tree)**.** *A labelled tree* is a tree where every node is labelled by a symbol $a \in \mathcal{A}$ (alphabet). △

Definition 1.41 (Ranked tree)**.** *A ranked tree* is a labelled tree with labels from ranked alphabet. Out-degree of a node $a \in \mathcal{F}$ is $arity(a)$. △

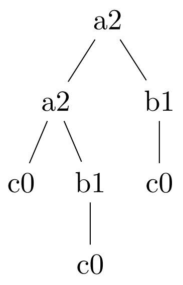


Figure 1.3: Tree t over ranked alphabet $\mathcal{F} = \{a2, b1, c0\}$.

Definition 1.42 (Ordered tree)**.** *An ordered tree* is a tree where direct descendants of a node are ordered. △

Note. Throughout this work it is understood that all trees are directed, rooted, labelled, ranked and ordered at the same time.

Example 1.5. Consider a tree t from figure 1.3. This tree is directed, rooted, labelled, ranked and ordered. The root of a tree is $a2$ with ordered tuple of children ($a2$, $b1$). Every node is labelled by a symbol from $\mathcal{F}$ and has arity. ■

1.6.1 Linear Notations of Trees

Trees can also be seen as strings. Postfix and prefix notations are examples of linear notations. In this work, only postfix notation is focused on.

Definition 1.43. Function $root(t)$ returns the root node of a tree t. △

Definition 1.44 (Postfix notation of ranked tree)**.** The postfix notation of a tree t is denoted by $post(t)$. It is defined inductively:

1. $post(t) = root(t)$ if $root(t)$ is also a leaf,
2. $post(t) = post(b_1) \cdot post(b_2) \cdot \ldots \cdot post(b_n) \cdot root(t)$, where $b_1, b_2, \ldots, b_n$ are direct descendants of $root(t)$. △

Notation $post(L)$ is also used where L is a tree language with meaning of $post(L) = \{post(t) : t \in L\}$.

Notice that given a tree t and its postfix notation $post(t)$, all subtrees of t in postfix notation are substrings of $post(t)$. However, not every substring of a tree in postfix notation is a postfix notation of a subtree.

Example 1.6. Consider tree t from figure 1.3. Postfix notation of a tree t is $post(t) = c0\, c0\, b1\, a2\, c0\, b1\, a2$. However, substring $c0\, b1\, a2$ of $post(t)$ is not a linearised subtree of t. ■

CHAPTER 2

Regular Tree Expressions

Regular tree expressions are a generalization of regular (string) expressions on trees. There is, however, one issue that needs to be solved. The concatenation operation has no relevance on trees therefore the regular tree expressions need to solve this issue by some kind of generalization of this operation which is called substitution. Regular tree expressions are defined similarly as in [CDG+07, PJM11].

2.1 Introduction

Tree regular expressions as defined in this thesis are over ranked alphabet denoted as $\mathcal{F}$. Unlike strings, which can be concatenated only at its ends (and beginnings), trees can be connected in multiple places. A notion for regular tree expressions needs to be defined to mark some specific positions where concatenation and iteration can take place in trees. Those places are represented by constant symbols (symbols of arity 0) $\square_1, \square_2, \ldots$. Set of all such symbols is usually denoted by $\mathcal{K}$ and it is is disjoint from $\mathcal{F}$, i.e., $\mathcal{F} \cap \mathcal{K} = \emptyset$. Now, trees over alphabet $\mathcal{F} \cup \mathcal{K}$ can be constructed.

Note that there is also different notation of regular tree languages. This different notation uses only one alphabet, $\mathcal{F}$, and allows the substitution operation on any constant symbol from $\mathcal{F}$ [LSZ13]. Both notations are equal in terms of expressiveness.

2.2 Operations

In this section individual operations on trees are presented. Regular tree expressions are defined formally using these operations later.

2.2.1 Substitution

Substitution is an operation that replaces some symbol $\square_i$ by another tree. Substitution operation is denoted by $t\{\square_1 \leftarrow t_1, \square_2 \leftarrow t_2, \ldots, \square_n \leftarrow t_n\}$. This means that symbol $\square_i$ of tree t will be replaced by tree t_i.

Substitution operation can be also defined on languages (set of trees). The semantics of this operation is that $\square_i$ symbol is substituted by all trees from language creating a new set of trees.

Definition 2.1 (Tree substitution)**.** The tree substitution is defined by the following identities:

- $\square_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = L_i$, for $i = 1, \ldots, n$,
- $a\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \{a\}, \forall a \in \mathcal{F} \cup \mathcal{K}$ such that arity of a is 0 and $a \neq \square_1, \ldots, a \neq \square_n$,
- $f(s_1, \ldots, s_n)\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \{f(t_1, \ldots t_n) | t_i \in s_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}\}$.

The tree substitution on languages is defined as follows:
$L_1 \cdot \square L_2 = \{t_1 \cdot \square t_2 : t_1 \in L_1,\ t_2 \in L_2\}$ △

Example 2.1. Let $\mathcal{F} = \{a_2, b_0, c_0\}$ and $\mathcal{K} = \{\square\}$. Let $t = a_2(b_0, \square)$ and $L = \{a_2(b_0, c_0),\ c_0\}$. Then $t\{\square \leftarrow L\} = \{a_2(b_0, a_2(b_0, c_0)),\ a_2(b_0, c_0)\}$. ■

Example 2.2. Let $\mathcal{F} = \{a_2, b_0, c_0\}$ and $\mathcal{K} = \{\square\}$. Let $L_1 = \{a_2(b_0, \square),\ \square\}$ and $L_2 = \{c_0\}$. Then $L_1 \cdot \square L_2 = \{a_2(b_0, c_0),\ c_0\}$. ■

2.2.2 Iteration

For the definition of a iteration operator of regular tree expressions, closure of a regular tree language and sequence of successive iterations need to be defined.

Definition 2.2 (Iteration of a language)**.** Given a language L, the tree language of $T(\mathcal{F} \cup \mathcal{K})$ and element $\square \in \mathcal{K}$, the sequence $L^{n,\square}$ is defined by the equalities:

- $L^{0,\square} = \{\square\}$,
- $L^{n+1,\square} = L \cdot \square L^{n,\square}$. △

Now, closure of a regular tree language can be defined.

Definition 2.3 (Closure of a tree language)**.** The closure $L^{*,\square}$ is the union of $L^{n,\square}$ for all $n \geq 0$, i.e., $L^{*,\square} = \bigcup_{n \geq 0} L^{n,\square}$. △

From the definitions 2.2 and 2.3 it is more than obvious that $\{\square\} \subseteq L^{*,\square}$ holds.

Example 2.3. Let $\mathcal{F} = \{a_2, b_0, c_0\}$ and $\mathcal{K} = \{\square\}$. Let $L = a_2(b_0, \square)$, then

$$\begin{aligned} L^{*,\square} =& \{\square\} \cup \\ & \{a_2(b_0, \square)\} \cup \\ & \{a_2(b_0, \square),\ a2(b_0, a_2(b_0, \square))\} \cup \\ & \{a_2(b_0, \square),\ a2(b_0, a_2(b_0, \square)),\ a_2(b_0, a2(b_0, a_2(b_0, \square)))\} \cup \\ & \dots \end{aligned}$$ ■

2.2.3 Alternation

The tree alternation operation of regular tree languages is not different to an alternation of string languages.

Definition 2.4 (Alternation of tree languages)**.** Given two regular tree languages L_1, L_2 of $T(\mathcal{F} \cup \mathcal{K})$, the alternation of L_1 to L_2 is denoted by $L_1 + L_2$. Result of this operator is a set of trees obtained by union of regular tree languages L_1 and L_2, i.e., $L_1 \cup L_2$. △

2.3 Regular Tree Expression

Using operations described in the previous section, regular tree expressions can be defined.

Definition 2.5 (Regular tree expression)**.** The set $RTE(\mathcal{F}, \mathcal{K})$ of regular tree expressions over alphabets $\mathcal{F}$ and $\mathcal{K}$ is the smallest set such that:

- the empty set $\emptyset$ is in $RTE(\mathcal{F}, \mathcal{K})$,
- if $a \in F_0 \cup K$ is a constant, then $a \in RTE(\mathcal{F}, \mathcal{K})$,
- if $f \in F_n$ has arity $n > 0$ and $E_1, \dots, E_n$ are regular tree expressions of $RTE(\mathcal{F}, \mathcal{K})$ then $f(E_1, \dots, E_n)$ is a regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$,
- if E_1, E_2 are regular tree expressions of $RTE(\mathcal{F}, \mathcal{K})$ then $E_1 + E_2$ is a regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$,
- if E_1, E_2 are regular tree expressions of $RTE(\mathcal{F}, \mathcal{K})$ and $\square$ is and element of $\mathcal{K}$ then $E_1 \cdot \square E_2$ is a regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$,
- if E is regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$ and $\square$ is and element of $\mathcal{K}$ then $E^{*,\square}$ is a regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$. △

Definition 2.6. Properties of operations and languages generated by regular tree expressions are identified by the following equalities:

- $L(\emptyset) = \emptyset$,
- $L(a) = \{a\}$ for $a \in F_0 \cup K$,
- $L(f(E_1, \dots, E_n)) = \{f(s_1, \dots, s_n) \mid s_1 \in L(E_1),\ s_2 \in L(E_2), \dots, s_n \in L(E_n)\}$,
- $L(E_1 + E_2) = L(E_1) \cup L(E_2)$,
- $L(E_1 \cdot \Box E_2) = L(E_1)\{\Box \leftarrow L(E_2)\}$,
- $L(E^{*,\Box}) = L(E)^{*,\Box}$. △

Example 2.4 (Language of regular tree expression)**.** Let $\mathcal{F} = \{a_2, b_0, c_0\}$ and $\mathcal{K} = \{\Box\}$. Then

$$a_2(b_0, \Box)^{*,\Box} \cdot \Box c_0$$

is a regular tree expression of $RTE(\mathcal{F}, \mathcal{K})$ which denotes the following set of trees:

$$\{c_0, a_2(b_0, c_0), a_2(b_0, a_2(b_0, c_0)), a_2(b_0, a_2(b_0, a_2(b_0, c_0))), \dots\}.$$

Regular tree expressions and the trees can also be written in their syntax tree form as can be seen in figures 2.1 and 2.2. ■

Example 2.5 (List in LISP)**.** Let $\mathcal{F} = \{nil, cons, 0, 1, 2, \dots\}$ where symbol $cons$ has arity 2 and other symbols have arity 0. Furthermore, let $\mathcal{K} = \{\Box_1, \Box_2\}$. Then

$$(cons(\Box_1, \Box_2)^{*,\Box_2} \cdot \Box_2 nil) \cdot \Box_1 (0 + 1 + 2 + \dots)$$

is the regular tree expressions that denotes the language of LISP-like lists of integers greater or equal than zero:

$$\{nil,\ cons(0, nil),\ cons(0, cons(5, nil)),\ cons(3, cons(2, cons(1, nil))),\ \dots\}$$

■

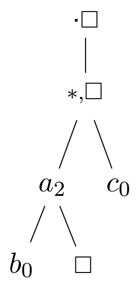


Figure 2.1: Syntax tree of the regular tree expression from example 2.4.

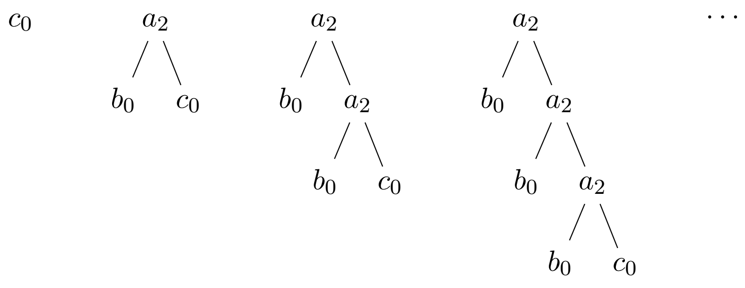


Figure 2.2: Some trees matched by the regular tree expression in example 2.4.

Chapter 3

Related Works

This chapter presents some related results in the domain of string and tree languages. Then it shows the conversion between finite tree automata and context-free grammars. Finally, Glushkov's algorithm for strings is described in order to introduce the reader to the principles of the algorithm before it is adapted for regular tree expressions and pushdown automata.

3.1 Related Works in the String Languages Domain

In 1943, McCulloch and Pitts presented a paper [MP43] in which they studied finite state systems (neural nets). Following their ideas, Kleene published a paper [Kle56] on finite automata and regular expressions where the equivalence between finite automata and regular expressions is proved. It led to numerous follow-up papers that deal with conversions between these two useful formal models.

Several algorithms solving this problem in the string domain exist [Mel03, ASU86]. At least three of them can be mentioned here: Glushkov's Algorithm [Glu61], Brzozowski's Regular Expression Derivatives Algorithm and Thompson's Construction Algorithm [Tho68]. It has also been proven that Thompson's and Glushkov's finite automata are similar. Every Glushkov's Automaton is hidden inside the corresponding finite automaton created by Thompson's algorithm [GPW98].

Brzozowski's algorithm uses regular expression derivatives in order to create finite automaton. The problem with this algorithm is that it can not be said whether two regular expressions (obtained by derivations) are describing same language. Hence, there is no guarantee that minimal deterministic automaton is obtained. For every unrecognized equivalence new state is added to the automaton which can make the state set large. The algorithm in its naive version can also get stuck in an infinite loop as new derivatives are created when equivalence is not recognized. However, as was proved in the original paper, the algorithm will always terminate if some trivial axioms are applied to the regular expressions before equivalence tests.

Thompson's algorithm constructs nondeterministic finite automaton with ε-transitions for given regular expression. It inductively creates automata for subexpressions which are then joined into automaton recognizing the language defined by the expression.

Glushkov's algorithm will be described later in section 3.4 in detail. This thesis adapts Glushkov's algorithm for trees, so the reader might get more familiar with this method beforehand.

For more detailed descriptions of these algorithms please see either original papers ([Glu61, Brz64, Tho68]), [Mel03]. Or [Pec14] where these algorithms are also implemented.

3.2 Related Works in the Tree Languages Domain

With Kleene's theorem lifted from strings to trees, the research also followed to the tree matching. It is proven that regular tree expressions describe exactly regular tree languages which can be recognized by finite tree automata[CDG$^+$07].

Kuske and Meinecke published method [KM08] of creating finite tree automata from regular tree expressions. To achieve this, they adapted partial derivatives from regular (string) expressions to regular tree expressions. However, this method does not create pushdown automaton, it creates tree automaton instead.

In 2011, Polách [PJM11, Pol11] presented a method of transforming a regular tree expression to an equivalent real-time height-deterministic pushdown automaton. This automaton reads input of ranked or unranked ordered trees either in postfix or postfix bar notation. The proposed method was inspired by Thompson's Algorithm for converting regular (string) expressions to finite automata. Pushdown automata for subexpressions are recursively created and then joined into one automaton recognizing the trees described by the expression. This method yields real-time height-deterministic pushdown automaton which can be determinised [NS07, PTJM16] however it contains many epsilon transitions.

The variation of Glushkov's automaton for regular tree expressions to finite tree automata conversion was presented in the paper of Laugerotte et al. [LSZ13]. They defined the Follow, First and Last functions for trees and proved that they can create tree automaton recognizing the same language using these functions. The result of this algorithm is homogeneous nondeterministic bottom-up finite tree automaton. Note that bottom-up tree automata can always be determinised[CDG$^+$07]. They also adapted so called ZPC structure to trees to make the computation of follow function asymptotically faster.

3.3 Regular Tree Languages, Context-Free Grammars and Pushdown Automata

One of the foundation stones of Arbology research is the observation that every tree can be linearised and the linear form is unique for each tree. At least ranked or bar notations can be mentioned. Both of these notations has prefix and postfix versions.

Regular tree languages are, in a sense, subclass of deterministic context-free languages. When considering linear tree notations, regular tree languages can be recognized by deterministic pushdown automata. However, this relation is not true if reversed. Not every deterministic context-free language is linearised regular tree language. Regular tree languages are proper subclass of deterministic context-free languages. This was shown by Janoušek and Melichar [JM09].

Janoušek and Melichar also presented transformation of a bottom-up finite tree automaton (finite automata define regular tree language) $\mathcal{A}$ to context-free grammar generating $L(\mathcal{A})$ in postfix notation. Having this grammar constructed, it is not hard to create a pushdown automaton recognizing linearised trees from $L(\mathcal{A})$, i.e., $post(L(\mathcal{A}))$. The resulting grammar will be LR(0) if the given bottom-up tree automaton is deterministic. Note that every bottom-up tree automaton can be determinised [CDG$^+$07].

The method for transformation of a bottom-up tree automata to an equivalent pushdown automata will be shown. Firstly, a context-free grammar corresponding to the bottom-up tree automaton is created. Next, conversion from the grammar to a pushdown automaton accepting by empty pushdown store is shown. Complete method with a proof can be found in [JM09].

Definition 3.1 (FTA to equivalent CFG)**.** Let $\mathcal{A} = (Q, F, Q_f, \Delta)$ be bottom up finite tree automaton. Then a context-free grammar generating $L(\mathcal{A})$ in postfix notation with appended right marker $\dashv$ is $G_{\mathcal{A}} = (N, T, P, S')$, where $N = \{S'\} \cup \{S_q : q \in Q\}$, $T = F$, and $P = \{S' \to S_q \dashv : q \in Q_f\} \cup \{S_q \to S_{q_1} S_{q_2} \cdots S_{q_n} a : a(q_1, q2, \cdots q_n) \to q \in \Delta\}$. $\triangle$

Please note that the right marker $\dashv$ is added for the purpose of accepting with empty pushdown store in the pushdown automaton corresponding to this grammar.

Definition 3.2 (CFG into PDA)**.** Let $\mathcal{A} = (Q, F, Q_f, \Delta)$ be bottom up finite tree automaton and $G_{\mathcal{A}} = (N, T, P, S')$ the corresponding context-free grammar generating trees from $L(\mathcal{A})$ in their postfix notation. The corresponding pushdown automaton is $M_{\mathcal{A}} = (\{q\}, T, G, \delta, q, \perp, \emptyset)$, where $G = Q \cup \{\perp, \dashv\}$. Transition function is defined as: $\delta(q, \dashv, \perp\ \alpha) = (q, \varepsilon)\ \forall S' \to \alpha \dashv \in P$ and $\delta(q, a, \alpha) = (q, A)\ \forall A \to \alpha a \in P,\ A \neq S'$. $\triangle$

Proposition 3.1 (Regular tree languages and CFL)**.** Deterministic context-free languages are proper superset of languages described by regular tree expressions in their postfix linear notation.

Corollary 3.1. From proposition 3.1 and Kleene's theorem for regular tree languages[CDG+07] it is more than obvious that it is possible to convert regular tree expression to a pushdown automaton recognizing linearised regular tree language.

3.4 Glushkov's Algorithm

In 1961, Victor M. Glushkov presented a method for conversion of regular (string) expression to finite automaton. This method, nowadays known as Glushkov's algorithm or Glushkov's nondeterministic finite automaton yields a nondeterministic finite automaton without ε-transitions [Glu61, CZ00, Mel03].

Glushkov's algorithm also produces quite small automaton in comparison with Thompson's algorithm and has no ε-transitions which is advantageous when simulating. Let n be the number of symbols of the alphabet occurring in the regular expression. Then the resulting Glushkov's automaton has exactly $n + 1$ states and at most n^2 transitions. It is also homogeneous, i.e., all transitions leading to certain state are labelled with the same symbol (see definition 1.24).

The idea behind this algorithm is simple. Let us consider regular expression E over alphabet Σ. Every occurrence of symbols from Σ (only symbols, neither ε nor operators) is subscripted in a way that two distinct occurrences of the same symbol have different numbers assigned. For example, starting from $E = (a+\varepsilon)bc+d^*$, subscripted regular expression $E' = (a_1+\varepsilon)b_2c_3+d_4^*$ is obtained. The set of these subscripted symbols is denoted by $\mathrm{Pos}(E)$.

Definition 3.3 (σ mapping)**.** σ maps subscripted symbol from $\mathrm{Pos}(E)$ to corresponding symbol in Σ, i.e., $\sigma : \mathrm{Pos}(E) \mapsto \Sigma$. For example: Let us consider $a \in \Sigma$ and $a_1 \in \mathrm{Pos}(E)$. Then $\sigma(a_1) = a$. △

Positions (i.e., elements of $\mathrm{Pos}(E)$ set) can be used as states of the finite automaton. These states will then represent up to what symbol the input string was read. If one symbol a_i can appear only after a_j, then there will be transition between states a_i and a_j.

Before the algorithm is presented, few more functions that are used in the algorithm are defined.

Definition 3.4 (Pos Set)**.** Let E be a regular expression over alphabet Σ and E' its subscripted counterpart. $\mathrm{Pos}(E) = \{x_i : \sigma(x_i) \in \Sigma,\ x_i \text{ occurs in } E'\}$. This set contains all the subscripted symbols that occurs in E'. For example, for $E' = a_1^* + (b_2c_3^*) + \varepsilon$ the set $\mathrm{Pos}(E) = \{a_1, b_2, c_3\}$.

This function is defined in equation (3.1). △

Definition 3.5 (First Set)**.** Let E be a regular expression over alphabet Σ and E' its subscripted counterpart. $\mathrm{First}(E') = \{a_j : a_j w \in L(E')\}$, i.e., all the symbols that can be in first position of some string from the language denoted by labelled regular expression E.

This function computation is defined in equation (3.2). △

Definition 3.6 (Last Set)**.** Let E be a regular expression over alphabet Σ and E' its subscripted counterpart. $\mathrm{Last}(E') = \{a_i : wa_i \in L(E')\}$, i.e., all the symbols that can be in last position of some string from the language denoted by labelled regular expression E.

This function computation is defined in equation (3.3). △

Definition 3.7 (Follow Set)**.** Let E be a regular expression over alphabet Σ and E' its subscripted counterpart. $\mathrm{Follow}(E', a_i) = \{a_j : ua_ia_jv \in L(E')\}$, i.e., all the symbols a_j that can occur after a_i in some string from the language denoted by labelled regular expression E.

This function computation is defined in equation (3.4). △

The functions First, Last and Follow can be also defined inductively using the rules which can be seen in equations (3.1) to (3.4) [CZ00].

$$\begin{aligned}
\mathrm{Pos}(\emptyset) &= \mathrm{Pos}(\varepsilon) = \emptyset \\
\mathrm{Pos}(a) &= \{a\} \\
\mathrm{Pos}(E_1 + E_2) &= \mathrm{Pos}(E_1) \cup \mathrm{Pos}(E_2) \\
\mathrm{Pos}(E_1 \cdot E_2) &= \mathrm{Pos}(E_1) \cup \mathrm{Pos}(E_2) \\
\mathrm{Pos}(E^*) &= \mathrm{Pos}(E)
\end{aligned} \tag{3.1}$$

$$\begin{aligned}
\mathrm{First}(\emptyset) &= \mathrm{First}(\varepsilon) = \emptyset \\
\mathrm{First}(a) &= \{a\} \\
\mathrm{First}(E_1 + E_2) &= \mathrm{First}(E_1) \cup \mathrm{First}(E_2) \\
\mathrm{First}(E_1 \cdot E_2) &= \begin{cases} \mathrm{First}(E_1) & \text{if } \varepsilon \notin L(E_1) \\ \mathrm{First}(E_1) \cup \mathrm{First}(E_2) & \text{if } \varepsilon \in L(E_1) \end{cases} \\
\mathrm{First}(E^*) &= \mathrm{First}(E^+) = \mathrm{First}(E)
\end{aligned} \tag{3.2}$$

$$\begin{aligned}
\mathrm{Last}(\emptyset) &= \mathrm{Last}(\varepsilon) = \emptyset \\
\mathrm{Last}(a) &= \{a\} \\
\mathrm{Last}(E_1 + E_2) &= \mathrm{Last}(E_1) \cup \mathrm{Last}(E_2) \\
\mathrm{Last}(E_1 \cdot E_2) &= \begin{cases} \mathrm{Last}(E_2) & \text{if } \varepsilon \notin L(E_2) \\ \mathrm{Last}(E_1) \cup \mathrm{Last}(E_2) & \text{if } \varepsilon \in L(E_2) \end{cases} \\
\mathrm{Last}(E^*) &= \mathrm{Last}(E^+) = \mathrm{Last}(E)
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
\mathrm{Follow}(\varepsilon, x) &= \mathrm{Follow}(\emptyset, x) = \mathrm{Follow}(a, x) = \emptyset \\
\mathrm{Follow}(E_1 + E_2, x,) &= \begin{cases} \mathrm{Follow}(E_1, x) & \text{if } x \in \mathrm{Pos}(E_1) \\ \mathrm{Follow}(E_2, x) & \text{if } x \in \mathrm{Pos}(E_2) \end{cases} \\
\mathrm{Follow}(E_1 \cdot E_2, x) &= \begin{cases} \mathrm{Follow}(E_1, x) & \text{if } x \in \mathrm{Pos}(F) \setminus \mathrm{Last}(F) \\ \mathrm{Follow}(E_1, x) \cup \mathrm{First}(E_2) & \text{if } x \in \mathrm{Last}(F) \\ \mathrm{Follow}(E_2, x) & \text{if } x \in \mathrm{Pos}(G) \end{cases} \\
\mathrm{Follow}(E^*, x) &= \mathrm{Follow}(E^+, x) \\
\mathrm{Follow}(E^+, x) &= \begin{cases} \mathrm{Follow}(E, x) & \text{if } x \in \mathrm{Pos}(E) \setminus \mathrm{Last}(E) \\ \mathrm{Follow}(E, x) \cup \mathrm{First}(F) & \text{if } x \in \mathrm{Last}(E) \end{cases}
\end{aligned} \tag{3.4}$$

Now, everything that is needed for the Glushkov's algorithm is defined. Recall that input of this algorithm is a regular expression and it yields a nondeterministic finite automaton as seen in algorithm 3.1.

Example 3.1 (Construction of Glushkov's NFA)**.** Let $\Sigma = \{a, b, c\}$ and $E = ab^*a + ac + (b^*a(\varepsilon + b))^*$.

Firstly, the symbol occurrences in the expression E are numbered:

$$E' = a_1 b_2^* a_3 + a_4 c_5 + (b_6^* a_7(\varepsilon + b_8))^*.$$

Now, sets $\mathrm{First}(E')$, $\mathrm{Last}(E')$ and $\mathrm{Follow}(E', a_i)$ $\forall a_i \in \mathrm{Pos}(E)$ are computed. Notice that $\varepsilon \in L(E)$.

$$\begin{aligned}
\mathrm{First}(E') &= \{a_1, a_4, b_6, a_7\} & \mathrm{Last}(E') &= \{a_3, c_5, a_7, b_8\} \\
\mathrm{Follow}(E', a_1) &= \{b_2, a_3\} & \mathrm{Follow}(E', c_5) &= \{\} \\
\mathrm{Follow}(E', b_2) &= \{b_2, a_3\} & \mathrm{Follow}(E', b_6) &= \{b_6, a_7\} \\
\mathrm{Follow}(E', a_3) &= \{\} & \mathrm{Follow}(E', a_7) &= \{b_6, a_7, b_8\} \\
\mathrm{Follow}(E', a_4) &= \{c_5\} & \mathrm{Follow}(E', b_8) &= \{b_6, a_7\}
\end{aligned}$$

Algorithm 3.1: Glushkov's NFA Construction

Input: Regular expression E over alphabet Σ.
Output: Nondeterministic finite automaton A such that $L(A) = L(E)$.

1. Subscript all the occurrences of symbols of Σ with positive integer numbers $1, 2, \ldots, n$. Two distinct occurrences of same symbol must have different numbers assigned. Denote the subscripted regular expression by E'.
2. Compute sets $\mathrm{Pos}(E)$, $\mathrm{First}(E')$, $\mathrm{Follow}(E', a)\ \forall a \in \mathrm{Pos}(E)$ and $\mathrm{Last}(E')$.
3. Create set of states $Q = \{q_0\} \cup \mathrm{Pos}(E)$ and set of final states $F = \mathrm{Last}(E') \cup \{q_0 : \varepsilon \in L(E)\}$.
4. δ is defined:
   a) $a_i \in \delta(q_0, \sigma(a_i))\ \forall a_i \in \mathrm{First}(E')$,
   b) $b_i \in \delta(a_i, \sigma(b_i))\ \forall b_i \in \mathrm{Follow}(E', a_i)\ \forall a_i \in \mathrm{Pos}(E)$.
5. Resulting automaton $A = (Q, \Sigma, \delta, q_0, F)$.

Construction of a nondeterministic finite automaton is now straightforward.

$$A = (\{q_0, a_1, b_2, a_3, a_4, c_5, b_6, a_7, b_8\}, \{a, b, c\}, \delta, q_0, \{q_0, a_3, c_5, a_7, b_8\})$$

The set of transition rules δ is defined by following transitions:

$$\begin{array}{llll}
\delta(q_0, a) = \{a_1, a_4, a_7\} & \delta(b_2, a) = \{a_3\} & \delta(b_6, a) = \{a_7\} & \delta(b_8, a) = \{b_6\} \\
\delta(q_0, b) = \{b_6\} & \delta(b_2, b) = \{b_2\} & \delta(b_6, b) = \{b_6\} & \delta(b_8, b) = \{a_7\} \\
\delta(a_1, a) = \{a_3\} & \delta(a_4, c) = \{c_5\} & \delta(a_7, a) = \{a_7\} & \\
\delta(a_1, b) = \{b_2\} & & \delta(a_7, b) = \{b_6, b_8\} &
\end{array}$$

Resulting nondeterministic finite automaton A can be seen on figure 3.1. ■

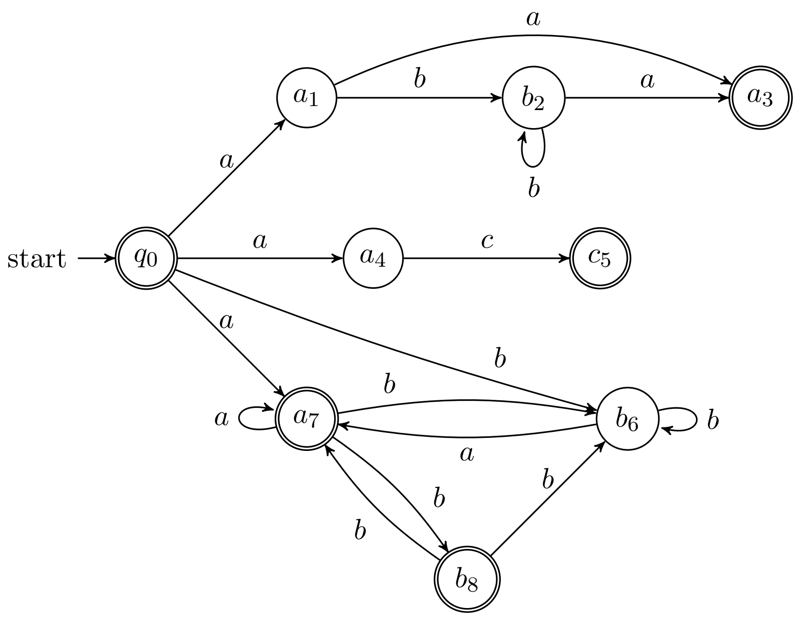


Figure 3.1: Automaton created by algorithm 3.1 in example 3.1.

Chapter **4**

Converting Regular Tree Expressions to Pushdown Automata

This chapter presents the method for creating a real-time height-deterministic pushdown automaton recognizing the postfix notation of trees described by regular tree expression. Firstly, ideas that are building stones of this method are presented. Then the algorithm is shown and its properties are formally discussed. Finally, some complex examples are presented.

4.1 Analysis

Before stating the algorithm itself, some ideas have to be presented and defined.

Firstly, it is shown how to create a pushdown automaton for a single tree in postfix notation. There is also a constraint that the automaton should be determinisable to make the simulations easier because runs of nondeterministic pushdown automata can not be simulated easily. From previous chapter it is known that a language of postfix notations of trees from a regular tree language is a proper subclass of deterministic context-free languages.

Then, ideas how to analyse the structure of a regular tree expression are described. The regular tree expression identifies a regular tree language.

Finally, a method how to create a pushdown automaton utilizing described regular tree expression analysis is presented.

4.1.1 Pushdown Automaton For Postfix Notation

The conversion method is expected to create a pushdown automaton recognizing language of linearised trees in postfix notation described by a regular tree expression. Recall how postfix notation works (see definition 1.44).

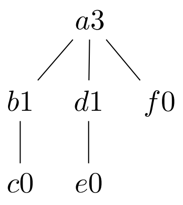


Figure 4.1: Tree t over alphabet $\Sigma = \{a3, b1, c0, d1, e0, f0\}$.

Notice that in the postfix notation, when a node a occurs, all of its subtrees and subtrees of nodes on the same level that are to the left of node a (ordered trees are considered), have already appeared in the string. This comes trivially from the notation definition.

To illustrate previous sentence, consider tree t from figure 4.1. Postfix notation of tree t is $post(t) = c0\, b1\, e0\, d1\, f0\, a3$. Observe, that in $post(t)$, when $d1$ appears, $c0\, b1$ (left sibling of $d1$ and its subtree) and $e0$ (subtree of $d1$) already appeared.

Observation from previous paragraphs is used to make the tree accepted by pushdown automaton. Information about what subtrees the automaton read is stored on the pushdown store. Input symbol is allowed to be processed only when all of the children subtrees have been read. This is guaranteed by pushdown store operations. Subtrees can be identified by their root symbol. Considering the tree t from figure 4.1, $a3$ is read only when $b1\, d1\, f0$ are on the pushdown store and therefore it is sure that the subtrees of $a3$ were read.

Now, pushdown store operations are analysed. Basically, the information about read symbol needs to be stored for later use (to check that subtree was read). For simplicity, let us now assume that pushdown store alphabet is equal to the input alphabet set. So when symbol a is read, it is also pushed on the top of the pushdown store to keep the information. However, before that, all the symbols representing subtrees of a must be popped as the information is no longer required and the subtree is now represented by a symbol solely.

This way, it is more than obvious that this is very helpful in determining whether correct tree was read. When the input is read, on the top of the pushdown store should be symbol corresponding to $root(t)$ and directly underneath it should be the initial pushdown store symbol. This is trivial observation.

Example 4.1. Let t be a tree t from figure 4.1. Postfix notation of tree t is $post(t) = c0\, b1\, e0\, d1\, f0\, a3$.

To illustrate previous method, let us assume that $L = \{t\}$. Pushdown automaton recognizing trees from L in their linear postfix notation will be created.

Statements in previous paragraphs imply that transitions must be created for every symbol of the alphabet of t. Single state pushdown automaton that accepts by empty pushdown store is considered. Recall that the top of the pushdown store is situated on the right. Initial pushdown store symbol is $\perp$.

$$\delta(q, c0, \varepsilon) = \{(q, c0)\} \tag{4.1}$$
$$\delta(q, b1, c0) = \{(q, b1)\} \tag{4.2}$$
$$\delta(q, e0, \varepsilon) = \{(q, e0)\} \tag{4.3}$$
$$\delta(q, d1, e0) = \{(q, d1)\} \tag{4.4}$$
$$\delta(q, f0, \varepsilon) = \{(q, f0)\} \tag{4.5}$$
$$\delta(q, a3, b1\, d1\, f0) = \{(q, a3)\} \tag{4.6}$$
$$\delta(q, \varepsilon, \perp\ a3) = \{(q, \varepsilon)\} \tag{4.7}$$

Now, let us explain why the transition rules were created like this. Transition rules marked by equations (4.1), (4.3) and (4.5) surely work as was described: Whenever they read a symbol (a constant symbol here), they pop all the children (constants have no children) and push the input symbol on the top of the pushdown store. Transition rules marked by equations (4.2) and (4.4) correspond to input symbols $b1$ and $d1$. Again, they pop all children ($c0$ and $f0$ respectively) and push the input symbol on the pushdown store. The rule equation (4.6) pops all the children (top of the pushdown store is situated on the right, hence the rightmost children is on the top of the pushdown store) and push $a3$ on the top of the pushdown store. The last rule, equation (4.7) only checks whether t was accepted (i.e., the pushdown store content is $\perp\ a3$ and cleans the pushdown store contents to accept.

The reader now should see what is happening on the pushdown store, but let us show the configurations of the pushdown automaton when reading $post(t)$. Notation $\vdash^i$ means that transition rule i from the range of equations (4.1) to (4.7) is used.

$$\begin{aligned}
&(q,\ c0\, b1\, e0\, d1\, f0\, a3,\ \perp) \vdash^{4.1}\\
&\quad(q,\ b1\, e0\, d1\, f0\, a3,\ \perp c0) \vdash^{4.2}\\
&\quad\quad(q,\ e0\, d1\, f0\, a3,\ \perp b1) \vdash^{4.3}\\
&\quad\quad\quad(q,\ d1\, f0\, a3,\ \perp b1\, e0) \vdash^{4.4}\\
&\quad\quad\quad\quad(q,\ f0\, a3,\ \perp b1\, d1) \vdash^{4.5}\\
&\quad\quad\quad\quad\quad(q,\ a3,\ \perp b1\, d1\, f0) \vdash^{4.6}\\
&\quad\quad\quad\quad\quad\quad(q,\ \varepsilon,\ \perp a3) \vdash^{4.7}\\
&\quad\quad\quad\quad\quad\quad(q,\ \varepsilon,\ \varepsilon)
\end{aligned}$$

■

4.1.2 Analysing Regular Tree Expression

In the previous subsection, it was assumed that only one tree was processed. Such problem is easy as pushdown automaton is not even required (it is regular language because there as only one finite string in it). Now, what if a set of trees described by regular tree expression (a regular tree language) is required to be recognized? A regular tree expression is required to be analysed first.

Recall that in Glushkov's algorithm, the regular (string) expression is analysed by functions First, Last and Follow. This analysis provides the information about which symbols can be matched at the beginning, at the end and after some symbol respectively. There was also a function Pos that assigned unique indices to every occurrence of symbols in the regular expression. These functions need to be adapted for regular tree expressions.

The pushdown automaton recognizes trees in their linearised postfix notation. Therefore the last symbol of the input string will always be the root of the tree (trivial observation from definition 1.44). Modified First function will return a set of symbols that can be root of some tree described by a regular tree expression. The definition will be given later. Please note that the First function has nothing to do with a first symbol of string (representing a linearised tree) but rather with the first symbol of a tree, which is in fact a root of a tree.

The Follow function is little bit trickier. In string, one symbol can be followed by multiple single symbols. In trees, one symbol is followed by its children where a number of children depends on the arity of the symbol. Hence symbol a is followed by a set of tuples of symbols of size $arity(a)$.

Pos function to differentiate between multiple occurrences of the same symbol is also utilized. The set this function returns contains all the indexed occurrences of all symbols appearing in a expression. Only symbols of Σ are indexed. The labels do not really matter, the only constraint is that two distinct occurrences of the same symbol have different labels.

The equivalent of Last function is not required for trees in postfix notation it is in fact covered by Follow. It is an open problem whether some adaptation of this method for another linear notations will utilize this function. However, it can probably be said that Last function is equivalent to Follow function for symbols with zero arity, i.e., symbols with no children.

4.1.2.1 Computation of the First Set

Recall that the First function returns set of labelled symbols from Pos that can be root of any tree described by the regular tree expression.

Based on the definition of the regular tree expressions, First function is defined inductively.

$$
\begin{aligned}
\text{First}(\emptyset) &= \emptyset \\
\text{First}(a(E_1, E_2, \dots, E_n)) &= \{a\} \\
\text{First}(E_1 + E_2) &= \text{First}(E_1) \cup \text{First}(E_2) \\
\text{First}(E_1 \cdot \Box E_2) &= \begin{cases} \text{First}(E_1) & \text{if } \Box \notin \text{First}(E_1) \\ (\text{First}(E_1) \setminus \{\Box\}) \cup \text{First}(E_2) & \text{if } \Box \in \text{First}(E_1) \end{cases} \\
\text{First}(E^*) &= \{\Box\} \cup \text{First}(E)
\end{aligned} \tag{4.8}
$$

Proposition 4.1 (First function)**.** Function First(E') from equation (4.8) correctly returns all occurrences of symbols that can be root of any tree described by a regular tree expression E.

For a proof of proposition 4.1 see section 4.3.1.

Example 4.2. Let E be a regular tree expression from figure 4.2, then First(E') = $\{b0_2,\ a2_1,\ a2_3\}$. The process of computation is illustrated on figure 4.3. ■

4.1.2.2 Computation of the Follow Set

The Follow function analyses the structure of a regular tree expression. It computes a set of tuples of symbols which can be direct descendants of a subscripted symbol from $\mathcal{F}$ in a regular tree expression. Recall that in the Glushkov's Automaton construction for regular (string) expressions the analogous function returns a set of symbols which could appear after some occurrence of symbol in the expression. This is because of the linear structure of the strings. In the trees, all possible direct descendants tuples are returned.

Based on the definition of the regular tree expressions, the computation of the Follow function is defined using an algorithm 4.1. The idea behind algorithm is to maintain so called *substitution map* while recursively traversing a regular expression syntax tree. The substitution map contains roots of all possible trees that can get substituted for each $\Box \in \mathcal{K}$. If $\Box$ symbol appears between direct descendants of the symbol for which follow is being computed, it will get substituted by elements of substitution map for given $\Box$ symbol. Note that it is possible that another element of $\mathcal{K}$ is present in the substitution table for $\Box$. This happens when the right element of substitution operator is also $\mathcal{K}$ element. So, some preprocessing is required to handle this as is later shown.

In the algorithm 4.1 function `ReplaceConstants` is used. This function takes the substitution map and tuple of children and replaces all symbols from $\mathcal{K}$ in the tuple of children with all possible elements for which the symbol can get substituted. This however makes the output of the function possibly exponential as shown later in the thesis.

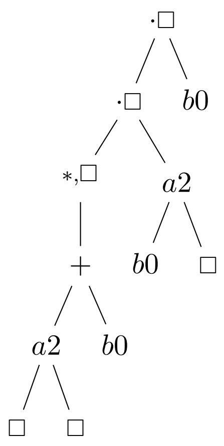


Figure 4.2: Regular tree expression $E = (a2(\square,\square) + b0)^{*,\square} \cdot \square\ a2(b0,\square) \cdot \square\ b0$

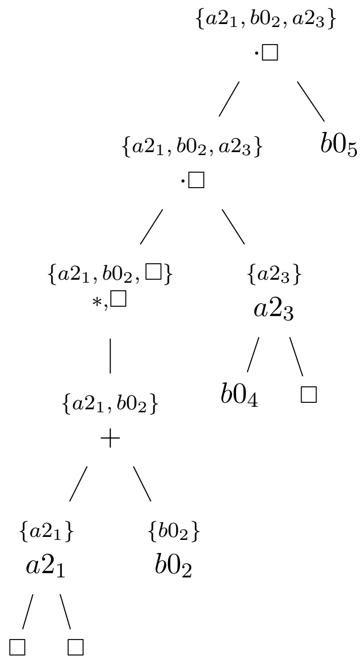


Figure 4.3: Regular tree expression E' from figure 4.2 when processed with First function. Symbols in $\{\}$ are elements of First set for given subexpression. Those subexpressions which have no elements of First set are not needed for the computation.

To illustrate previous paragraphs consider a case when processing a subtree of regular tree expression $a(\square_1)$. Also assume that the substitution map looks like this: $\{\square_1 \rightarrow \{b, \square_2\}, \square_2 \rightarrow \{c\}\}$. Then all possible follow tuples of symbol a are: $\{(b), (c)\}$. The children tuple (c) appeared in the set because symbol $\square_2$ was in the substitution map for $\square_1$ symbol, therefore all elements of $\square_2$ also belong to $\square_1$.

Proposition 4.2 (Follow function)**.** Function Follow(E', a) correctly returns a set of tuples representing all possible tuples of direct children of a subscripted node a.

For a proof of proposition 4.2 see section 4.3.1.

Example 4.3. Let E be a regular tree expression from figure 4.2, then First$(E') = \{b0_2,\ d1_1,\ a2_3\}$. The substitution map for individual nodes of expression is illustrated on figure 4.4.

Computation of Follow$(E', a2_1)$ is shown: By the function definition, the computation recursively descents to the node $a2_1(\square, \square)$ and the computation takes the if branch where function $ReplaceConstants((\square, \square), subMap)$ is called. The substitution map is now as follows: $\{\square \rightarrow \{a2_3, a2_1, b0_2\}\}$. Now all possible combinations how to substitute $\square$ elements in children tuple $(\square, \square)$ are computed. Therefore the result is: $\{(a2_3, a2_3), (a2_3, a2_1), (a2_3, b0_2),$ $(a2_1, a2_3), (a2_1, a2_1), (a2_1, b0_2), (b0_2, a2_3), (b0_2, a2_1), (b0_2, b0_2)\}$.

Follow$(E', a2_3) = \{(b0_4, b0_5)\}$. Follow of symbols $b0_2$, $b0_4$ and $b0_5$ is $\emptyset$, these symbols have zero arity.

$\blacksquare$

4.1.3 Pushdown Automaton Construction

In the previous section it is shown how to compute First and Follow sets. Therefore it is known what symbols are the last to be read (even though we call the function First, this is because of postfix notation) and what are the roots of subtrees (direct children) of certain node. We can use this information to create pushdown automaton transitions.

Single state pushdown automaton that accepts by empty pushdown store is created. To make this possible, an end-of-string marker $\dashv$ needs to be appended to the end of input (which is a postfix notation of a tree). This is necessary as information about what has been read is kept on the pushdown store and at the same time it tells what is yet to be read. If the pushdown store is empty (or to be more precise, if only initial pushdown store symbol and symbol representing root of the tree is on the pushdown store), whole linearised tree was read. Thus the string is accepted only when the pushdown store is empty after processing.

Simple definition of φ mapping is presented first. It maps a tuple from the Follow set to a string of pushdown store symbols.

Algorithm 4.1: Computation of Follow function.

```
Function Follow($E$, $a$)
    $subMap \leftarrow$ initialize map
    return FollowRec($E$, $a$, $subMap$)

Function FollowRec($E$, $a$, $subMap$)
    switch $E$ do
        case $E_1 + E_2$ do
            return FollowRec($E_1$, $a$, $subMap$) $\cup$ FollowRec($E_2$, $a$,
             $subMap$)
        end
        case $E_1 \cdot \Box E_2$ do
            $subMapL \leftarrow subMap$              /* copy map */
            $subMapL[\Box] =$ First($E_2$)  /* replace mapping for $\Box$ */
            return FollowRec($E_1$, $a$, $subMapL$) $\cup$ FollowRec($E_2$, $a$,
             $subMap$)
        end
        case $E_1^{*,\Box}$ do
            $subMap[\Box] = subMap[\Box] \cup$ First($E_1$)
            return FollowRec($E$, $a$, $subMap$)
        end
        case $f(E_1, E_2, \ldots, E_n)$ do
            if $a = f$ then
                return ReplaceConstants($children$, $subMap$)
            else
                $ret \leftarrow \emptyset$
                for $E_i$ in $E_1, E_2, \ldots, E_n$ do
                    $ret = ret \cup$ FollowRec($E_i$, $a$, $subMap$)
                end
                return $ret$
            end
        end
        case $\emptyset$ do
            return $\emptyset$
        end
    end
```

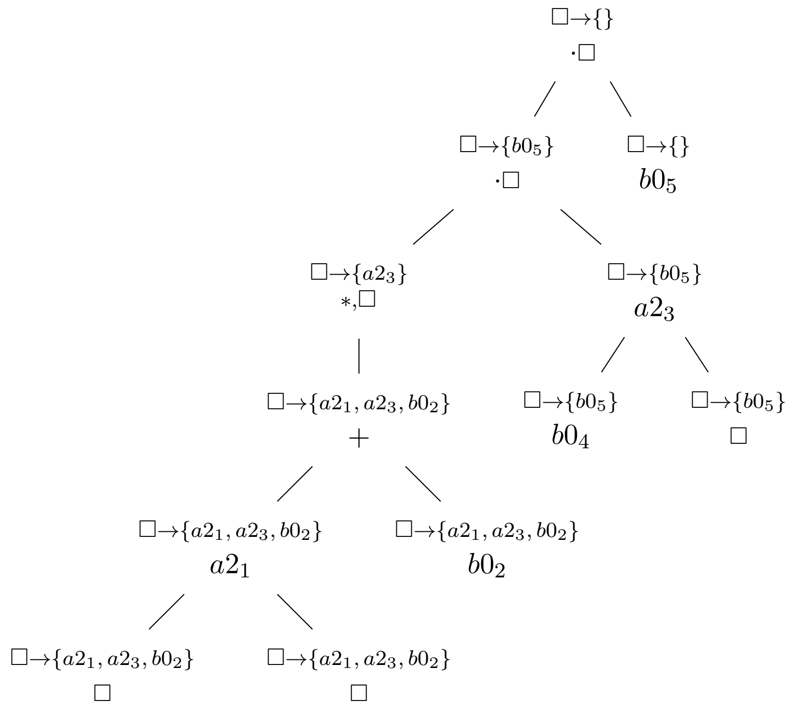


Figure 4.4: Regular tree expression E' from figure 4.2 with substitution mapping.

Definition 4.1 (φ)**.** We define function $\varphi : \Sigma^n \mapsto G^*$ (let G be a pushdown store alphabet), which maps an element (tuple) of Follow(E', a) to a string of pushdown store symbols. $\triangle$

Example 4.4. Let E' be a labelled regular tree expression and let a tuple $f = (a_1, b_2, c_3) \in$ Follow(E', a_i). Then $\varphi(f) = a_1 b_2 c_3$.

Also $\sigma(a_1) = a$ (recall σ mapping from definition 3.3). $\blacksquare$

Mapping δ (automaton transition function) is based on the observations from previous chapters. It was stated that symbol a can be read only when all its subtrees were read. So pushdown store operations for symbol a are that all possible tuples of children are removed from the top of the pushdown store and symbol corresponding to a is pushed.

Pushdown automaton A recognizing postfix notations of trees described by a regular tree expression E over $\mathcal{F} \cup \mathcal{K}$ is defined in the following way:

- set of states Q is equal only to $\{q\}$ because single-state automaton is created,
- alphabet is equal to the non-constant alphabet of E, i.e., $\mathcal{F}$ and $\dashv$ symbol,

- pushdown store alphabet Γ consists of the initial pushdown store symbol $\perp$ and all the elements of $\mathrm{Pos}(E)$,
- set of transition rules δ is constructed as follows:
  - For all symbols $c_i \in \mathrm{Pos}(E)$, $\sigma(c_i) \in \mathcal{F}0$ transition rule (q, c_i) is added to $\delta(q, \sigma(c_i), \varepsilon)$ set,
  - for all symbols $a_i \in \mathrm{Pos}(E), \sigma(c_i) \notin \mathcal{F}_0$ and every $f \in \mathrm{Follow}(E', a_i)$ transition rule (q, a_i) is added to $\delta(q, \sigma(a_i), \varphi(f))$,
  - for all symbols $a_i \in \mathrm{First}(E')$ transition rule (q, ε) is added to $\delta(q, \dashv, \perp a_i)$ set.
- Resulting automaton $A = (\{q\}, \mathcal{F} \cup \{\dashv\}, \Gamma, \delta, \perp, q, \{\})$. Automaton accepts the string by empty pushdown store. The top of the pushdown store is situated on the right.

Note that this automaton is neither real-time height-deterministic nor another (known) determinisable subclass of pushdown automata. Real-time height-deterministic pushdown automata must accept input by entering final state after the string was read [NS07, PTJM16]. However, it is trivial to modify this automaton in the way it accepts by final state. Another state, f, which is also final state is added. Mapping δ must be altered too. All the transitions which were created by the $\mathrm{First}(E')$ rule, i.e., the transitions reading $\dashv$ and checking for correct final pushdown store content, will now lead to final state f instead of q.

To clarify, let us define the altered pushdown automaton. This time it will not violate the final state acceptance condition which is mandatory for real-time height-deterministic pushdown automata.

Pushdown automaton A (with final state) recognizing postfix notations of trees described by regular tree expression E over alphabet $\mathcal{F} \cup \mathcal{K}$ in the following way:

- set of states is equal to $\{q, f\}$,
- alphabet is equal to the non-constant alphabet of E, i.e., $\mathcal{F}$ and $\dashv$ symbol,
- pushdown store alphabet Γ consists of the initial pushdown store symbol $\perp$ and all the elements of $\mathrm{Pos}(E)$,
- mapping δ consists of these transitions:
  - For all symbols $c_i \in \mathrm{Pos}(E)$, $\sigma(c_i) \in \mathcal{F}_0$ transition rule (q, c_i) is added to $\delta(q, \sigma(c_i), \varepsilon)$ set,
  - for all symbols $a_i \in \mathrm{Pos}(E), \sigma(c_i) \notin \mathcal{F}_0$ and every $f \in \mathrm{Follow}(E', a_i)$ transition rule (q, a_i) is added to $\delta(q, \sigma(a_i), \varphi(f))$,

  - for all symbols $a_i \in \text{First}(E')$ transition rule (f, ε) is added to $\delta(q, \dashv, \perp a_i)$ set.

- Resulting automaton $A = (\{q, f\}, \mathcal{F} \cup \{\dashv\}a, \Gamma, \delta, \perp, q, \{f\})$. Automaton accepts string by entering final state after the string was read. The top of the pushdown store is on the right.

Notice that change of pushdown store height for every symbol in input alphabet is always the same. When reading symbol a, $arity(a)$ symbols are always popped from the pushdown store exactly 1 symbol is pushed to the pushdown store. The only exception is for $\dashv$ symbol. In this case, two symbols are popped and nothing is pushed on the pushdown store. This observation comes handy later in section 4.3 while discussing if this method yields real-time height-deterministic pushdown automaton.

Example 4.5. The example with regular tree expression from figure 4.2 is continued. Sets First and Follow were computed in examples 4.2 and 4.3.

The pushdown automaton $A = (\{q, f\}, \mathcal{F} \cup \{\dashv\}, \Gamma, \delta, \perp, q, \{f\})$ is constructed according to algorithm 4.2. The input alphabet set is equal to $\{a2,\ b0,\ \dashv\}$ and the pushdown store alphabet is $\Gamma = \{\perp,\ a2_1,\ b0_2,\ a2_3,\ b0_4,\ b0_5\}$.

Mapping δ is constructed as follows:

$$\begin{array}{rlrl}
\delta(q, b0, \varepsilon) &= \{(q, b0_2), (q, b0_4), (q, b0_5)\} & & \\
\delta(q, a2, a2_3a2_3) &= \{(q, a2_1)\} & \delta(q, a2, a2_3a2_1) &= \{(q, a2_1)\} \\
\delta(q, a2, a2_3b0_2) &= \{(q, a2_1)\} & \delta(q, a2, a2_1a2_3) &= \{(q, a2_1)\} \\
\delta(q, a2, a2_1a2_1) &= \{(q, a2_1)\} & \delta(q, a2, a2_1b0_2) &= \{(q, a2_1)\} \\
\delta(q, a2, b0_2a2_3) &= \{(q, a2_1)\} & \delta(q, a2, b0_2a2_1) &= \{(q, a2_1)\} \\
\delta(q, a2, b0_2b0_2) &= \{(q, a2_1)\} & & \\
\delta(q, a2, b0_4b0_5) &= \{(q, a2_3)\} & & \\
\delta(q, \dashv, \perp a2_1) &= \{(f, \varepsilon)\} & \delta(q, \dashv, \perp b0_2) &= \{(f, \varepsilon)\} \\
\delta(q, \dashv, \perp a2_3) &= \{(f, \varepsilon)\} & &
\end{array}$$

■

Example 4.6. Regular tree expression from figure 4.2 describes for instance following two trees from figure 4.5. Postfix notation of the first tree is $b0\,b0\,a2$. The second tree is linearised as $b0\,b0\,a2\,b0\,b0\,a2\,a2\,b0\,a2$.

Now, both these strings (with $\dashv$ marker appended) are accepted by the automaton. Beware of nondeterminism when simulating run.

The first string is accepted correctly with two ways, either with $a2_1$ on the top or $a2_3$ on the top of the pushdown store before reading $\dashv$. The first

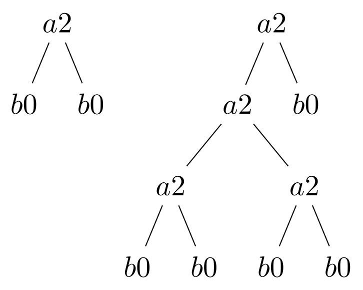


Figure 4.5: Some trees matched by the regular tree expression from figure 4.2.

possibility comes directly from iteration and then no substitution was made. The second way is with $a2_3$ symbol on the top of the pushdown store. This happens when $\Box$ is matched in iteration subtree and then substituted by the tree with $a2_3$ as a root. ∎

4.1.4 Homogeneous Automaton

Glushkov's algorithm for regular (string) expressions creates homogeneous finite automaton (see definition 1.24). There is no such notion defined for pushdown automata.

Something similar to homogeneity can be observed in the created automaton. Notice that for every pushdown store symbol b (except the initial symbol) there exists exactly one input symbol a in a way that when symbol b appears on the top of the pushdown store it is always after a move with input symbol a was made. Also empty pushdown store can be reached only after a move that reads symbol $\dashv$. It is not known yet whether this property can be regarded as homogeneity and it might be interesting topic for research.

For homogeneous finite automata there exists a theorem about the number of states obtained after determinisation. Upper bound for number of states is $\sum_{a\in\Sigma}(2^{|Q(a)|} - |\Sigma| + 1)$[Hol15]. It is not known whether something similar holds also for determinisable pushdown automata.

4.2 Algorithm

Now, algorithm 4.2 converting a regular tree expression into equivalent pushdown automaton is presented. Functions First, Follow and Pos from previous section are used.

Let us state some theorems about algorithm 4.2.

Theorem 4.1 (Language equivalence)**.** Algorithm 4.2 creates pushdown automaton A such that $L(A) = \mathit{post}(L(E))$, where E is given regular tree expression.

Algorithm 4.2: Glushkov's hdPDA Construction

Input: Regular tree expression E over alphabet Σ.

Output: Real-time height-deterministic pushdown automaton A such that $L(A) = L(post(E))$.

1. Label all the occurrences of symbols of Σ with numbers $1, 2, \dots, n$. Distinct occurrences of the same symbol must have different labels assigned. Insert all labelled symbols into the $\mathrm{Pos}\, E$ set. We denote labelled expression by E'.
2. Compute sets $\mathrm{First}(E')$ and $\mathrm{Follow}(E', a)$ $\forall a \in \mathrm{Pos}(E)$ by the algorithms from previous sections.
3. Create set of states $Q = \{q, f\}$ and set of final states $F = \{f\}$.
4. Let the pushdown store alphabet $\Gamma = \{\bot \cup \mathrm{Pos}(E)\ \}$.
5. Mapping δ is defined:
   a) For all symbols $c_i \in \mathrm{Pos}(E)$, $\sigma(c_i) \in \mathcal{F}_0$ add the transition $(q, c_i) \in \delta(q, \sigma(c_i), \varepsilon)$,
   b) for all symbols $a_i \in \mathrm{Pos}(E)$, $\sigma(c_i) \notin \mathcal{F}_0$ and all tuples $f \in \mathrm{Follow}(E', a_i)$ add $(q, a_i) \in \delta(q, \sigma(a_i), \varphi(f))$,
   c) for all symbols $a_i \in \mathrm{First}(E')$ add $(q, \varepsilon) \in \delta(q, \dashv, \bot\ a_i)$.
6. Resulting automaton $A = (Q, \Sigma \cup \{\dashv\}, G, \delta, q, F)$ accepts the input word by final state.

Theorem 4.2 (rhPDA)**.** Algorithm 4.2 creates real-time height-deterministic pushdown automaton.

Proposition 4.3 (PDA's transition function size)**.** Automaton created by algorithm 4.2 has transition function exponential in size.

For proofs of theorems 4.1 and 4.2 and proposition 4.3 see section 4.3.

4.3 Formal Analysis

In this section the algorithm is discussed formally. Language equivalence issues will be addressed and after it is proven that the created pushdown automaton is real-time height-deterministic. Finally, time and space complexities of the presented algorithm are shown.

4.3.1 Language Equivalence

Proposition 4.1 (First function)**.** Function First(E') from equation (4.8) correctly returns all occurrences of symbols that can be root of any tree described by a regular tree expression E.

Proof. This will be proven by induction on the structure of regular tree expression (see definition 2.5).

Case $E = a(E_1, E_2, \ldots, E_n)$: This expression is one of the trivial cases. Only a can be root of a tree represented by E. Note the possibility of $a = \square$.

Case $E = \emptyset$: This is second trivial case as there is no tree represented by this expression.

For next cases, let us assume that the proposition is true for expressions E_1 and E_2.

Case $E = E_1 + E_2$: From the definition of regular tree expressions, this expression represents union of roots of trees described by E_1 and E_2.

Case $E = E_1^{*,\square}$: From the definition of an iteration element of a regular tree expression it can be seen that root of such tree can be only root of the iterated element or the substitution symbol. Therefore only $\square$ or elements of First(E_1) can be roots.

Case $E = E_1 \cdot \square E_2$: This case is the only tricky one and there are two possibilities. In the first case, suppose that $\square$ is not an element of First(E_1), then only First(E_1) is returned as no substitution can alter the root of E_1. In the second case, suppose that $\square$ is an element of First(E_1) then the symbol $\square$ gets substituted by trees from E_2. In other words the $\square$ is removed from set First(E_1) they are returned all together with roots of E_2. $\square$

Proposition 4.2 (Follow function)**.** Function Follow(E', a) correctly returns a set of tuples representing all possible tuples of direct children of a subscripted node a.

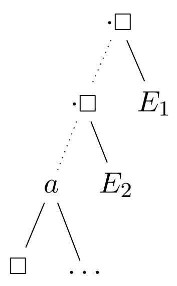
Proof. Consider all the possibilities that can happen while computing Follow function for symbol a:

Case 1 (no substitution symbol):

$$\begin{tikzcd}
 & a \arrow[dl, no head] \arrow[dr, no head] & \\
b_1 & \ldots & b_n
\end{tikzcd}$$

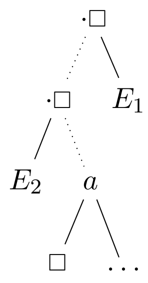
This case is trivial as no substitution is required, the algorithm correctly computes $(b_1, \ldots, b_n)$. No substitution happens.

Case 2a (inside substitution node):



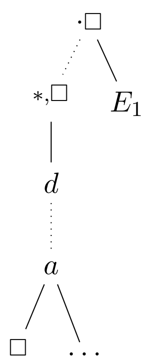
In this case a symbol is followed by $(\square, \ldots)$ tuple. It is easy to observe that $\square$ gets substituted for every symbol from First(E_2).

Case 2b (inside substitution node):



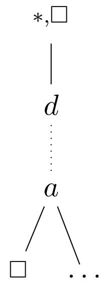
In this case a symbol is again followed by $(\square, \ldots)$ tuple. However, the $\square$ symbol here will get substituted for First(E_1). This follows directly from definition of regular tree expressions.

Case 3a (inside iteration node):



Symbol a symbol is again followed by $(\square, \ldots)$ tuple. From the definition of regular tree expression iteration, the $\square$ symbol can get substituted for d (i.e., First of the iteration element) or by $\square$. If substitution for $\square$ happens it gets again substituted for symbols from First(E_1).

Case 3b (inside iteration node):



This case is almost same as the previous one. The only difference is that the topmost element of substitution not present. Then this $\square$ element will remain and whole expression gets invalid.

There are few more trivial cases following directly from definitions. Other complex cases are not possible. $\square$

Theorem 4.1 (Language equivalence)**.** Algorithm 4.2 creates pushdown automaton A such that $L(A) = post(L(E))$, where E is given regular tree expression.

Proof. The proof consists of two parts: $post(L(E)) \subseteq L(A)$ and $L(A) \subseteq post(L(E))$.

Case $post(L(E)) \subseteq L(A)$: Or in other words it is proved that the pushdown automaton accepts at least the same set of words as the regular tree expression describes, i.e., all the linearised words from $post(L(E))$ are in $L(A)$ as well. Proof of this case comes directly from the proof of the Follow and First functions. The functions analysed all possible combinations of parent-children relations and these were converted to transition function for postfix notation. When the input string except $\dashv$ symbol is processed, it continues only if $\perp f$ is on the top of the pushdown store where f is an element of First set. Why this principle of postfix notation processing works was presented in the beginning of this chapter.

Case $L(A) \subseteq post(L(E))$: Or in other words it is proven that every word from $L(A)$ is in $post(L(E))$ too, i.e., there is no word from $L(A)$ that is not in $post(L(E))$. If there is a word from $L(A)$ that is not in $post(L(E))$ then either the computation of First or Follow functions were wrong or the transitions created from Follow sets would allow the automaton to accept something more. The functions First and Follow are proven to be correct. Third possibility is not possible from the definition of postfix notation and Follow function. The automaton tracks read subtrees on the pushdown store encoded by their representants, i.e., by their roots. Representant of a root is only allowed to be inserted if representants of its children are removed.

Both cases holds hence $L(A) = post(L(E))$. □

4.3.2 Real-Time Height-Deterministic Pushdown Automaton

Theorem 4.2 (rhPDA)**.** Algorithm 4.2 creates real-time height-deterministic pushdown automaton.

Proof. Recall definition of height-deterministic and real-time pushdown automata from definition 1.33.

Transition rules of the automaton are always in the form that when an input symbol a is read, $arity(a)$ symbols are removed from the pushdown store and one symbol is pushed on the top of the pushdown store. The only exception is for symbol $\dashv$ which pops two symbols and pushes none. This guarantees that the pushdown store height is always predetermined for all nondeterministic paths on an input word. Therefore the automaton fulfils the conditions height-determinism.

In real-time pushdown automata, all outgoing transitions are allowed to be only ε-transitions or only symbol transitions. Automaton created by algorithm 4.2 has no ε-transitions, hence is also real-time[NS07]. □

4.3.3 Time and Space Complexity

Proposition 4.3 (PDA's transition function size)**.** Automaton created by algorithm 4.2 has transition function exponential in size.

Proof. Exponential size of the transition mapping comes from the computation of Follow. Consider a symbol a whose direct descendants are only $\square$ symbols, i.e., $a(\square, \square, \ldots, \square)$ and $\square$ is substitutable for some elements from the set $S_\square$. Then, the set of follow tuples of a is equal to the set $S_\square \times S_\square \times \ldots \times S_\square$ which has $|S_\square|^{arity(a)}$ elements. From algorithm 4.2 it is obvious that for every element of the set there is one transition in the resulting automaton. $\square$

The upper bound for the number of transitions created from the Follow elements will be $\sum_{a_i \in \mathrm{Pos}(E)} m(a_i)^{arity(a_i)-s(a_i)} \leq |E||E|^{max(arity(\mathcal{F}))}$ where $|E|$ is the number of symbol nodes in E, $m(a_i)$ is size of substitution map for symbol a_i and $s(a_i)$ is the number of children of a_i that are from alphabet $\mathcal{F}$. This size of the transition function for a single symbol is really the worst case unexpected in practice. It is still unacceptable though. This thesis serves only as a proof of a concept that such a conversion method based on adaptation of Glushkov's algorithm is possible. Another method with better complexity based for computation of follow will have to be invented.

Several experiments with the size of the transition function after the automaton was determinised were made. It does not seem that determinisation would help lower the size of a transition mapping.

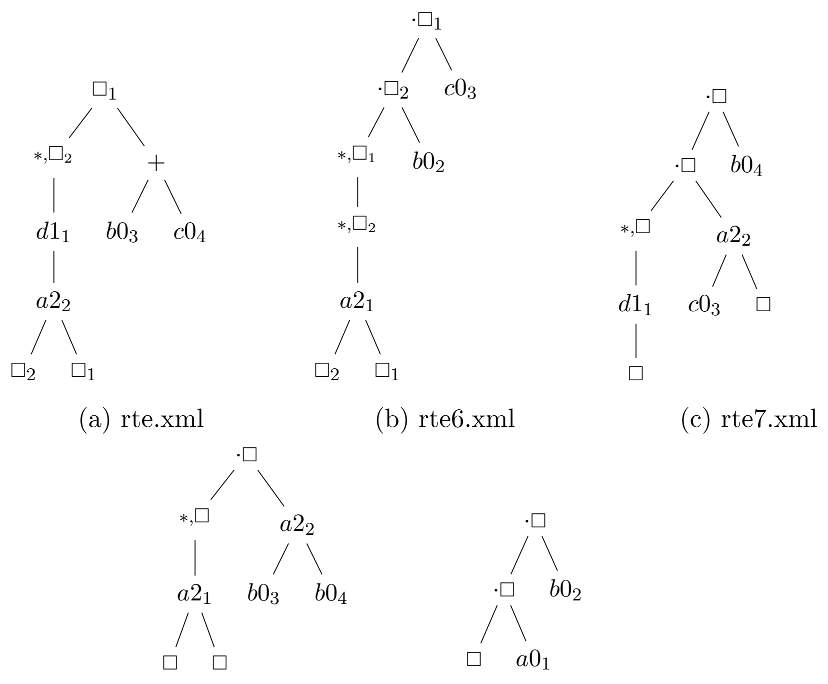
There are several possibilities how to address the complexity issue in the future. First idea stands on sharing one pushdown store symbol for several other ones. In function *ReplaceConstants* while computing Follow a Cartesian product is in fact being computed (which makes the complexity bad). It may be possible to avoid this by using a single pushdown store symbol corresponding to a set of elements from substitution map for $\square$ element.

Another way to solve this may be the use of $\square$ symbol on the stack. Consider an iteration over $\square$ where $\square$ gets substituted for some other expression F. When reading a subexpression F, the automaton may pop corresponding pushdown store symbols but push a $\square$ also to remember what substitutions are further possible. For an expression $E = (a2(\square, \square)) \cdot \square\, b0$ the interesting part of transition rules for pushdown automaton would look like this: $b0 \mid \varepsilon \to b0$, $b0 \mid \varepsilon \to \square$, $a2 \mid \square\square \to a2$, $a2 \mid \square\square \to \square$.

Both of these attempts to lower the asymptotic complexity will be a subject of investigation in the near future.

4.4 Examples

In the end of this complex chapter several regular tree expressions along with corresponding pushdown automata are shown. All expressions have already indexed occurrences of symbols from $\mathcal{F}$ as the algorithm dictates.



(a) rte.xml (b) rte6.xml (c) rte7.xml

(d) rte4.xml (e) rte5.xml

Figure 4.6: Regular tree expressions. Every expression is labelled with name of corresponding XML file from the folder with examples in implementation.

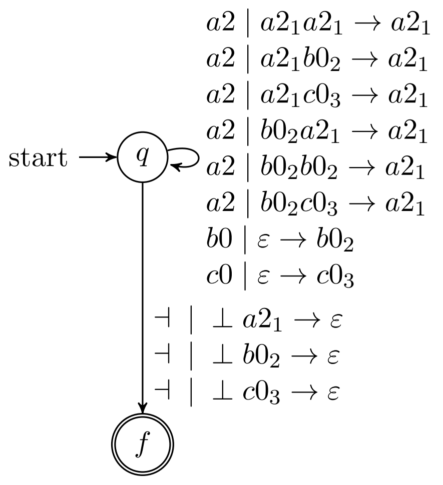
Example 4.7. Regular tree expression E' from figure 4.6a is invalid. Symbol $\square_2$ never gets substituted. Hence no pushdown automaton is created. ■

Example 4.8. Regular tree expression E' from figure 4.6b has equivalent pushdown automaton in figure 4.7a. The expression has following properties:

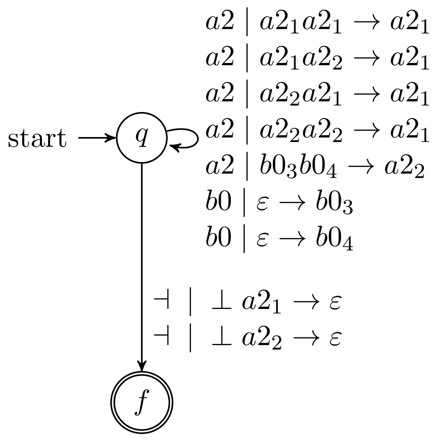
$$\begin{aligned}
\text{First}(E') &= \{a2_1,\ b0_2,\ c0_3\}\\
\text{Follow}(E', a2_1) &= \{(a2_1, a2_1), (a2_1, b0_2), (a2_1, c0_3), (b0_2, a2_1),\\
&\quad (b0_2, b0_2), (b0_2, c0_3)\}\\
\text{Follow}(E', b0_2) &= \text{Follow}(E', c0_3) = \emptyset
\end{aligned}$$ ■

Example 4.9. Regular tree expression E' from figure 4.6c has equivalent pushdown automaton in figure 4.7d. The expression has following properties:

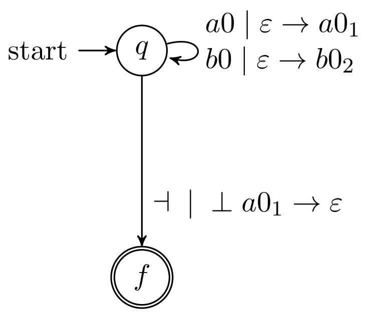
$$\begin{aligned}
\text{First}(E') &= \{d1_1,\ a2_2\}\\
\text{Follow}(E', d1_1) &= \{(d1_1), (a2_2)\}\\
\text{Follow}(E', a2_2) &= \{(c0_3, b0_4)\}\\
\text{Follow}(E', c0_3) &= \text{Follow}(E', b0_4) = \emptyset
\end{aligned}$$ ■


$a2 \mid a2_1a2_1 \to a2_1$
$a2 \mid a2_1b0_2 \to a2_1$
$a2 \mid a2_1c0_3 \to a2_1$
$a2 \mid b0_2a2_1 \to a2_1$
$a2 \mid b0_2b0_2 \to a2_1$
$a2 \mid b0_2c0_3 \to a2_1$
$b0 \mid \varepsilon \to b0_2$
$c0 \mid \varepsilon \to c0_3$
start
q
$\dashv \mid \perp a2_1 \to \varepsilon$
$\dashv \mid \perp b0_2 \to \varepsilon$
$\dashv \mid \perp c0_3 \to \varepsilon$
f

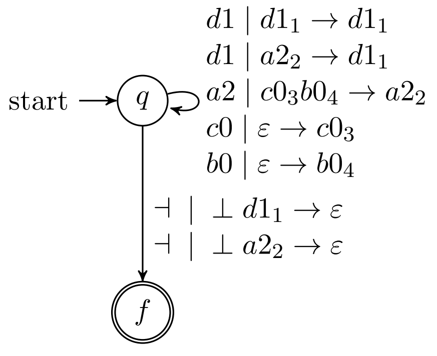
(a) rte6.xml


$a2 \mid a2_1a2_1 \to a2_1$
$a2 \mid a2_1a2_2 \to a2_1$
$a2 \mid a2_2a2_1 \to a2_1$
$a2 \mid a2_2a2_2 \to a2_1$
$a2 \mid b0_3b0_4 \to a2_2$
$b0 \mid \varepsilon \to b0_3$
$b0 \mid \varepsilon \to b0_4$
start
q
$\dashv \mid \perp a2_1 \to \varepsilon$
$\dashv \mid \perp a2_2 \to \varepsilon$
f

(b) rte4.xml


$a0 \mid \varepsilon \to a0_1$
$b0 \mid \varepsilon \to b0_2$
start
q
$\dashv \mid \perp a0_1 \to \varepsilon$
f

(c) rte5.xml


$d1 \mid d1_1 \to d1_1$
$d1 \mid a2_2 \to d1_1$
$a2 \mid c0_3b0_4 \to a2_2$
$c0 \mid \varepsilon \to c0_3$
$b0 \mid \varepsilon \to b0_4$
start
q
$\dashv \mid \perp d1_1 \to \varepsilon$
$\dashv \mid \perp a2_2 \to \varepsilon$
f

(d) rte7.xml

Figure 4.7: Pushdown automata equivalent to those from figure 4.6.

Example 4.10. Regular tree expression E' from figure 4.6d has equivalent pushdown automaton in figure 4.7b. The expression has following properties:

$$\begin{aligned}\text{First}(E') =& \{a2_1,\ a2_2\}\\ \text{Follow}(E', a2_1) =& \{(a2_1, a2_1), (a2_1, a2_2), (a2_2, a2_1), (a2_2, a2_2), \}\\ \text{Follow}(E', a2_2) =& \{b0_3, b0_4\}\\ \text{Follow}(E', b0_3) =& \text{Follow}(E', b0_4) = \emptyset\end{aligned}$$ ■

Example 4.11. Regular tree expression E' from figure 4.6e has equivalent pushdown automaton in figure 4.7c. The expression has following properties:

$$\begin{aligned}\text{First}(E') =& \{a0_1\}\\ \text{Follow}(E', a0_1) =& \text{Follow}(E', b0_2) = \emptyset\end{aligned}$$ ■

Chapter 5

Implementation

This chapter presents ALIB. Then it describes the process of implementing the algorithm described in the previous chapter into this library.

5.1 Automata Library

For the implementation, ALIB framework have been chosen. ALIB is a set of libraries and executables written especially for working with data structures in the area of formal languages, automata and graphs[1]. The library is written in C++ programming language (using C++11 standard).

ALIB started as a idea of Jan Trávníček[2] in 2013 and was introduced in bachelor's thesis of Martin Žák [Žák14]. Then, it could only manipulate with various types of grammars, automata and regular expressions. More features were implemented since then. In 2014, algorithms for converting and manipulating with regular languages [Pec14] and pushdown automata determinisation [Ves14] were added. In 2015, ALIB gained the ability to work with Arbology data structures (tree automata) [Pla15] and graphs [Ros15]. As of April 2016, ALIB is gaining the ability to benchmark algorithms and algorithms for LR parsers construction are being implemented.

These were only major changes. Of course, there were also many algorithms implemented when necessary. Now, ALIB contains decent collection of formal languages algorithms and also many stringology and Arbology algorithms (string or tree searching and indexing, etc.). However, many common stringology algorithms (e.g. suffix array and many others) are still missing[3].

Original goal was to provide sample implementations of algorithms in formal languages taught in various courses at Czech Technical University at

[1]This is valid as of April 2016. We always expect more features coming.

[2]Leader, or to be more accurate, *benevolent dictator for life* (see wikipedia page of this term) and main developer of the project.

[3]Hopefully, ALIB will be released under GPL license soon. Then, you are welcome to send your patches and implementations of algorithms.

Prague, Faculty of Information Technology. The algorithms are being written with focus on readability so they can be used for teaching purposes. Of course, the efficiency is not brilliant but asymptotic complexity of implemented algorithms is preserved. Nowadays, there is also possibility of writing efficient variations of certain algorithms.

The library also serves for fast implementation of new algorithms because you usually do not need to implement logic for manipulating with data structures. ALIB usually already handles this. This is one of the reasons ALIB was chosen although the data structures for regular tree expressions were not present and had to be implemented.

The ALIB project is divided into several libraries and executables. The executables usually serve as an interface to algorithms. Input and output is mostly done via Extensible Markup Language (XML) files, however there also exists possibility of creating input in some other, specialized or more human-friendly, formats for various data structures. For example, strings, regular expressions and finite automata can be written in human-friendly way.

ALIB follows the pipes and filters design pattern and UNIX philosophy. The UNIX philosophy for creating executables is that one executable should do one thing and it should do it properly. Then output of such executable can be modified using pipes and filters design pattern. One can create input, *filter*[4] it with executable A and then process its output with executable B to *filter* it with executable B.

In ALIB, one executes complex operations (usually) using system pipes. Complex operation can be for instance conversion of regular expression into minimal deterministic finite automaton using Glushkov's algorithm with and output is given as a transition diagram in PDF format. In order to do that, one should do the following sequence of operations (note that the following is just a verbose description of the shell command from figure 5.1):

i) create regular expression from human-readable notation using `aconvert` executable,

ii) convert it with Glushkov's algorithm using `aconversion` executable,

iii) use `adeterminize` to create deterministic finite automaton,

iv) remove unnecessary and unreachable states with the help of `atrim`,

v) minimize the automaton using `aminimize` executable,

vi) run `aconvert` executable once again to get dot language[GN00] description of automaton's transition diagram and

vii) draw the diagram it with `dot`[5] executable.

[4]By *filtering* some executable modifying input in some way is understood.

[5]Please note that `dot` executable is not part of ALIB project. It is part of *graphviz* package, see [GN00].

```
$ ./aconvert --regexp_from_string | ./aconversion -t fa -a glushkov \
| ./adeterminize | ./atrim | ./aminimize \
| ./aconvert --automaton_to_dot | dot -Tx11
```

Figure 5.1: Sample command-line command for converting regular (string) expression to equivalent minimal deterministic finite automaton.

It is also planned to expose ALIB's API bindings for some scripting language like Python or Ruby. Speed up of filters is expected because XML input and output (de)serialization in every executable of a complex command is a considerable slowdown, especially for large data structures. Then, one should be able to write the sequence of operations without unnecessary serialization in every step. Of course, it is possible to write complex operations directly as C++ executable, it is however not so user-friendly.

5.1.1 Parts of Automata Library

As was already said, ALIB consists of several libraries and executables. The core libraries are the following:

`alib2data` *(Data Structures)* contains data structures and basic operations on them, e.g., automata, grammars or string and trees.

`alib2algo` *(Algorithms)* contains implementations of algorithms over data structures, e.g., automata determinisation or conversions between computation models.

`alib2elgo` *(Efficient Algorithms)* contains time efficient implementations of some algorithms. So far some finite automata transformations like algorithm for removing ε-transitions were added.

`alib2common` *(Common Code Bases)* contains several classes needed in other libraries. It contains sighandlers, wrappers for internal object representations, exceptions and so on..

`alib2raw` *(Tree and XML)* contains algorithms for conversion of any XML document into tree data structure from ALIB and vice versa.

`alib2std` *(STL Extensions)* has various extensions of C++ Standard Template Library [ISO12], e.g. functions for printing (nested) containers.

`alib2str` *(Data Structures Parse Library)* contains string serialisers and parsers for various data structures.

As you can see, the `alib2algo` and `alib2data` are the most important parts for developing new algorithms and data structures respectively. The data structure representing regular tree expression is implemented in `alib2data` library. The conversion algorithm becomes part of the `alib2algo` library.

5.2 Changes in Automata Library

ALIB already had support for regular (string) expressions thanks to [Žák14, Pec14]. This was helpful in implementation because regular tree expressions are quite similar. The project also has support for pushdown automata operations and creation as well as determinisation of some pushdown automata subclasses (including real-time height-deterministic pushdown automata) [Ves14, PTJM16].

5.2.1 Implementation of Regular Tree Expressions

Regular tree expressions are implemented very similarly to regular expressions in ALIB. They only differ in structure of nodes and alphabet.

Every regular tree expression operator is implemented as a class inheriting from abstract class `rte::FormalRTEElement`. Here, by the term formal in `rte::FormalRTEElement`, we mean representation that is directly following the definition, i.e., every alternation has only two elements, left and right. It can not have form $E = E_1 + E_2 + \ldots + E_n$ but rather $E = E_1 + (E_2 + \ldots + (E_{n-1} + E_n))$. This is because the implementation of regular (string) expression also so called *unbounded* representation (the first example of alternation in the previous sentence, i.e., alternation or concatenation nodes can have multiple children). It was decided to keep the naming consistent with implementation of regular (string) expressions. In the future, some form of unbounded regular tree expression may be presented.

Hence, following classes representing the regular tree expression were implemented in `alib2data/src/rte/formal` directory:

1. `rte::FormalRTEElement` representing abstract element of RTE tree,
2. `rte::FormalRTEAlternation` representing union node, i.e., $E_1 + E_2$,
3. `rte::FormalRTEEmpty` representing empty expression, i.e., $\emptyset$.
4. `rte::FormalRTEIteration` representing iteration over $\square \in K$, i.e., $E^{*,\square}$,
5. `rte::FormalRTESymbol` representing symbol, which can has some children, i.e., $f(E_1, E_2, \ldots,)$.
6. `rte::FormalRTESubstitution` representing substitution over $\square$ element, i.e., $E_1 \cdot \square E_2$.

Of course, there is also wrapper class `rte::FormalRTE` holding the pointer to the root `rte::FormalRTEElement`. Both alphabets ($\mathcal{F}$ and $\mathcal{K}$) are encapsulated in single class.

5.2.1.1 XML Format of Regular Tree Expression

ALIB supports serialization and deserialisation of internal data structures into XML format which is the main communication format between executables. The XML format of regular tree expressions was chosen in such way that it would represent syntax tree of regular tree expression.

The name of opening tag for ALIB is `FormalRTE`. Firstly, as a regular tree expression is defined with two alphabets, $\mathcal{F}$ and $\mathcal{K}$, two alphabets are also in the XML file. So first child of `FormalRTE` node is node `alphabet` containing $\mathcal{F}$. Next child of `FormalRTE` node is `constantAlphabet` which handles $\mathcal{K}$. Children of `alphabet` and `constantAlphabet` nodes are `RankedSymbol` elements, which already existed in ALIB.

When both alphabets are specified, an expression starts (on the same level as alphabets). A regular tree expression consists of several nodes: Alternation, substitution, iteration and symbol node.

Alternation element is designed as a `alternation` XML node. It has two children – left and right operands of alternation, which are also regular tree expressions.

Substitution elements has two children from definition and also holds an information about a substitution node. Proposed `substitution` node for XML also has two children (left and right operands of the operation). However, ALIB has limited support for XML attributes which was the original idea how to store the substitution symbol. This was bypassed by storing the substitution symbols as first child of `substitution` node. Left and right operands of substitution operation were therefore moved to second and third child positions.

The iteration node (`iteration`) is similar. It is supposed to have one child but we also need the information about the substitution element. Same solution as in substitution node was used.

A symbol node holds a ranked symbol and it has amount of children which is equal to the arity of the ranked symbol. The `symbol` node is opened and `RankedSymbol` element is inside. If symbol has children then for every child another `symbol` is recursively present inside.

Exhaustive example can be found in appendix A. Methods for serialization and deserialisation of regular tree expressions are implemented in classes `rte::RTEToXMLComposer` and `rte::RTEFromXMLParser` respectively.

5.2.2 Implementation of Conversion Method

The conversion method itself consists of regular tree expression analysis (via Pos, First and Follow functions) and the construction of a pushdown automaton using these functions. In the following subsections both parts are focused on.

5.2.2.1 Algorithms for Regular Tree Expression Analysis

Functions Pos, Follow and First are implemented as recursive functions in class `rte::GlushkovTraversal`.

The function Pos traverses a syntax tree of regular tree expression and assigns to every symbol from $\mathcal{F}$ an index. It implements preorder traversal and assigns numbers from increasing sequence $1 \dots n$. Every labelled symbol is then inserted into the return set as an instance of `rte::GlushkovSymbol` class holding a label and a pointer to a tree node (`rte::FormalRTEElement`).

First function is implemented as a recursive function exactly according to the equation (4.8). It returns set of `rte::GlushkovSymbol`.

Function Follow has no specialities in implementation too. It follows the algorithm 4.1 and returns a set of tuples of `rte::GlushkovSymbol`. Functions `ReplaceConstants` and `preprocessSubMap` in `rte::GlushkovTraversal` are added for preprocessing the substitution map (case when symbol from constant alphabet maps to another constant symbol or self) and for computation of all possible follow tuples respectively.

Note that $\square \in \mathcal{K}$ symbol might appear in a follow tuple of some symbol. As was explained in previous chapter, this is invalid regular tree expression because elements of $\mathcal{K}$ are not part of an input alphabet. In this case exception (instance of `exception::CommonException` class) is thrown.

5.2.2.2 Algorithm for Creating Pushdown Automaton

The implementation of the method itself is simple when we have the functions for computing the functions from previous section implemented. Nondeterministic pushdown automaton (this data structure was already present in ALIB) is initialized and it is assigned a set of states and alphabet as stated in algorithm 4.2. As stated before, only automata which do not have $\square$ symbols in input alphabet are created.

This method yields nondeterministic pushdown automaton. However, the automaton fulfils the requirements to be real-time height-deterministic and can be determinised using the algorithm for real-time height-deterministic automata[PTJM16]. This algorithm was already implemented in ALIB and can be invoked using `adeterminize` executable. The method is implemented in `alib2algo` library inside `rte::convert::ToPushdownAutomatonGlushkov` class. Call to this method was also registered in already existing `aconversion` executable as algorithm called `glushkovrte`. Using `aconversion` one can then run the algorithm from command line. Such complex conversion command might then look like this: `$ ./aconversion -t pda -a glushkovrte -i rte.xml | ./adeterminize`.

5.3 Testing

Every software needs to be tested and this enhancement of ALIB is not different. As this work presents new algorithm, no reference implementation for output comparison was available. Therefore a set of tests was designed to cover as much cases as possible. Also unit tests for regular tree expressions were implemented.

For the testing, it is helpful to use the `arun` executable from ALIB. This executable can simulate run of deterministic finite and deterministic pushdown automata. This can be utilized in checking whether an input string is really accepted by the created pushdown automaton.

Single test consists of given regular tree expression that is converted to equivalent deterministic pushdown automaton. Then we analyse the regular expression by hand and create a script that generates the same language in its postfix linear form. Words generated by such script (which is in fact an implementation of context-free grammar) are than read by the pushdown automaton to check whether they are really accepted by the language. All words generated by the script must be accepted. The script is implemented in the Python language. Negative test cases were also added to check whether the automaton does not accept some word it should not.

The test cases are located in the `examples2/rte/tests` directory located in ALIB source tree. For every test the corresponding script for tree generation is attached.

5.3.1 Results

The test cases revealed several imperfections in the implementation. The most severe bug was however not in the implementation but in the algorithm computing the Follow function.

In the Follow function, while processing a substitution node, it was forgotten to forward the not altered substitution map to right subexpression. This was fortunately discovered in the early stages of this work.

Several other mistakes were also found by the set of test cases, mainly in implementation of regular tree expressions. However, most of them were not crucial.

Conclusion

In this chapter this thesis is summarised, the results and contributions are concluded. Some topics for future work and improving current state of presented algorithm are presented.

Evaluation of Goals

Several goals were stated in the assignment of this thesis. Firstly, regular tree expressions and the construction of Glushkov's automaton were to be studied. This was fulfilled in chapters 2 and 3.

Next goal was to propose a method constructing a pushdown automaton accepting a language given by a regular tree expression. This is main contribution of the thesis and chapter 4 is dedicated to this goal. Summary of this goal is given in next subsection.

The method was also implemented using C++ language and the implementation is part of the Automata Library, a framework for implementing algorithms in formal languages. Details can be found in chapter 5.

Contribution of the Thesis

The main goal was to propose an pushdown automaton equivalent to a regular tree expression. This goal was fulfilled in chapter 4.

Proposed method for regular tree expression matching follows Arbology principles – a tree is processed in its linear form using (deterministic) pushdown automaton. This automaton is created using principles introduced by Glushkov's algorithm for converting regular (string) expressions to finite automata. Functions from Glushkov's algorithm analysing a regular expression were modified for regular tree expressions. Using these functions an algorithm for creating pushdown automaton accepting linearised postfix notation of trees was presented. The resulting pushdown automaton is also real-time height-deterministic which means that it can be always determinised[NS07, PTJM16] and simulated.

Future Work

In the time of writing the thesis, interesting problems were opened. Some of them are interesting to deal with in the future. Firstly, it would be interesting to see if this method (with slight modifications) can work also with other linear notations, mainly bar notations. It should be straight forward to modify the algorithm for prefix ranked notation.

The issue with exponential number of transitions in the resulting pushdown automaton should also be dealt with. This is caused by exponential number of follow tuples for some symbol because of the Cartesian product in the computation. The computation of Follow function is also asymptotically slow then.

Regular tree expressions also really do not need to be defined over two disjoint alphabets $\mathcal{F}$ and $\mathcal{K}$. Only one alphabet allowing substitution operation at every constant symbol is also sufficient. The presented method could be adapted for this notation. Then the problem with invalid regular tree expressions would disappear.

The question whether homogeneity could be defined also for pushdown automata was opened.

Arbology group had presented another algorithm for conversion of regular tree expressions to pushdown automata. It was an adaptation of Thompson's conversion method[PJM11]. The algorithm presented in this topic is based on Glushkov's method. It would be interesting to see if also Brzozowski's method of regular expression derivatives could be adapted for trees.

Bibliography

[AM04] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

[arb] Arbology www pages. `http://arbology.fit.cvut.cz`. Accessed: 2016-04-04.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[CDG^{+}07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

[CZ00] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1–2):75 – 90, 2000.

[Glu61] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[GPW98] Dora Giammarresi, Jean-Luc Ponty, and Derick Wood. A reexamination of the Glushkov and Thompson constructions. 1998.

[Hol15] Jan Holub. *BIE-AAG – Automata and Grammars (slides of university lectures)*, 2015. Prague, Czech Technical University in Prague, Faculty of Information Technology, 2015.

[ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[JM09] Jan Janoušek and Bořivoj Melichar. On regular tree languages and deterministic pushdown automata. *Acta Inf.*, 46(7):533–547, 2009.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[KM08] Dietrich Kuske and Ingmar Meinecke. Construction of tree automata from regular expressions. In *Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan, September 16-19, 2008. Proceedings*, pages 491–503, 2008.

[LSZ13] Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi. From regular tree expression to position tree automaton. In *Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*, pages 395–406, 2013.

[Mel03] Bořivoj Melichar. *Jazyky a překlady.* Vydavatelství ČVUT, Praha, vyd. 2. přeprac. edition, 2003.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[NS07] Dirk Nowotka and Jiří Srba. *Mathematical Foundations of Computer Science 2007: 32nd International Symposium, MFCS 2007 Český Krumlov, Czech Republic, August 26-31, 2007 Proceedings*, chapter Height-Deterministic Pushdown Automata, pages 125–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Pec14] Tomáš Pecka. Automatová knihovna – převody mezi regulárními výrazy, regulárními gramatikami a konečnými automaty, Bachelor's thesis, Czech Technical University in Prague, 2014.

[PJM11] Radomír Polách, Jan Janoušek, and Bořivoj Melichar. From regular tree expression to position tree automaton. In *Proceedings of the 7th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 70–77, 2011.

[Pla15] Štěpán Plachý. Automatová knihovna – stromové automaty a algoritmy nad stromy, Bachelor's thesis, Czech Technical University in Prague, 2015.

[Pol11] Radomír Polách. Tree pattern matching and tree expressions. Master's thesis, Czech Technical University in Prague, 2011.

[PTJM16] Radomír Polách, Jan Trávníček, Jan Janoušek, and Bořivoj Melichar. On determinisation of visibly and real-time height-deterministic pushdown automata. *Submitted to Computer Languages, Systems & Structures*, 2016.

[Ros15] David Rosca. Automatová knihovna – isomorfismus planárních grafů, Bachelor's thesis, Czech Technical University in Prague, 2015.

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[Ves14] Jan Veselý. Automatová knihovna – determinizace konečných a zásobníkových automatů, Bachelor's thesis, Czech Technical University in Prague, 2014.

[Žák14] Martin Žák. Automatová knihovna – vnitřní a komunikační formát, Bachelor's thesis, Czech Technical University in Prague, 2014.

Appendix A

RTE XML Format

In this appendix example of XML representation is given. Regular tree expression containing all possible node types is presented with corresponding XML representation. The XML representation is described in section 5.2.1.1.

ALIB has no support for Unicode characters. In following XML example, $\square$ symbol is represented by character `z`.

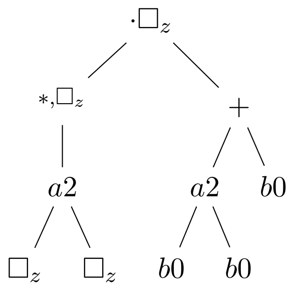


Figure A.1: Regular tree expression $E = a2(\square_z, \square_z)^{*,\square_z} \cdot \square_z (a2(b0, b0) + b0)$ presented in its syntax tree form. E is defined over alphabets $\mathcal{F} = \{a2, b0\}$ and $\mathcal{K} = \{\square_z\}$.

Listing A.1: XML representation of regular tree expression from figure A.1 with character z representing $\square_z$ symbol.

```
<FormalRTE>
 <alphabet>
  <RankedSymbol>
   <LabeledSymbol>
    <PrimitiveLabel><Character>a</Character></PrimitiveLabel>
   </LabeledSymbol>
   <Unsigned>2</Unsigned>
  </RankedSymbol>
  <RankedSymbol>
   <LabeledSymbol>
    <PrimitiveLabel><Character>b</Character></PrimitiveLabel>
  </LabeledSymbol>
  <Unsigned>0</Unsigned>
 </RankedSymbol>
</alphabet>
<constantAlphabet>
 <RankedSymbol>
  <LabeledSymbol>
   <PrimitiveLabel><Character>z</Character></PrimitiveLabel>
  </LabeledSymbol>
  <Unsigned>0</Unsigned>
 </RankedSymbol>
</constantAlphabet>

<substitution>
 <RankedSymbol>
  <LabeledSymbol>
   <PrimitiveLabel><Character>z</Character></PrimitiveLabel>
  </LabeledSymbol>
  <Unsigned>0</Unsigned>
 </RankedSymbol>
 <iteration>
  <RankedSymbol>
   <LabeledSymbol>
    <PrimitiveLabel><Character>z</Character></PrimitiveLabel>
   </LabeledSymbol>
   <Unsigned>0</Unsigned>
  </RankedSymbol>
  <symbol>
   <RankedSymbol>
    <LabeledSymbol>
     <PrimitiveLabel><Character>a</Character></PrimitiveLabel>
    </LabeledSymbol>
    <Unsigned>2</Unsigned>
   </RankedSymbol>
   <symbol>
    <RankedSymbol>
     <LabeledSymbol>
      <PrimitiveLabel><Character>z</Character></PrimitiveLabel>
     </LabeledSymbol>
     <Unsigned>0</Unsigned>
    </RankedSymbol>
   </symbol>
   <symbol>
    <RankedSymbol>
     <LabeledSymbol>
      <PrimitiveLabel><Character>z</Character></PrimitiveLabel>
     </LabeledSymbol>
     <Unsigned>0</Unsigned>
    </RankedSymbol>
   </symbol>
  </symbol>
 </iteration>
 <alternation>
  <symbol>
```

```
    <RankedSymbol>
     <LabeledSymbol>
      <PrimitiveLabel><Character>a</Character></PrimitiveLabel>
     </LabeledSymbol>
     <Unsigned>2</Unsigned>
    </RankedSymbol>
    <symbol>
     <RankedSymbol>
      <LabeledSymbol>
       <PrimitiveLabel><Character>b</Character></PrimitiveLabel>
      </LabeledSymbol>
      <Unsigned>0</Unsigned>
     </RankedSymbol>
    </symbol>
    <symbol>
     <RankedSymbol>
      <LabeledSymbol>
       <PrimitiveLabel><Character>b</Character></PrimitiveLabel>
      </LabeledSymbol>
      <Unsigned>0</Unsigned>
     </RankedSymbol>
    </symbol>
   </symbol>
   <symbol>
    <RankedSymbol>
     <LabeledSymbol>
      <PrimitiveLabel><Character>b</Character></PrimitiveLabel>
     </LabeledSymbol>
     <Unsigned>0</Unsigned>
    </RankedSymbol>
   </symbol>
 </substitution>
</FormalRTE>
```

Appendix B

Acronyms

ALIB	Automata Library
DFA	Deterministic finite automaton
NFA	Nondeterministic finite automaton
PDA	Pushdown automaton
XML	Extensible Markup Language

Appendix C

Contents of Enclosed CD

```
readme.txt ........................ the file with CD contents description
├── src.................... the directory with source codes of implementation
└── thesis................................ the directory containing the thesis
    ├── src .................................... the thesis LaTeX source codes
    └── Thesis_Pecka_Tomas_2016.pdf .............. the thesis in PDF format
```