FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Regular Expressions with Subpattern Recursion
Student:	Bc. Vojtěch Hruša
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

Study string (regular) expressions extended with subpattern recursion. Select an appropriate software project (either a tool or a library) and formally describe supported regular expression syntax and semantics (focusing on subpattern recursion). In addition, describe the expressive power of the expression and give proof that the statement is correct. Select the software project focusing on novelty (i.e., subpattern recursion in its expression should not be already described in the literature).

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 10 February 2021 in Prague.

Master's thesis

REGULAR EXPRESSIONS WITH SUBPATTERN RECURSION

Bc. Vojtěch Hruša

Faculty of Information Technology CTU in Prague
Department of Theoretical Computer Science
Supervisor: Ing. Ondřej Guth, Ph.D.
June 25, 2021

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Bc. Vojtěch Hruša. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Bc. Vojtěch Hruša. *Regular Expressions with Subpattern Recursion.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

I would like to thank my supervisor Ing. Ondřej Guth, Ph.D. for the help and advice with the thesis and also for his eternally positive attitude that made this thesis accomplishable. Furthermore, I would like to thank my friends and family for creating blue skies and golden sunshine all along the way.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on June 25, 2021 .....

Abstrakt

Tato práce se zabývá regulárními výrazy s rekurzivním zanořením. Jedná se o rozšíření regulárních výrazů možností opakovaného využití podvýrazů v jiných částech výrazu. Mimo jiné i v rámci samotného podvýrazu. Nejprve zkoumáme různá rozšíření regulárních výrazů, jejich vyjadřovací sílu a přistupy k jejich zkoumání. Poté představíme nově vymyšlené algoritmy pro převod mezi regulárními výrazy s rekurzivním zanořením a bezkontextovými gramatikami. Nakonec ukážeme, že množina jazyků generovaná regulárními výrazy s rekurzivním zanořením je právě množina všech bezkontextových jazyků.

Klíčová slova regulární výrazy, rekurzivní zanoření, regulární výrazy s rekurzivním zanořením, rozšíření regulárních výrzů, vyjadřovací síla

Abstract

This thesis studies regular expressions with subpattern recursion. It is an extension of regular expressions that allows reusing subpatterns in other parts of the expression, including inside the actual subpattern. Firstly, we study various extensions of regular expressions, their expressive power, and research techniques. We then introduce novel algorithms for transforming between regular expressions with subpattern recursion and context-free grammars. Finally, we show that regular expressions with subpattern recursion match exactly context-free languages.

Keywords regular expressions, subpattern recursion, regular expressions with subpattern recursion, extension of regular expressions, expressive power

List of Abbreviations

RE	Regular Expression
ERE	Extended Regular Expression
PCRE	Perl Compatible Regular Expressions
DFA	Deterministic Finite Automaton
NFA	Nondeterministic Finite Automaton
PDA	Pushdown Automaton
CFL	Context-free Language
CSL	Context-sensitive Language
CFG	Context-free Grammar
CSG	Context-sensitive Grammar

Chapter 1

Introduction

In this thesis, we study a specific extension of regular expressions. The formal language theory gives us a solid background in classifying languages, grammars, automata, and other constructs, including regular expressions. However, practical usage in fields like text searching or lexical analysis led to implementation of more convenient and user-friendly versions of regular expressions. Some of these extensions even increase the expressive power of regular expressions. There are many such extensions. For example, backreferences, lookahead assertions, or *subpattern recursion*, which we will focus on in the thesis.

Motivation

The motivation for using practical regular expressions, such as the PCRE library, is clear. In many cases, programmers stumble upon problems regarding matching some strings with a pattern. Instead of creating a complicated algorithm with many cases for different inputs, it may be possible to create a simple pattern that does most of the work automatically.

It may seem that regular expression is a concept advantageous exclusively to a knowledgable audience. However, a common user may benefit from it as well. Consider, for example, an avid reader of ebooks. An attractive, uncommon sentence once caught his attention. However, he does not remember what book the sentence was in. He might try searching all of his books with a full-text search, but only if he remembered the sentence precisely. Suppose he does not remember it; however, he remembers some parts of it. In this case, he might be able to find it by searching with an appropriate regular expression.

Another advantage of regular expressions is their availability. There are direct implementations or libraries in virtually all programming languages and environments. Additionally, regular expressions are available in various text editors or search engines.

Goals of the Thesis

This master's thesis aims to get familiar with regular expressions extended with subpattern recursion and similar extensions of regular expressions. We choose to work with the PCRE [4] library of regular expressions. From its broad collection of features, we choose only some basic ones along with the subpattern recursion. We then want to study the expressive power of chosen expressions or find major obstacles in the process.

Organization

The thesis is split into three main chapters. In the preliminaries chapter, we present basic definitions used throughout the thesis. We also show an observation and offer examples for some of the complicated concepts.

In the following chapter, which is focused on known results, we summarize some of the results regarding research of extended regular expressions, and we describe some of them in greater detail. Namely, we go through regular expressions with backreferences. There we describe pumping lemma and present other known results such as that they are context-sensitive and incomparable with context-free languages. Following with μ-regular expressions, we then go through conversions between μ-regular expressions, context-free grammars, and pushdown automata. In the final part, we introduce regular expressions with lookahead assertions and present some examples.

Finally, in the last chapter, we present our results. The chapter is divided into three parts. In the first part, we present an algorithm for transforming context-free grammars to equivalent regular expressions with subpattern recursion. A consequence of this method is applied in the second part, where we show that the Kleene star is redundant in a system of regular expressions with subpattern recursion. In the final part, we utilize a consequence of the redundancy of Kleene star to build an algorithm capable of transforming regular expressions with subpattern recursion to equivalent context-free grammars. After all of these preparations, we formulate the main result at the end of the chapter.

Chapter 2

Preliminaries

In this chapter, we provide some basic overview and definitions used throughout the thesis. Most of them are commonly used basic terms in the field of formal languages. We then also define the extended regular expressions, which we focus on in this thesis.

2.1 Formal Languages and Regular Expressions

We start the chapter with defining formal languages and regular expressions.

▶ **Definition 2.1** (Language)**.** Language L *over alphabet* Σ*:* $L \subseteq \Sigma^*$*. That is a set of strings containing symbols from* Σ*. Possible operations with Languages:*

- *set operations:* union, intersection, subtraction.
- concatenation*:* $L_1L_2 = \{xy : x \in L_1, y \in L2\}$ *(or* $L_1.L_2$ *— but we will omit the dot)*
- exponentiation*:* $L^n = L.L^{n-1}$ *and* $L^0 = \{\epsilon\}$
- iteration*:* $L^* = \bigcup_{n=0}^{\infty} L^n$ *(also called* Kleene star*)*

▶ **Definition 2.2** (Classical regular expression)**.** *Let us have an alphabet (a set of characters)* Σ*. Then:*

- *each* $x \in \Sigma$ *is a regular expression,*
- $\emptyset$ *and* ϵ *are regular expressions,*
- *for regular expressions* a, b*:*
  - $(a|b)$ *is regular expression*
  - (ab) *is regular expression*
  - $(a)^*$ *is regular expression*

▶ **Definition 2.3** (Value of classical regular expression)**.** *We denote the* value *or* Language matched *by some regular expression* a *by* $L(a)$*. And it is defined as follows:*

- *for each* $x \in \Sigma : L(x) = \{x\}$
- $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$
- *for regular expressions* a, b*:*
  - $L(a|b) = L(a) \cup L(b)$
  - $L(ab) = L(a)L(b)$
  - $L(a^*) = (L(a))^*$

Now we provide a new definition of *extended regular expressions*. That is the classical regular expressions extended by *some* features of regular expressions used by pcre.org.

We will use the following set of special characters $M =$ { `\`,`^`, `.`,`[`,`]`, `|`, `(`, `)`, `?`,`*`,`+`, `{`,`}` } and we call them *metacharacters*.

▶ **Definition 2.4** (Extended regular expression or ERE)**.** *Let* Σ *be an alphabet that does not include any metacharacters. But* Σ *may include* $\backslash m$ *for* $m \in M$*.*

(1) $\emptyset$, ϵ *and each* $x \in \Sigma$ *are regular expressions,*

(2) for regular expressions a, b*:*

- $(a|b)$ *is regular expression*
- (ab) *is regular expression*
- $(a)^*$ *is regular expression*

(3) `.` *is regular expression,*

(4) for regular expression a*:*

- $(a)+$ *is regular expression*
- $(a)?$ *is regular expression*

(5) for $i \in \mathbb{N}$ *and* $x_1, x_2, \ldots, x_i \in \Sigma$*:*

- $[x_1, x_2, \ldots, x_i]$ *is regular expression*
- $[\hat{}x_1, x_2, \ldots, x_i]$ *is regular expression*

(6) for $i < j$, $i, j \in \mathbb{N}$*, some ordering of* Σ *and* $x_i, x_j \in \Sigma$*:*

- $[x_i\text{-}x_j]$ *is regular expression*
- $[\hat{}x_i\text{-}x_j]$ *is regular expression*

(7) for $i \leq j$, $i, j \in \mathbb{N}_0$ *and regular expression* a*:*

- $(a)\{i\}$ *is regular expression*
- $(a)\{i,\}$ *is regular expression*
- $(a)\{i, j\}$ *is regular expression*

(8) for $i \in \mathbb{N}$*:*

- $(?i)$ *is regular expression.*
- $(?\text{-}i)$ *is regular expression.*

The definition requires each composed regular expression to be surrounded by a bracket pair. We will often omit the bracket pair, but only if such omission does not create ambiguity. Additionally, we assume that the Kleene star * has higher precedence than concatenation, which has higher precedence than |. For example, the expression $abcde$ is equal to $a(b(c(de)))$ and the expression $ab|cd$ is equal to $((ab)|(cd))$.

We denote $L(a)$ the value (or language) of regular expression a. Consider constructs as shown in Definition 2.4. The values of them are following:

▶ **Definition 2.5** (Value of extended regular expression)**.** *We start with the classical regular expressions (items (1) and (2)) and then extend them.*

(1) $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, *and for each* $x \in \Sigma : L(x) = \{x\}$,

(2) for regular expressions a, b*:*

- $L(a|b) = L(a) \cup L(b)$
- $L(ab) = L(a)L(b)$
- $L(a^*) = (L(a))^*$

(3) $L(.) = \bigcup_{x \in \Sigma} L(x)$,

(4) for regular expression a*:*

- $L(a+) = L(aa^*)$
- $L(a?) = L((a|\epsilon))$

(5) for $i \in \mathbb{N}$ *and* $x_1, x_2, \ldots, x_i \in \Sigma$*:*

- $L([x_1, x_2, \ldots, x_i]) = L(x_1|x_2|\ldots|x_i)$
- $L([\hat{}x_1, x_2, \ldots, x_i]) = L(y_1|y_2|\ldots|y_j)$, *where* $\{y_1, y_2, \ldots, y_j\} = \Sigma - \{x_1, x_2, \ldots, x_i\}$

(6) for $i < j,\ i, j \in \mathbb{N}$*, some ordering of* Σ *and* $x_i, x_j \in \Sigma$*:*

- $L([x_i\text{-}x_j]) = L(x_i|x_{i+1}|\ldots|x_j)$
- $L([\hat{}x_i\text{-}x_j]) = L(y_1|y_2|\ldots|y_k)$, *where* $\{y_1, y_2, \ldots, y_k\} = \Sigma - \{x_i, x_{i+1}, \ldots, x_j\}$

(7) for $i \leq j,\ i, j \in \mathbb{N}_0$ *and regular expression* a*:*

- $L(a\{i\}) = L(a)^i$
- $L(a\{i,\}) = L(a)^i L(a^*)$
- $L(a\{i, j\}) = \bigcup_{k=i}^{j} L(a)^k$

(8) Assign numbers $1, 2, 3, \ldots$ *from left to right to each opening bracket of a regular expression,* excluding *opening brackets followed by* `?`*. That is, the left-most opening bracket in an expression is assigned number* 1*, the second opening bracket is assigned number* 2*, and so on. See an example of this numbering in Example 2.1. Then for* $i \in \mathbb{N}$*:*

- *if there is an opening bracket with assigned number* i *to the left of* $(?i)$*, then let* a *be the regular expression enclosed by the opening bracket number* i *and the corresponding closing bracket. (That is* i*th defined bracket pair.) Then* $L((?i)) = L(a)$
- *if there is at least* i *opening brackets to the left of* $(\text{-}i)$ *and* b *is the number assigned to the closest opening bracket to the left of* $(\text{-}i)$*, then let* a *be the regular expression enclosed by the opening bracket number* $b - i + 1$ *and the corresponding closing bracket. (That is* i*th last defined pair.) Then* $L((?\text{-}i)) = L(a)$

See an example of usage in Example 2.2.

In the following examples, we use letters $a, b, c, \ldots$ as symbols from alphabet Σ.

▶ **Example 2.1** (Numbering of brackets)**.** Consider the following regular expression:

$$(a((b|c)d)(?2)(e|f)(?\text{-}2)(g))$$

The assignment of numbers to opening brackets according to item 8 in Definition 2.5 is:

$$({}_1a({}_2({}_3b|c)d)(?2)({}_4e|f)(?\text{-}2)({}_5g))$$

See that each opening bracket not followed by `?` is assigned an increasing number from left to right. For clarity, we show the corresponding closing brackets as well:

$$({}_1a({}_2({}_3b|c)_3d)_2(?2)({}_4e|f)_4(?\text{-}2)({}_5g)_5)_1$$

▶ **Example 2.2** (Usage of $(?i)$)**.** Consider this regular expression from the previous example:

$$(a((b|c)d)(?2)(e|f)(?\text{-}2)(g))$$

There is an opening bracket with assigned number 2 to the left of (?2). See the bracket pair marked within the expression:

$$(a({}_2(b|c)d)_2(?2)(e|f)(?\text{-}2)(g))$$

Therefore, $L((?2))$ is equal to $L((b|c)d)$ in this particular case.

Additionally, there are at least 2 opening brackets to the left of (?-2) and the closest opening bracket to the left of (?-2) is assigned number 4. From $b - i + 1$ we get $4 - 2 + 1 = 3$. See these bracket pairs marked within the expression:

$$(a(({}_3b|c)_3d)(?2)({}_4e|f)_4(?\text{-}2)(g))$$

Therefore, $L((?\text{-}2))$ is equal to $L(b|c)$ in this particular case.

Note that the definition restricts only the opening bracket to be to the left of the $(?i)$ expression, not the corresponding closing bracket. Therefore, putting this expression inside ith bracket pair allows us to use it recursively.

▶ **Example 2.3** (Recursive usage of $(?i)$)**.** Consider the following regular expression:

$$(\epsilon|(a(?1)b))$$

Bracket pair with assigned number 1 is in this case the whole expression $({}_1\epsilon|(a(?1)b))_1$, which means that

$$L((?1)) = L(\epsilon|(a(?1)b)).$$

Let us now examine the value of this expression and denote it by $\alpha = L(\epsilon|(a(?1)b))$.

$$\begin{aligned} \alpha &= L(\epsilon|(a(?1)b)) \\ &= L(\epsilon) \cup L((a(?1)b)) \\ &= \{\epsilon\} \cup L((a(?1)b)) \end{aligned}$$

$$\begin{aligned} L((a(?1)b)) &= L(a)L((?1))L(b) \\ &= \{a\}L((?1))\{b\} \end{aligned}$$

$$\begin{aligned} L((?1)) &= L(\epsilon|(a(?1)b)) \\ &= \alpha \end{aligned}$$

There we get $\alpha = \{\epsilon, a\alpha b\}$. If we substitute α inside the set, we get: $\alpha = \{\epsilon, ab, aa\alpha bb\}$. By repeating this substitution arbitrary number of times, we get: $\alpha = \{\epsilon, ab, aabb, aaabbb, \ldots\}$. That is a non-regular language $\{a^n b^n, n \in \mathbb{N}_0\}$.

▶ **Observation 2.1.** *Items (1) through (7) can be expressed using classical regular expressions only. We say that two regular expressions a, b are* equivalent $a \equiv b$ *if $L(a) = L(b)$.*

- *Items* (1) *and* (2) *in the extended definition describe the classical regular expressions.*
- *Items* (3) *through* (7) *can be rewritten by equivalent regular expressions:*

  (3) $. \equiv (x_1|x_2|\ldots|x_x)$ *for each* $x_i \in \Sigma$

  (4) for regular expression a:
  - $a+ \equiv aa^*$
  - $a? \equiv (a|\epsilon)$

  (5) for $i \in \mathbb{N}$ *and* $x_1, x_2, \ldots, x_i \in \Sigma$*:*
  - $[x_1, x_2, \ldots, x_i] \equiv (x_1|x_2|\ldots|x_i)$
  - $[\hat{}x_1, x_2, \ldots, x_i] \equiv (y_1|y_2|\ldots|y_j)$, *where* $\{y_1, y_2, \ldots, y_j\} = \Sigma - \{x_1, x_2, \ldots, x_i\}$

  (6) for $i < j,\ i, j \in \mathbb{N}$, *some ordering of* Σ *and* $x_i, x_j \in \Sigma$*:*
  - $[x_i\text{-}x_j] \equiv (x_i|x_{i+1}|\ldots|x_j)$
  - $[\hat{}x_i\text{-}x_j] \equiv (y_1|y_2|\ldots|y_k)$, *where* $\{y_1, y_2, \ldots, y_k\} = \Sigma - \{x_i, x_{i+1}, \ldots, x_j\}$

  (7) for $i \leq j,\ i, j \in \mathbb{N}_0$ *and regular expression a:*
  - $a\{i\}) \equiv aa\ldots a$ *(i-times a)*
  - $a\{i,\}) \equiv a\{i\}a^*$
  - $a\{i,j\}) \equiv (a\{i\}|a\{i+1\}|\ldots|a\{j\})$

As we can see, the extended regular expressions without item (8) can be expressed using only the tools of the classical regular expressions. Therefore, its expressive power is the same as of classical regular expressions and it can match the regular languages only. The potential addition to the expressive power is the item (8). And it indeed does add some expressive power as we show later in the thesis.

2.2 Grammars and Operations

Now we present common definitons of grammars, their values and some operations with them. Later on, we will focus on the relation of regular expressions and grammars.

▶ **Definition 2.6** (Grammar)**.** Grammar $G = (N, \Sigma, R, S)$, *where:*

- N *is finite set of non-terminal symbols*
- Σ *is finite set of terminal symbols (additionally,* N *and* Σ *are disjoint)*
- R *is finite set of rules (that is finite subset of* $(N \cup \Sigma)N(N \cup \Sigma)^* \times (N \cup \Sigma)^*$*)*
- $S \in N$ *is the starting symbol*

▶ **Definition 2.7** (Regular grammar)**.** *Consider grammar* $G = (N, \Sigma, R, S)$*. We call the grammar* G regular *if each rule in* R *has one of the following forms:*

1. $A \to aB$
2. $A \to a$
3. $S \to \epsilon$ *(if S is not present on the right-hand side of any other rule)*

for $A, B \in N$ and $a \in \Sigma$.

▶ **Definition 2.8** (Context-free grammar or CFG)**.** *Consider grammar $G = (N, \Sigma, R, S)$. We call the grammar G* context-free *if each rule in R has the following form:*

1. $A \to \alpha$

for $A \in N$ and $\alpha \in (N \cup \Sigma)^*$.

▶ **Definition 2.9** (Context-sensitive grammar or CSG)**.** *Consider grammar $G = (N, \Sigma, R, S)$. We call the grammar G* context-sensitive *if each rule in R has one of the following forms:*

1. $\gamma A \delta \to \gamma \alpha \delta$
2. $S \to \epsilon$ *(if S is not present on the right-hand side of any other rule)*

for $A \in N$, $\alpha \in (N \cup \Sigma)+$ and $\gamma, \delta \in (N \cup \Sigma)^*$.

▶ **Definition 2.10** (Unrestricted grammar)**.** *Consider grammar $G = (N, \Sigma, R, S)$. We call the grammar G* unrestricted *if each rule in R has the following form:*

1. $\gamma A \delta \to \alpha$

for $A \in N$, and $\alpha, \gamma, \delta \in (N \cup \Sigma)^*$.

▶ **Definition 2.11** (Derivation)**.** *Consider grammar $G = (N, \Sigma, R, S)$ and $\alpha, \beta, a, b \in (N \cup \Sigma)^*$. Then $\alpha a \beta$* derives *(or* yields*) $\alpha b \beta$ if $(a \to b) \in R$. We denote this by $a \Rightarrow b$*

▶ **Definition 2.12** (Reflexive transitive closure)**.** *We say that α* yields *β* after any number of steps *($\alpha \Rightarrow^* \beta$) if for some $i \in \mathbb{N}_0$ exists sequence $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_i$ where $\alpha_0 = \alpha$ and $\alpha_i = \beta$.*

▶ **Definition 2.13** (Language generated by grammar)**.** *Consider grammar $G = (N, \Sigma, R, S)$. Then* language generated by grammar *G is $L(G) = \{\alpha : \alpha \in \Sigma^*, \exists S \Rightarrow^* \alpha\}$. That is a set of all possible words using alphabet Σ that can be derived after any number of steps from the starting symbol S.*

Chapter 3

Known Results

The formal language theory and related concepts such as formal grammars or regular expressions are well-established terms in mathematics and computer science. The same cannot be said about commonly used, yet not formally established, extensions of regular expressions. We now show some results in researching regular expressions with extensions related to subpattern recursion. We start with one of the most commonly used extensions – backreferences. We show some results regarding their expressive power. We then move on to μ-regular expressions – a theoretical concept but less commonly used tool in practice. However, they received quite significant research attention. In the final section, we study lookahead assertions. This feature is available, for example, in PCRE, and it can be very useful tool for advanced pattern matching. Also, it turns out that lookahead assertions significantly increase the expressive power of regular expressions.

3.1 Regular Expressions with Backreferences

Campeanu et al. [1], in their formal study, show some properties of regular expressions with backreferences, prove a pumping lemma for them, and show that languages represented by them are incomparable with context-free languages.

Preliminaries

For the following part, we will extend classical regular expressions defined in Definition 2.2 with backreferences to semi-regular expressions. Note that usage of this term in the original article is different. We use it this way to avoid usage of ambiguous terms throughout the thesis.

▶ **Definition 3.1** (Semi-regular expression)**.** *Let α be a classical regular expression over alphabet*

$$\Sigma \cup \{\backslash m : m \in \mathbb{N}\}.$$

Consider each bracket pair in α numbered from left to right, according to the occurrence of the opening bracket of each pair.

If each occurrence of the backreference $\backslash m$ $(m \in \mathbb{N})$ *in* α *is preceded by the closing bracket of the* m*th bracket pair. Then* α *is* semi-regular expression.

▶ **Definition 3.2** (Set of occurrences of subexpressions)**.** $SUB(\alpha) :=$ *set of all occurrences of subexpressions in a semi-regular expression* α*.*

Intuitively, that is set of all substrings of α, which are also proper semi-regular expressions.

▶ **Definition 3.3** (Match)**.** *A* match *of a semi-regular expression* α *is a rooted, ordered tree* T_α*. Each vertex of* T_α *contains an ordered pair which is an element of* $\Sigma^* \times SUB(\alpha)$*.* T_α *is constructed using following rules:*

1. *Root of* $T_\alpha := (w, \alpha), w \in \Sigma^*$*.*
2. *For a vertex* u *of* T_α*: assume that* $u = (w, \beta)$*, where* $\beta = (\beta_1)(\beta_2) \in SUB(\alpha)$*. Then* u *has two successors* (w_1, β_1) *and* (w_2, β_2)*, where* $w = w_1 w_2$*.*
3. *For a vertex* u *of* T_α*: assume that* $u = (w, \beta)$*, where* $\beta = (\beta_1)|(\beta_2) \in SUB(\alpha)$*. Then* u *has one successor that is either* (w, β_1) *or* (w, β_2)*.*
4. *For a vertex* u *of* T_α*: assume that* $u = (w, \beta)$*, where* $\beta = (\beta_1)* \in SUB(\alpha)$*. Then there are two cases:*
   - *if* $w \neq \epsilon$*, then* u *has* $k \geq 1$ *successors* (w, β_1)*.*
   - *if* $w = \epsilon$*, then* u *has one successor* (ϵ, β) and it is a leaf *of* T_α*.*
5. *For a vertex* u *of* T_α*: assume that* $u = (w, a)$*, where* $a \in \Sigma$*. Then* $w = a$ *and* u *is a leaf of* T_α*.*
6. *For a vertex* u *of* T_α*: assume that* $u = (w, \backslash m)$*, where* $m \in \mathbb{N}$*. Then* u *is a leaf of* T_α*. Additionally, let* β_m *be the subexpression enclosed in* m*th bracket pair and let* u_{β_m} *be the previous vertex in* T_α *(considering standard left-to-right ordering of a tree) with* β_m *as the second component. Then* $w = w_m$*, where* w_m *is the first component of* u_{β_m}*. If there is no vertex containing* β_m *then* $w = \epsilon$*.*

Finally, value of semi-regular expressions can now be defined using the previous definition of matches.

▶ **Definition 3.4** (Value of semi-regular expressions)**.** *The* value *or* Language matched *by some semi-regular expression* α *is defined as:*

$$L(\alpha) = \{w \in \Sigma^* : (w, \alpha) \text{ is a root of some match } T_\alpha\}$$

Pumping lemma

Different versions of pumping lemmas are generally used to show that some languages are not regular. We now show how they approached pumping lemma for semi-regular expressions, which can then be used to show that some languages cannot be matched by semi-regular expressions.

▶ **Lemma 3.1** (Pumping lemma)**.** *Consider a semi-regular expression* α*. Then there is a constant* N *such that if* $w \in L(\alpha)$ *and* $|w| > N$*, then exists a decomposition* $w = x_0 y x_1 y x_2 \dots x_m$ *for some* $m \geq 1$*, while following rules hold:*

1. $|x_0y| \leq N$
2. $|y| \geq 1$ *(that is:* $y \neq \epsilon$*)*
3. $x_0y^jx_1y^jx_2 \dots x_m \in L(\alpha)$ *for all* $j > 0$

Proof. Choose $N = |\alpha| \cdot 2^t$, where t is the number of backreferences in α and let $w \in L(\alpha)$, where $|w| > N$. Matched word without usage of backreferences or a Kleene star cannot be longer than $|\alpha|$. Additionally, each backreference can double the length of a matched word at most (that is 2^t in the choice of N). There is, therefore, a part of the word w that matches a Kleene star with at least two iterations. Choose such left-most occurrence and denote the part of α it matches as e^*.

Let $w = x_0yz$, where y is the first iteration of e^* in w. Thus we have $|y| \geq 1$ (because y is not empty) and $|x_0y| \leq N$ (because y is the left-most match of a non-empty Kleene star and it contains only one iteration of it). Items 1 and 2 are therefore satisfied.

Then there are two cases:

- e^* *is not backreferenced*: We choose $z = x_1$, $w = x_0yx_1$ then satisfies the item 3 and therefore lemma holds for $m = 1$.
- e^* *is backreferenced*: We choose

  $$z = x_1yx_2yx_3 \dots x_m,$$

  where each backreference of e^* denoted as y in z. The whole word is then

  $$w = x_0yx_1yx_2yx_3 \dots x_m$$

  and

  $$x_0y^jx_1y^jx_2y^j \dots x_m \in L(\alpha)$$

  for all $j > 0$, because each y can be iterated arbitrarily, as it contains the Kleene star. Then w satisfies the item 3 and therefore lemma holds.

◀

Properties

We now show some theorems presented in the original article.

▶ **Theorem 3.1.** *Languages matched by semi-regular expressions are context-sensitive.*

Proof. To prove this theorem, it is sufficient to show that each language matched by some semi-regular expression α is accepted by a linear bounded automaton. If α does not include any backreference, then it is a classical regular expression, there is a finite automaton accepting words matched by α and so the language is context-sensitive.

If there are some backreferences present in α, then we need a buffer (stored on the tape) for each bracket pair in α to store the part of input string that matches the given subexpression. For

convenience, we assign the brackets in α numbers from 1 as they appear in α. Similarly as in Example 2.1.

Let us now construct a NFA (non-deterministic finite automaton) with (numbered) brackets and backreferences as input symbols. Then build a Turing machine following transitions of the constructed NFA:

- for $(_i$ transition: do not read input symbol, start storing matched input symbols into the ith buffer.
- for $)_i$ transition: do not read input symbol, stop storing matched input symbols into the ith buffer.
- for $\backslash i$ transition: compare input symbols with the ith buffer.

The number of buffers is constant (depending on the number of backreferences in α) and each buffer needs at most the space of the size of the input word. Therefore, the tape is bounded by a linear function and the constructed Turing machine is a linear bounded automaton. Thus, languages matched by semi-regular expressions are context-sensitive. ◀

▶ **Theorem 3.2.** *The family of languages matched by semi-regular expressions is incomparable with the family of context-free languages.*

Proof. Consider the following semi-regular expression

$$(aa^*)b\backslash 1b\backslash 1.$$

It matches the language

$$L' = \{a^n b a^n b a^n : n \geq 1\}$$

which is not context-free. Therefore, semi-regular expressions can match some languages that are not context-free.

On the other hand, consider the language

$$L = \{a^n b^n : n > 0\},$$

which is context-free. We now show, that it cannot be expressed by a semi-regular expression.

Assume that L can be expressed by a semi-regular expression α. Consider the word $w = a^N b^N$, where N is the constant of Lemma 3.1. Additionally, according to Lemma 3.1, w can be decomposed to

$$x_0 y x_1 y x_2 \dots x_m, m \geq 1.$$

Because $|y| \geq 1$ and $|x_0 y| \leq N$, we know that $y = a^i$ for some $i \geq 1$. Then

$$w' = x_0 y^2 x_1 y^2 \dots x_m$$

should also be part of the language L. But w' contains more symbols a than symbols b, and thus it is not part of the language L, and we get a contradiction.

There are semi-regular expressions which can match languages that are not context-free and also there are context-free languages that cannot be matched by any semi-regular expression. Therefore, context-free languages and languages matched by semi-regular expressions are incomparable. ◀

Overall, Campeanu et al. [1] treated regular expressions with backreferences formally for the first time. They introduced a pumping lemma, showed that languages matched by regular expressions with backreferences are proper subset of context-sensitive languages but they are incomparable with context-free languages. Additionally, they showed that regular expressions with backreferences are closed under union but not under complementation.

3.2 μ-regular Expressions

In this section, we focus on another extension of regular expressions – μ-regular expressions. The fact that μ-regular expressions match precisely context-free languages was considered a folklore theorem according to Leiß [3]. We will focus on researches on their expressive power in this section. We summarize some results of Thieman [6] regarding construction of pushdown automata from μ-regular expressions and then show some results of Gruppen [2] regarding construction of context-free grammar from μ-regular expressions and the other way around.

Preliminaries

▶ **Definition 3.5** (μ-regular pre-expression)**.** *The set of* μ-regular pre-expressions $\mathbf{R}(\Sigma, X)$ *over alphabet* Σ *and set of variables* X *is inductively defined as:*

- $0 \in \mathbf{R}(\Sigma, X)$
- $1 \in \mathbf{R}(\Sigma, X)$
- $a \in \Sigma \Rightarrow a \in \mathbf{R}(\Sigma, X)$
- $r, s \in \mathbf{R}(\Sigma, X) \Rightarrow rs \in \mathbf{R}(\Sigma, X)$
- $r, s \in \mathbf{R}(\Sigma, X) \Rightarrow r|s \in \mathbf{R}(\Sigma, X)$
- $r \in \mathbf{R}(\Sigma, X) \Rightarrow r^* \in \mathbf{R}(\Sigma, X)$
- $x \in X \Rightarrow x \in \mathbf{R}(\Sigma, X)$
- $r \in \mathbf{R}(\Sigma, X \cup \{x\}) \Rightarrow \mu x.r \in \mathbf{R}(\Sigma, X)$

▶ **Definition 3.6** (μ-regular expression)**.** *The set of* μ-regular expressions *over alphabet* Σ *is defined as* $\mathbf{R}(\Sigma) := \mathbf{R}(\Sigma, \emptyset)$.

▶ **Example 3.1** (μ-regular pre-expression)**.** For $a, b \in \Sigma$ and $x \in X$ the following expression is an example of μ-regular pre-expression:

$$axb + 1$$

In the previous example we showed a μ-regular pre-expression that is not μ-regular expression. In the following example, we show similar expression that satisfies the definition of μ-regular expression.

▶ **Example 3.2** (μ-regular expression)**.** For $a, b \in \Sigma$ the following expression is an example of μ-regular expression:

$$\mu x.axb + 1$$

We now show some functions that we will need in the next part.

- We use $P(S)$ to denote the power set of S.
- We use $\eta : X \to P(\Sigma^*)$ as a *variable environment*. Essentially, it assigns some language to a variable $x \in X$.
- We use $\eta[x \to L]$ as a variable environment with L as a fixed output for x. That is, η such that $\eta(x) = L$
- We use $\mathbf{L} : \mathbf{R}(\Sigma, X) \times (X \to P(\Sigma^*)) \to P(\Sigma^*)$ as the language matched by a μ-regular expression. Its full definition is then inductively described in Definition 3.7.
- We use $\text{lfp}L.\mathbf{L}(r, \eta)$ as the *least fixed point* operator on $P(\Sigma^*)$. This application gives us the smallest set $L \subseteq \Sigma^*$ such that $L = \mathbf{L}(r, \eta)$. (Note that Thieman [6] shows that such L always exists)

▶ **Definition 3.7** (Value of μ-regular pre-expressions)**.** Value *of (or* language matched *by) a μ-regular pre-expression is defined as:*

- $\mathbf{L}(0, \eta) = \emptyset$
- $\mathbf{L}(1, \eta) = \{\epsilon\}$
- $\mathbf{L}(a, \eta) = \{a\}$*, for* $a \in \Sigma$
- $\mathbf{L}(\alpha\beta, \eta) = \mathbf{L}(\alpha, \eta)\mathbf{L}(\beta, \eta)$*, where* α *and* β *are μ-regular pre-expressions*
- $\mathbf{L}(\alpha|\beta, \eta) = \mathbf{L}(\alpha, \eta) \cup \mathbf{L}(\beta, \eta)$*, where* α *and* β *are μ-regular pre-expressions*
- $\mathbf{L}(x, \eta) = \eta(x)$*, for* $x \in X$
- $\mathbf{L}(\mu x.r, \eta) = lfpL.\mathbf{L}(r, \eta[x \to L])$

▶ **Definition 3.8** (Value of μ-regular expressions)**.** *For an expression without free variables* $r \in \mathbf{R}(\Sigma)$*,* Value *of (or* language matched *by) r is defined as:* $\mathbf{L}(r) = \mathbf{L}(r, \emptyset)$.

We now take the expression we saw before and show what language does it match.

▶ **Example 3.3** (Value of μ-regular expression)**.**

$$\begin{aligned}
\mathbf{L}(\mu x.axb + 1, \eta) &= \text{lfp}L.\mathbf{L}(axb + 1, \eta[x \to L]) \\
&= \text{lfp}L.\mathbf{L}(axb, \eta[x \to L]) \cup \mathbf{L}(1, \eta[x \to L]) \\
&= \text{lfp}L.\mathbf{L}(a, \eta[x \to L])\mathbf{L}(x, \eta[x \to L])\mathbf{L}(b, \eta[x \to L]) \cup \{\epsilon\} \\
&= \text{lfp}L.\{a\}\eta[x \to L](x)\{b\} \cup \{\epsilon\} \\
&= \text{lfp}L.\{a\}L\{b\} \cup \{\epsilon\} \\
&= \{a^n b^n : n \in \mathbb{N}_0\}
\end{aligned}$$

Link between PDA and μ-regular expressions

The idea of partial derivatives of regular expressions was generalized for μ-regular expressions by Thieman [6] and then used to create a pushdown automaton from a μ-regular expression.

We first remind ourselves how derivation of classical regular expressions works in the following definition:

▶ **Definition 3.9** (derivation of classical regular expressions)**.** *Derivation* $\frac{d}{dx}$ *of a classical regular expression* E *by some string* $x \in \Sigma^*$ *is defined as:*

- $\frac{dE}{d\epsilon} = E$
- *for symbols of the alphabet* $a, b \in \Sigma$*:*
  - $\frac{d\emptyset}{da} = \{\}$
  - $\frac{d\epsilon}{da} = \{\}$
  - $\frac{db}{da}$:
    - $\frac{db}{da} = \epsilon$, *for* $a = b$
    - $\frac{db}{da} = \{\}$, *for* $a \neq b$
  - $\frac{d(E|F)}{da} = \frac{dE}{da} | \frac{dF}{da}$
  - $\frac{d(EF)}{da}$:
    - $\frac{d(EF)}{da} = \frac{dE}{da}F$, *for* $\epsilon \notin L(E)$
    - $\frac{d(EF)}{da} = \frac{dE}{da}F | \frac{dF}{da}$, *for* $\epsilon \in L(E)$
  - $\frac{d(E^*)}{da} = \frac{d(E)}{da}E^*$
- *for* $x = a_1 a_2 \dots a_n$*:*
  - $\frac{dE}{dx} = \frac{d}{da_n}(\frac{d}{da_n - 1}(\dots(\frac{dE}{da_1})\dots))$

To extend the classical derivation of regular expressions, one must deal with the recursive μ operator. To achieve this, Thieman uses a stack to store the current context and starts a new derivation inside the μ operator. However, this simple approach can fail in certain cases. To ensure it works every time, it is essential to distinguish between left-recursive occurrences of a variable and guarded occurrences. The two cases must be treated differently. See the original article [6] for details.

We now remind ourselves how the construction of a finite automaton by the method of partial derivatives works. The idea is that we derive the expression by each symbol of the alphabet. This gives us a new set of expressions. We then derive each of the newly created expressions by each symbol of the alphabet, and we again get a new set of expressions. We repeat this as long as new, non-similar expressions are being created. After this, we create a state of finite automaton for each expression that we created along the way. The original expression becomes the initial state, and each expression that can generate ϵ becomes a final state. Finally, transitions are added followingly. There is a transition $S \to T$ for symbol a *if and only if* derivation of the expression S by the symbol a yields the expression T.

To extend this method to construction of pushdown automaton from a μ-regular expression, Thieman uses the top of the stack of the pushdown automaton as a state. This means that it

is necessary to distinguish between stack operations and word operations. To achieve this, an extra symbol is added that works as a separator on the stack. We refer to the original article [6] for a detailed explanation of the whole process, and furthermore, for the proofs of correctness of the presented results.

Link between CFG and μ-regular expressions

We now present an algorithm described by Gruppen [2] that converts a context-free grammar to a μ-regular expression.

Consider context-free grammar $G = \{N, \Sigma, R, S\}$. The idea of the algorithm is following:

We start by rearranging the rules. We take all rules with the same non-terminal symbol on the left-hand side and we merge them by alternating into one rule. For example, suppose that R contains three rules with S on the left-hand side. Particularly:

$$\begin{aligned} S &\to a \\ S &\to b \\ S &\to aSb \end{aligned}$$

In this case, we merge these three rules into the rule

$$S \to a|b|aSb$$

From now on, we will treat non-terminal symbols as variables. Let us now take the contents of the newly created rule with S on the left-hand side and make it a μ expression. In our case, it would be:

$$\mu S.a|b|aSb$$

We now substitute all free variables with μ expression created from the corresponding rule. There are no free variables in this so the algorithm is finished at this point.

We now show an illustrative example of this algorithm:

▶ **Example 3.4.** Consider context-free grammar $G = \{\{S, T, U\}, \{a\}, R, S\}$, where

$$\begin{aligned} R = \{ & S \to aST \\ & S \to U \\ & T \to TU \\ & T \to S \\ & U \to \epsilon \} \end{aligned}$$

When rearranged, the rules look like this:

$$S \to aST|U$$
$$T \to TU|S$$
$$U \to \epsilon$$

The next step is creating the initial expression:

$$\mu S.aST|U$$

There are two free variables T and U, so we substitute them with their corresponding expressions. The whole expression is then:

$$\mu S.aS(\mu T.TU|S)|(\mu U.\epsilon)$$

However, we added a new variable by this substitution that is not bound at this point. So we substitute it again:

$$\mu S.aS(\mu T.T(\mu U.\epsilon)|S)|(\mu U.\epsilon)$$

There we get the final μ-regular expression that matches the same language as the input context-free grammar G.

We now follow with Gruppen's [2] algortihm of the conversion of a μ-regular expression to a context-free grammar.

Firstly, we need to establish extended context-free grammar. To a common context-free grammar we add a set of variables and we change the right-hand side of rules to be strings of terminal symbols, non-terminal symbols *and* variables. It than looks like this:

$$G' = (N, \Sigma, R', S, X),$$

where X is the set of variables and

$$R' : N \to (N \cup \Sigma \cup X)^*.$$

Note that if $X = \emptyset$, the G' behaves exactly as common context-free grammar.

Now we can define function $\phi : \mathbf{R}(\Sigma, X) \to G'$:

- $\phi(0) = (\{S\}, \Sigma, \{S \to \emptyset\}, S, \{\})$
- $\phi(1) = (\{S\}, \Sigma, \{S \to \epsilon\}, S, \{\})$
- $\phi(a) = (\{S\}, \Sigma, \{S \to a\}, S, \{\})$, where $a \in \Sigma$
- $\phi(x) = (\{S, x\}, \Sigma, \{S \to x\}, S, \{x\})$, where $x \in X$
- $\phi(rs) = (\{S\} \cup N_r \cup N_s, \Sigma, \{S \to S_r S_s\} \cup R_r \cup R_s, S, X_r \cup X_s)$,
  where $\phi(r) = (N_r, \Sigma, R_r, S_r, X_r)$ and $\phi(s) = (N, \Sigma, R_s, S_s, X_s)$

- $\phi(r|s) = (\{S\} \cup N_r \cup N_s, \Sigma, \{S \to S_r, S \to S_s\} \cup R_r \cup R_s, S, X_r \cup X_s)$, where $\phi(r) = (N_r, \Sigma, R_r, S_r, X_r)$ and $\phi(s) = (N, \Sigma, R_s, S_s, X_s)$
- $\phi(\mu x.r) = (\{x\} \cup N_r, \Sigma, \{x \to S_r\} \cup R_r, x, X_r \backslash \{x\})$, where $\phi(r) = (N_r, \Sigma, R_r, S_r, X_r)$

Now we can finally write that $\phi(r) = \phi(r, \emptyset)$ is a context-free grammar equivalent to the *mu*-regular expression r.

For clarity, we add our own example of μ-regular expression converted by the given method.

In the following example, the alphabet Σ stays always the same. Therefore, we will completely omit it from the notation of a grammar (i.e., for $G' = (N, \Sigma, R, S, X)$, we write only $G' = (N, R, S, X)$).

▶ **Example 3.5.** Consider following μ-regular expression over alphabet Σ:

$$E = \mu x.axb|1$$

We will follow the previous definition of ϕ step by step. We index particular sets N_r, R_r, X_r and non-terminal symbols S_r as we encounter them.

$$\phi(E) = \phi(\mu x.axb|1) = (\{x\} \cup N_1, \{x \to S_1\} \cup R_1, x, X_1 \backslash \{x\})$$

To fill in unknown values, we need to work out the next step:

$$(N_1, R_1, S_1, X_1) = \phi(axb|1) = ((\{S_1\} \cup N_2 \cup N_3, \{S_1 \to S_2, S_1 \to S_3\} \cup R_2 \cup R_3, S_1, X_2 \cup X_2))$$

Again, to fill these, we need to the next step with two different expressions. The first one is already the base case:

$$(N_3, R_3, S_3, X_3) = \phi(1) = (\{S_3\}, \{S_3 \to \epsilon\}, S_3, \{\})$$

The other one is a concatenation of, suppose, a and xb:

$$(N_2, R_2, S_2, X_2) = \phi(a(xb)) = ((\{S_2\} \cup N_4 \cup N_5, \{S_2 \to S_4 S_5\} \cup R_4 \cup R_5, S_2, X_4 \cup X_5))$$

This is again a base case:

$$(N_4, R_4, S_4, X_4) = \phi(a) = (\{S_4\}, \{S_4 \to a\}, S_4, \{\})$$

And another concatenatino of two base cases:

$$(N_5, R_5, S_5, X_5) = \phi(xb) = ((\{S_5\} \cup N_6 \cup N_7, \{S_5 \to S_6 S_7\} \cup R_6 \cup R_7, S_5, X_6 \cup X_7))$$

$$(N_6, R_6, S_6, X_6) = \phi(a) = (\{S_6, x\}, \{S_6 \to x\}, S_6, \{x\})$$
$$(N_7, R_7, S_7, X_7) = \phi(b) = (\{S_7\}, \{S_7 \to b\}, S_7, \{\})$$

We have worked out all the values of unknown sets. If we put them all together into the initial function $\phi(E)$, we get

$$\phi(E) = \phi(\mu x.axb|1) = (N, R, x, \{\}),$$

where

$$N = \{x, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}$$

and

$$\begin{aligned} R = \{x &\to S_1 \\ S_1 &\to S_2 \\ S_1 &\to S_3 \\ S_2 &\to S_4 S_5 \\ S_3 &\to \epsilon \\ S_4 &\to a \\ S_5 &\to S_6 S_7 \\ S_6 &\to x \\ S_7 &\to b\} \end{aligned}$$

After closer look, we can see that this output grammar generates the language $\{a^n b^n : n \in \mathbb{N}_0\}$, which is equal to the language matched by the input μ-regular expression.

3.3 Regular Expressions with Assertions

There are many various features and extensions implemented in PCRE expressions as well as in other commonly used libraries. However, since we focus on subpattern recursion in this thesis, the set of features which we selected to study (those described in Definition 2.4) is somewhat narrow. This section shows that by using a PCRE feature *lookahead* (and *lookbehind*, respectively) *assertion*, we can achieve even greater expressive power than by using backreferences or subpattern recursion.

We will present the lookahead and lookbehind assertion, and show some examples. We introduce assertions in a similar way as described by the PCRE project documentation [4], but we again use only a portion of the features. We then present two examples demonstrated by Popov [5]. It shows how to possibly match any context-sensitive language by using extended regular expressions with assertions. However, it cannot be used under the restriction of lookbehind assertion given by PCRE. The second example regards a context-sensitive (and not context-free) language that can be matched by an extended regular expression with lookahead assertion.

Preliminaries

Firstly, we will show how lookahead and lookbehind assertions work. We use our definition of extended regular expressions (Definition 2.4) as a core definition that we further extend by adding the following new rules.

For regular expression a:

- (`?=`a) is regular expression (lookahead assertion)
- (`?<=`a) is regular expression (lookbehind assertion)
- (`?!`a) is regular expression (negative lookahead assertion)
- (`?<!`a) is regular expression (negative lookbehind assertion)

We will not formally define the value of these expressions. Instead, we will describe how do they work intuitively and show some examples.

If we encounter the *lookahead assertion* while trying to match some input string (say at position i of the string), we check if the assertion matches the current part of the input string. This check either fails, which means that the whole match fails. Or, it is successful, in which case we do not continue matching (as we would if this was not the assertion but only a normal part of the expression). Instead, we return to position i of the input string and continue matching from there.

In other words, the lookahead assertion checks its part of the input string but does not "consume" it. So the next section of the expression checks the same part of the input string again, possibly with different rules.

In the case of *negative lookahead assertion*, the failure and success conditions are swapped. That is, if the negative lookahead assertion matches the current part, the whole match fails. On the other hand, if the assertion fails, the match of the input string continues.

In the following example, we use letters $a, b, c, \ldots$ as symbols from the alphabet Σ. Note that we could easily construct a classical regular expression that would match the same language. However, the following example aims to help at understanding how the concept works. We will show more advanced examples later.

▶ **Example 3.6** (Simple expression with (negative) lookahead assertion)**.** Consider the following expression:

$$reg(\texttt{?!}exp).^*$$

This expression always matches "reg". Then it checks if string "exp" follows. If so, the whole match fails. Otherwise, it continues to match ".*", which matches any string.

Therefore, the expression matches all strings beginning with "reg", except for "$regexp$".

The *lookbehind assertion* works in a very similar way. As the name suggests, the only difference is that it does not match the following part of the string but the preceding part. That means that if we encounter the lookbehind assertion (say at position i of the input string), we check if the assertion matches the part of the input string that ends directly before position i.

In the case of *negative lookbehind assertion*, we again simply swap the failure and success conditions of the lookbehind assertion.

Restrictions

It seems like the two assertions behave in a very similar way. However, there is a fundamental difference between the two. In our definition, both lookahead and lookbehind assertion accepted an arbitrary extended regular expression inside them. But there is an important restriction in PCRE. A *lookbehind assertion may only contain expression* such that *all the strings it matches have a fixed length.*

Suppose that we have an implementation without the lookbehind restriction. Then according to Popov [5], it is possible to match a context-sensitive language by using the following approach. However, keep in mind that lookbehind assertion in PCRE *is restricted.*

▶ **Example 3.7** (*Potential* (non-functional) matching of context-sensitive languages)**.** Consider a context-sensitive grammar, where all the rules have the following form:

$$\gamma A \delta \to \gamma \alpha \delta,$$

for $A \in N,\ \alpha \in (N \cup \Sigma)+$ and $\gamma, \delta \in (N \cup \Sigma)^*$.

This way, we would be able express each rule $\gamma A \delta \to \gamma \alpha \delta$ as:

$$(?{<}{=}\gamma)\ \alpha\ (?{=}\delta).$$

The problem with this approach is that since γ can include non-terminal symbols, we may need to express them with a recursive subpattern. However, the restriction of lookbehind assertions does not allow this.

We now know that we cannot match any arbitrary context-sensitive language with the method above, while the lookbehind assertion restriction stands. However, it is possible to express some interesting languages, which we show in the next part.

Advanced example

As the final note, we show Popov's [5] example that at least some context-sensitive languages (that are not context-free) can be matched using assertions. Namely, for the following example, we use only the lookahead assertion. We will show an expression that matches the language $\{a^n b^n c^n : n \geq 1\}$.

▶ **Example 3.8** (Expression with lookahead assertion matching a context-sensitive language)**.** Consider the following expression e over alphabet $\Sigma = \{a, b, c\}$:

$$e = (?{=}((a(?1)b)|ab)c)(a+((b(?4)c)|bc)).$$

For clarity, we show the corresponding numbers of each bracket pair. Note that we also mark the bracket pair of the assertion as la:

$$({}_{la}?{=}({}_1({}_2a(?1)b)_2|ab)_1c)_{la}({}_3a+({}_4({}_5b(?4)c)_5|bc)_4)_3.$$

Let us now examine this expression. The first part is a lookahead assertion and it contains bracket pair number 1 followed by one symbol c. We can see that bracket pair number 1 matches a nonempty sequence of symbols a followed by the same number of symbols b.

This means that the whole assertion checks whether the input string begins with a word from $\{a^i b^i c : i \geq 1\}$.

The second part, enclosed in bracket pair 3, must match the whole input since there was only a lookahead assertion before. It contains (at least one) repetition of symbol a followed by bracket number 4, which matches a nonempty sequence of symbols b followed by the equal number of symbols c.

This means that the whole second part checks whether the input string matches a word from $\{a^k b^l c^l : k \geq 1, l \geq 1\}$.

All together, we know that each matched word w must satisfy both

$$w \in \{a^i b^i c^j : i \geq, j \geq 1\}$$

and

$$w \in \{a^k b^l c^l : k \geq 1, l \geq 1\}.$$

Finally, combining these two gives us $w \in \{a^n b^n c^n : n \geq 1\}$ and therefore the language matched by the expression e is

$$L(e) = \{a^n b^n c^n : n \geq 1\}.$$

Chapter 4

Study of Expressive Power

In this chapter, we study expressive power and other properties of regular expressions with subpattern recursion and present our own results. We show that they can express more than classical regular expressions. The first section shows that they can match all context-free languages, contrary to regular expressions extended with backtracking. This first result then points us to an observation — there is an unexpected redundancy in our definition. Then we study them further and, having the work more manageable due to the previous observation, we show that any regular expression extended with subpattern recursion can be transformed to a context-free grammar. To put it differently, we can say that each extended regular expression matches some context-free grammar. All of the outcomes throughout the chapter then lead us to the final result we present at the very end of the chapter. The result is that extended regular expressions match exactly the set of context-free languages.

Acknowledgment

Please note that extended regular expressions (our definition according to PCRE) and μ-regular expressions (introduced in the previous chapter) are very similar, and it should be possible to convert one into the other. However, we decided to study, transform, and compare them with context-free languages independently.

4.1 Conversion of CFG to ERE

We first show an algorithm that can transform any context-free grammar into extended regular expression. It is inspired by a procedure described by Popov [5]. Input of the following algorithm is a context-free grammar $G = (N, \Sigma, R, A_1)$ and we will use the following notation throughout this section:

- denote the number of non-terminal symbols as n
- denote the non-terminal symbols as $A_1, A_2, \ldots, A_n$

- denote the number of rules with non-terminal symbol A_i on the left-hand side as $|A_i|$
- denote the right-hand sides of each rule with non-terminal symbol A_i on the left-hand side as $\alpha_i^{(1)}, \alpha_i^{(2)}, \ldots, \alpha_i^{(|A_i|)}$

Algorithm description

The algorithm starts with creating a new expression on the line 1 of Algorithm 1. Note that at this point, the expression may contain both terminal and non-terminal symbols and, therefore, it is not valid extended regular expression over alphabet Σ. However, the non-terminal symbols are rewritten throughout the algorithm such that the final expression is valid extended regular expression.

The array *subpatterns* defined on the line 2 of Algorithm 1 of size n stores the order number for each non-terminal symbol A_i in ith position. The order number simply denotes in what order did the non-terminal symbols (written inside a bracket pair) appear. That is, for example, $(?j)$ where $j = subpatterns[i]$, references symbol A_i.

Note that the while loop on line 5 of Algorithm 1 terminates when there are not any non-terminal symbols in E. The branch on line 7 can add arbitrary number of non-terminal symbols to the expression E. However, this branch is passed at most once for each non-terminal symbols. On the other hand, the branch on line 11, which is passsed in all other cases, always removes exactly one non-terminal symbol from E. Therefore, the algorithm will always end and the final expression contained in E will be a valid extended regular expression as described in Definition 2.4.

The actual description of the algorithm follows now:

```
   input  : context-free grammar G = (N, Σ, R, A_1),
            where N = {A_1, A_2, ..., A_n}
   output: extended regular expression E over alphabet Σ
 1 E := (α_1^(1) | α_1^(2) | ... | α_1^(|A_1|))   //initial expression
 2 subpatterns[i] := 0 for i ∈ {1, 2, ..., n}   //zero-initialized array of size n
 3 l := 1   //number of the last assigned subpattern
 4 subpatterns[1] := l
 5 while E contains at least one non-terminal symbol do
 6 |   let A_i = leftmost non-terminal symbol in E
 7 |   if subpatterns[i] = 0 then
 8 |   |   rewrite the leftmost occurrence of A_i in E with (α_i^(1) | α_i^(2) | ... | α_i^(|A_i|))
 9 |   |   l := l + 1
10 |   |   subpatterns[i] := l
11 |   else
12 |   |   j := subpatterns[i]   //the number of subpattern according to the
   |   |       currently processed non-terminal symbol
13 |   |   rewrite the leftmost occurrence of A_i in E with (?j)
14 output E
```

Algorithm 1: Constructing expression from context-free grammar

Examples

We first show a detailed example of a context-free grammar transformed by Algorithm 1 to an extended regular expression in Example 4.1. There, we go through the procedure step by step. The first example is then followed by Example 4.2, where we show equivalence of language generated by a simple context-free grammar and value of this grammar transformed to an extended regular expression by Algorithm 1.

▶ **Example 4.1.** Consider Algorithm 1 given grammar $G = (N, \Sigma, R, S)$, where:

$$\begin{aligned} N &= \{S, A, B\} \\ \Sigma &= \{a, b\} \\ R &= \{S \to ASA, \\ &\quad S \to \epsilon, \\ &\quad A \to aA, \\ &\quad A \to aAb, \\ &\quad A \to B, \\ &\quad B \to a\} \end{aligned}$$

To keep the example clear, we will rename non-terminal symbols consistently to the notation used in the algorithm in the following way:

$$S = A_1; A = A_2; B = A_3.$$

Then, we can write N as: $N = \{A_1, A_2, A_3\}$ and R as:

$$\begin{aligned} R = \{&A_1 \to A_2A_1A_2, \\ &A_1 \to \epsilon, \\ &A_2 \to aA_2, \\ &A_2 \to aA_2b, \\ &A_2 \to A_3, \\ &A_3 \to a\} \end{aligned}$$

The first step of the algorithm is the initialization, which happens on lines 1–4. It assigns the initial expression to E, then it fills array *subpatterns* with zeroes, assigns number 1 to the variable l and finally sets *subpatterns*[1] to 1. The variables contain following values after the initialization:

		subpatterns[i]		
E	l	1	2	3
$(A_2A_1A_2\|\epsilon)$	1	1	0	0

- Then, on line 5, we enter a while loop.
- We check E for non-terminal symbols and as there are non-terminal symbols, we continue on line 6.
- We find the leftmost non-terminal symbol in $E = (\mathbf{A_2}A_1A_2|\epsilon)$, which is A_2.

- On line 7, we check the condition $subpatterns[2] = 0$, which is true. So we continue in the true branch on line 8.
- We rewrite A_2 in E as $\mathbf{(aA_2|aA_2b|A_3)}$. E then has the following form: $(\mathbf{(aA_2|aA_2b|A_3)}A_1A_2|\epsilon)$.
- Finally, on lines 9 and 10, we update the array and l.
- We reached the end of the while loop. The variables contain the following values at this moment:

		subpatterns[i]		
E	l	1	2	3
$((aA_2\|aA_2b\|A_3)A_1A_2\|\epsilon)$	2	1	2	0

- We again check E for non-terminal symbols and as there still are some, we continue as before.
- We find the leftmost non-terminal symbol in $E = ((a\mathbf{A_2}|aA_2b|A_3)A_1A_2|\epsilon)$ to be A_2 again.
- The condition $subpatterns[2] = 0$ is false this time, so we continue in the false branch on line 12.
- Since $subpatterns[2] = 2$, we rewrite A_2 in E as $\mathbf{(?2)}$. E then has the following form: $((a\mathbf{(?2)}|aA_2b|A_3)A_1A_2|\epsilon)$.
- We reached the end of the while loop. The variables contain the following values at this moment:

		subpatterns[i]		
E	l	1	2	3
$((a(?2)\|aA_2b\|A_3)A_1A_2\|\epsilon)$	2	1	2	0

- In the following iteration we again find A_2 as the leftmost non-terminal symbol and we rewrite it. The variables then contain:

		subpatterns[i]		
E	l	1	2	3
$((a(?2)\|a(?2)b\|A_3)A_1A_2\|\epsilon)$	2	1	2	0

- In the next iteration we find A_3 as the leftmost non-terminal symbol.
- We update E, l and $subpatterns[3]$. The variables now contain:

		subpatterns[i]		
E	l	1	2	3
$((a(?2)\|a(?2)b\|(a))A_1A_2\|\epsilon)$	2	1	2	3

The array *subpatterns* contains a non-zero number for each non-terminal symbol. This means that in each following iteration, we replace one non-terminal symbol with a subpattern. This

repeats until E does not contain any non-terminal symbol. The remaining replacements on E then are

$$E = ((a(?2)|a(?2)b|(a))(?1)(A_2)|\epsilon)$$

and

$$E = ((a(?2)|a(?2)b|(a))(?1)(?2)|\epsilon)$$

respectively.

We have shown that the algorithm outputs extended regular expression $((a(?2)|a(?2)b|(a))(?1)(?2)|\epsilon)$ for grammar

$$\begin{aligned} G = \{&\{S, A, B\},\\ &\{a, b\},\\ &\{S \to ASA,\\ &\ S \to \epsilon,\\ &\ A \to aA,\\ &\ A \to aAb,\\ &\ A \to B,\\ &\ B \to a\},\\ &\ S\} \end{aligned}$$

▶ **Example 4.2.** Consider the context-free grammar $G = (N, \Sigma, R, S)$, where:

$$\begin{aligned} N &= \{S\}\\ \Sigma &= \{a, b\}\\ R &= \{S \to aSb,\\ &\quad\ S \to \epsilon\} \end{aligned}$$

The language generated by G is context-free and is equal to $L(G) = \{a^n b^n : n \in \mathbb{N}_0\}$.

If we use the grammar G as an input to Algorithm 1, we get $E = (a(?1)b|\epsilon)$ as output. Let us show the language matched by E, according to Definition 2.5:

$$\begin{aligned} L(E) &= L(a(?1)b) && \cup L(\epsilon)\\ &= L(a(?1)b) && \cup \{\epsilon\}\\ &= L(a)L((?1))L(b) && \cup \{\epsilon\}\\ &= \{a\}L((?1))\{b\} && \cup \{\epsilon\}\\ &= \{a\}L(E)\{b\} && \cup \{\epsilon\}\\ &= \{a\}(L(a(?1)b) \cup L(\epsilon))\{b\} && \cup \{\epsilon\}\\ &= \{a\}L(a(?1)b)\{b\} && \cup \{a\}\{\epsilon\}\{b\} && \cup \{\epsilon\}\\ &= \{a\}\{a\}L((?1))\{b\}\{b\} && \cup \{ab\} && \cup \{\epsilon\}\\ &= \{aa\}L(E)\{bb\} && \cup \{ab\} && \cup \{\epsilon\}\\ &= \{aaa\}L(E)\{bbb\} && \cup \{aabb\} && \cup \{ab\} && \cup \{\epsilon\}\\ &= \{aaaa\}L(E)\{bbbb\} && \cup \{aaabbb\} && \cup \{aabb\} && \cup \{ab\} && \cup \{\epsilon\}\\ &\ \ \vdots\\ &= \{a^n b^n : n \in \mathbb{N}_0\} \end{aligned}$$

It is clear that in this particular case, language generated by the grammar G is equal to the language matched by extended regular expression E constructed from G by Algorithm 1.

Correctness

We now show that we always achieve this exact result. Namely, we show that each regular expression E constructed by Algorithm 1 matches exactly the same language as the language generated by the input grammar G. To achieve this, we first present an alternative definition of language generated by a context-free grammar. (That is, an alternative definition to Definition 2.13.) Using this new definition, we then show the proof.

▶ **Definition 4.1** (Equivalent definition of language generated by context-free grammar)**.** *Let $G = (N, \Sigma, R, S)$ be a context-free grammar. For each non-terminal symbol $A \in N$: denote right-hand sides of rules with A on the left-hand side as $\{\alpha_1, \alpha_2, \ldots, \alpha_{|A|}\}$, where each $\alpha_i \in (N \cup \Sigma)^*$. For each right-hand side $\alpha \in (N \cup \Sigma)^*$: denote each symbol it contains as $\alpha = x_1 x_2 \ldots x_{|\alpha|}$, where each $x_i \in (N \cup \Sigma)$. If we inductively use the following rules:*

1. *for each $a \in \Sigma \cup \{\epsilon\} : L(a) = \{a\}$*
2. *for each $A \in N : L(A) = L(\alpha_1) \cup L(\alpha_2) \cup \ldots \cup L(\alpha_{|A|})$*
3. *for each $\alpha = x_1 x_2 \ldots x_{|\alpha|} \in (N \cup \Sigma)^* : L(\alpha) = L(x_1)L(x_2)\ldots L(x_{|\alpha|})$*

Then $L(G) = L(S)$.

We first need to show that the two definitions are equal.

▶ **Lemma 4.1** (Equivalence of definitions)**.** *For context-free grammar $G = (N, \Sigma, R, S)$, let language $L_1(G)$ and language $L_2(G)$ be languages generated by grammar G using Definition 2.13 and Definition 4.1 respectively. Then $L_1(G) = L_2(G)$.*

Proof. To prove $L_1(G) = L_2(G)$, we show that for each word $w \in \Sigma^* : w \in L_1(G) \Leftrightarrow w \in L_2(G)$.

"⇒": Suppose that $w \in L_1(G)$, which is defined as $S \Rightarrow^* w$ according to Definition 2.13. This means that there is a sequence of sentential forms

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow w.$$

The first derivation $S \Rightarrow \alpha_1$ replaces S by one of the right-hand sides of rules with S on the left-hand side α_1. According to rule (2) of Definition 4.1, $L(S)$ includes $L(\alpha_1)$.

The grammar G is context-free, so each subsequent derivation $\alpha_i \Rightarrow \alpha_{i+1}$ replaces a non-terminal symbol A by one of the rules. But according to rule (2), $L(A)$ includes each possibility for the replacement

Additionally, since G is context-free, each derivation $\alpha_i \Rightarrow \alpha_{i+1}$ replaces *exactly one* non-terminal symbol by one of the rules. Therefore, each terminal symbol in α_i stays in the place and it cannot be removed or changed in any way. The same effect in Definition 4.1 is achieved by rule (1), which assigns each terminal symbol its value and by rule (3), which concatenates values of all symbols and thus anchoring terminal symbols in one place. Therefore, for each word $w \in \Sigma^*$ the following implication holds

$$w \in L_1(G) \Rightarrow w \in L_2(G).$$

"⇐": Suppose that $w \in L_2(G)$. We will construct a sequence of sentential forms

$$seq = S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow w$$

to show that also $w \in L_1(G)$.

Consider $L(S)$ generated by rule (2) as

$$L(\alpha_1) \cup L(\alpha_2) \cup \ldots \cup L(\alpha_{|S|}).$$

Choose $L(\alpha_i)$ such that $w \in L(\alpha_i)$. There has to be at least one since $w \in L_2(G)$. The first derivation of *seq* is then $S \Rightarrow \alpha_i$.

For the next step, we recall that α_i is a concatenation

$$x_1 x_2 \ldots x_{|\alpha_i|}$$

of both terminal and non-terminal symbols and $L(\alpha_i)$ is generated by rule (3) as

$$L(x_1)L(x_2)\ldots L(x_{|\alpha_i|}).$$

We now choose x_j to be the leftmost non-terminal symbol. Its value

$$L(x_j) = L(\beta_1) \cup L(\beta_2) \cup \ldots \cup L(\beta_{|x_j|})$$

is then generated by rule (2) identically as before in case of $L(S)$. We again choose $L(\beta_k)$ such that $w \in L(\beta_k)$. The following derivation in *seq* is then $\alpha_i \Rightarrow \beta_k$.

We repeat this step until we get a sentential form with terminal symbols only, which is w.

That is, we constructed a sequence of sentential forms

$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow w$$

for a word from $L_2(G)$. Therefore, for each word $w \in \Sigma^*$ the following implication holds

$$w \in L_2(G) \Rightarrow w \in L_1(G).$$

◀

▶ **Theorem 4.1** (Equivalence of generated languages)**.** *Set of words generated by context-free grammar G and set of words matched by the extended regular expression constructed from grammar G by Algorithm 1 are equal.*

Proof. Consider context-free grammar $G = (N, \Sigma, R, S)$ and extended regular expression E created from G by Algorithm 1. Let L_G be the language generated by the grammar G and L_E the language matched by the extended regular expression E.

The first step of Algorithm 1 is creating the following expression

$$(\alpha_1^{(1)}|\alpha_1^{(2)}|\ldots|\alpha_1^{(|A_1|)}),$$

where $A_1 = S$. It then replaces all of the non-terminal symbols in

$$\alpha_1^{(1)}, \alpha_1^{(2)}, \ldots, \alpha_1^{(|A_1|)}$$

either with $(\alpha_i^{(1)}|\alpha_i^{(2)}|\ldots|\alpha_i^{(|A_i|)})$ or $(?j)$. Note that the latter has exactly the same value as the corresponding expression $(\alpha_j^{(1)}|\alpha_j^{(2)}|\ldots|\alpha_j^{(|A_j|)})$.

The value of E is according to Definition 2.5 following:

$$L_E = L(\alpha_1^{(1)}) \cup L(\alpha_1^{(2)}) \cup \ldots \cup L(\alpha_1^{(|A_1|)}).$$

(a) Subsequent values in each α are concatenated. (b) Each terminal symbol a in each α has value $L(a) = \{a\}$. (c) Each non-terminal symbol A_i in each α is replaced with a new expression with value

$$L(\alpha_i^{(1)}) \cup L(\alpha_i^{(2)}) \cup \ldots \cup L(\alpha_i^{(|A_i|)}).$$

For determining L_G, we use Definition 4.1. See that

$$L_G = L(S) = L(\alpha_1) \cup L(\alpha_2) \cup \ldots \cup L(\alpha_{|S|}),$$

which is the same as

$$L(\alpha_1^{(1)}) \cup L(\alpha_1^{(2)}) \cup \ldots \cup L(\alpha_1^{(|A_1|)}),$$

where $A_1 = S$, if we use the same notation that we used for the expressions.

Now we can see, that from this point on, determining the value of E (by Definition 2.5) and language generated by G (by Definition 4.1) are exactly the same. Point (a) of the previous paragraph corresponds to rule (3) of Definition 4.1, point (b) corresponds to rule (1) and point (c) corresponds to rule (2). ◀

4.2 Redundancy in ERE

An interesting observation follows from our approach to creating an extended regular expression in the previous section. We never used the Kleene star in our algorithm. However, we were still able to express any arbitrary context-free grammar. This indicates that the Kleene star might not be necessary for our definition at all.

First, recall our definitions in the preliminaries chapter. We simply took the classical regular expressions and extended them with new features according to common usage. Namely, to the following definition of classical regular expressions:

(1) $\emptyset$, ϵ and each $x \in \Sigma$ are regular expressions,

(2) for regular expressions a, b:

- $(a|b)$ is regular expression
- (ab) is regular expression
- $(a)^*$ is regular expression

We added the following features:

(3) . is regular expression,

(4) for regular expression a:

- $(a)+$ is regular expression
- $(a)?$ is regular expression

(5) for $i \in \mathbb{N}$ and $x_1, x_2, \ldots, x_i \in \Sigma$:

- $[x_1, x_2, \ldots, x_i]$ is regular expression
- $[\^{}x_1, x_2, \ldots, x_i]$ is regular expression

(6) for $i < j,\ i, j \in \mathbb{N}$, some ordering of Σ and $x_i, x_j \in \Sigma$:

- $[x_i\text{-}x_j]$ is regular expression
- $[\^{}x_i\text{-}x_j]$ is regular expression

(7) for $i \leq j,\ i, j \in \mathbb{N}_0$ and regular expression a:

- $(a)\{i\}$ is regular expression
- $(a)\{i,\}$ is regular expression
- $(a)\{i, j\}$ is regular expression

(8) for $i \in \mathbb{N}$:

- $(?i)$ is regular expression.
- $(?\text{-}i)$ is regular expression.

As we pointed out in Observation 2.1, some features (namely items (3) through (7)) of extended regular expressions are redundant. That means they serve only as a shortcut or syntactic sugar that makes practical work with them more comfortable. However, if we were to remove these features, we would still be able to express precisely the same set of languages as with the features present.

However, it turns out that the Kleene star (*) is also redundant in extended regular expressions. It is very much needed in classical regular expressions since we would be able to express only finite languages without it. (For example, it would be impossible to express regular language $L = \{a^n : n \in \mathbb{N}\}$ without the Kleene star) Yet, in extended regular expressions, we can achieve the same result of Kleene star by using the subpattern recursion.

We now show how to construct such expression. Then we formulate it formally as a lemma and present a proof. Let a be an extended regular expression. Then a^* can be replaced by e, where $e = (a(?1)|\epsilon)$ and the two expressions are equivalent.

▶ **Theorem 4.2** (An alternative to Kleene star)**.** *For extended regular expressions a and $e = (a(?1)|\epsilon)$, the following is true:*

$$L(a^*) = L(e)$$

Proof. We can rewrite the left-hand side as follows:

$$L(a^*) = L(a)^* = \{\epsilon\} \cup L(a) \cup L(a)L(a) \cup L(a)L(a)L(a) \cup \ldots$$

We can expand the right-hand side in the following way:

$$\begin{aligned} L(e) &= L((a(?1)|\epsilon)) \\ &= L(\epsilon) \cup L(a)L(e) \\ &= \{\epsilon\} \cup L(a)L((a(?1)|\epsilon)) \\ &= \{\epsilon\} \cup L(a)L(\epsilon) \cup L(a)L(a)L(e) \\ &= \{\epsilon\} \cup L(a) \cup L(a)L(a)L((a(?1)|\epsilon)) \\ &= \{\epsilon\} \cup L(a) \cup L(a)L(a) \cup L(a)L(a)L(e) \\ &\vdots \\ &= \{\epsilon\} \cup L(a) \cup L(a)L(a) \cup \dots \end{aligned}$$

It is now clear that $L(a^*) = L(e)$. ◀

We have shown that the Kleene star is indeed redundant in extended regular expressions. We can now present a minimal version of the definition of extended regular expressions. It is minimal in the sense that all features described in Definition 2.4 can be achieved using only the features in Definition 4.2.

▶ **Definition 4.2** (Extended regular expression (minimal version))**.** *Let Σ be an alphabet that does not include any metacharacters. But Σ may include $\backslash m$ for $m \in M$.*

(1) $\emptyset$, ϵ and each $x \in \Sigma$ are regular expressions,

(2) for regular expressions a, b:

- *$(a|b)$ is regular expression*
- *(ab) is regular expression*

(3) $(?i)$ is regular expression.

4.3 Conversion of ERE to CFG

This final section presents an algorithm that can transform any extended regular expression to a context-free grammar generating the same language. Being able to convert both CFG to ERE and the other way around means that the two systems have, in fact, the same expressive power. That is, the set of languages we can generate by using context-free grammars is the same as the set we can match by using extended regular expressions.

After showing the algorithm, we will finally be able to formulate the main contribution of this thesis. That is, extended regular expressions match exactly the context-free languages.

Algorithm description

The general idea of the algorithm is to create a non-terminal symbol for each bracket pair in the input expression. Rules of the output grammar are constructed so that each non-terminal

expression can generate precisely the language matched by the subexpression in corresponding bracket pair. Firstly, we introduce some preliminaries.

In the previous section, we demonstrated that each extended regular expression can be represented by an equivalent expression using only features listed in Definition 4.2. We will use this property for the input of Algorithm 2. Namely, we assume that the input extended regular expression E' is composed only of concatenations, alternations, and subpatterns of symbols of the alphabet Σ and ϵ.

We generally accept expressions with some brackets omitted since operation precedence prevents ambiguity in these cases. However, for the following algorithm, we require the input E' to include all brackets according to Definition 4.2.

Additionally, we will use the following notation throughout this section:

- we use r and s to denote unspecified regular expressions
- when talking about bracket pairs, we do not consider brackets that are enclosing some $(?i)$ as bracket pairs. (I.e., for counting)
- we denote the number of bracket pairs in E' as n'
- we denote the number of bracket pairs in E as n
- if we explicitly use notation ${}_{(i}r)_i$ as ith bracket pair with some subexpression r, it is always ith bracket pair *bound to the expression* E

The algorithm begins on the lines 1 and 2 by adding a bracket pair around the input expression E' and thus creating a new expression $E = (E')$. This operation adds one bracket pair, so the total number n of bracket pairs in E is one higher $n = n' + 1$.

We follow on line 3 by renumbering each occurrence of $(?i)$ in E with a number higher by one. We do this because the bracket pair that we added before has number 1. So the bracket pair with number 1 in the original expression E' has number 2 in E and so on. By this renumbering, we achieve that the expression E matches exactly the same language as the input E'. Therefore, we can only work with the equivalent expression E from now on.

Beginning on line 4, we define the set of non-terminal symbols as

$$N = \{S, A_1, A_2, \ldots, A_n\}$$

and then the set of rules as an empty set $R = \{\}$. Immediately after, we add the first rule $S \to A_1$ to R.

As the definition of N hints, we create a non-terminal symbol for each bracket pair in E. This will help us to make the rest of the algorithm clear. We have concluded the initialization, and we can now move further on.

Since we created a non-terminal symbol for each bracket pair, it is natural that we now want to create rules according to the contents of the corresponding bracket pair. We will now iterate over all bracket pairs in E in the loop on line 7.

First, we denote the contents of the current (say ith) bracket pair as c (as on line 8). Then, according to Definition 4.2, contents of c, with respect to the whole bracket pair, must be one of

input : extended regular expression E' over alphabet Σ
output: context-free grammar $G = \{N, \Sigma, R, S\}$

```
1  E := (E')  //add a bracket pair around the input E'
2  n := n' + 1  //number of bracket pairs in E
3  replace each (?i) ∈ E by (?j), where j = i + 1
4  N := {S, A_1, A_2, ..., A_n}
5  R := {}
6  add rule S → A_1 to R
7  for i ∈ {1, 2, ..., n} do
8      c := contents of ith bracket pair
9      switch c do
10         case c ∈ Σ ∪ {ϵ} do
11             add rule A_i → c to R
12         case c ≡ rs do
13             α := expressionToSymbol(r)
14             β := expressionToSymbol(s)
15             add rule A_i → αβ to R
16         case c ≡ r|s do
17             α := expressionToSymbol(r)
18             β := expressionToSymbol(s)
19             add rule A_i → α to R
20             add rule A_i → β to R
21         case c ≡ (?j) do
22             add rule A_i → A_j to R
23         case c ≡ (r) do
24             add rule A_i → A_{i+1} to R
25 Function expressionToSymbol(regular expression r):
26     if r ∈ Σ ∪ {ϵ} then
27         return r
28     else if r ≡ (?i) then
29         return A_i
30     else
31         i := number assigned to the outermost bracket pair in r
32         return A_i
33 output G = {N, Σ, R, S}
```

Algorithm 2: Constructing context-free grammar from expression

the following:

$$\begin{aligned}(\epsilon) &\Rightarrow c = \epsilon \\ (x) &\Rightarrow c = x, \text{where } x \in \Sigma \\ (rs) &\Rightarrow c = rs \\ (r|s) &\Rightarrow c = r|s \\ ((?i)) &\Rightarrow c = (?i)\end{aligned}$$

For the next part, suppose that function `expressionToSymbol`(r) takes an expression r and returns $\alpha \in \Sigma \cup N$, either a terminal or non-terminal symbol. We will cover the role of

`expressionToSymbol`(r) later on.

Now we add rules to R with A_i on the left-hand side. Represented by the switch statement on line 9, the number of rules and contents of the right-hand sides depend on the form of c:

- (**line 10**): $c \in \Sigma \cup \{\epsilon\}$
  If c is ϵ or a terminal symbol, we simply put c on the right-hand side and add this rule to R.
  - adding $A_i \to c$
- (**line 12**): $c \equiv rs$
  If c is a concatenation of two expressions r and s, we generate two symbols α and β as $\alpha =$ `expressionToSymbol`(r) and $\beta =$ `expressionToSymbol`(s) respectively. We then use the concatenation of the generated symbols as the right-hand side of a new rule.
  - adding $A_i \to \alpha\beta$
- (**line 16**): $c \equiv r|s$
  If c is an alternation of two expressions r and s, we again generate two symbols α and β as in the previous case. However, this time we add two rules with one of the symbols as the right-hand side of each respective rule.
  - adding $A_i \to \alpha$
  - adding $A_i \to \beta$
- (**line 21**): $c \equiv (?j)$
  If c is $(?j)$, a reference of subpattern enclosed in jth bracket pair, we add a rule with only the non-terminal symbol A_j on the right-hand side.
  - adding $A_i \to A_j$
- (**line 23**): $c \equiv (?j)$
  There is one last possibility if c does not match any of the previous characterization. It contains composed expression inside an extra bracket pair, $c = (r)$. The numbering in this part of E then looks like this: $\ldots({}_i({}_{i+1}r)_{i+1})_1\ldots$ This is just a redundant bracket pair, so we move on to the next step by adding a rule with the next non-terminal A_{i+1} on the right-hand side.
  - adding $A_i \to A_{i+1}$

The last part of Algorithm 2 left is a function called `expressionToSymbol`(r) on line 23, and we will explain its purpose now.

We use `expressionToSymbol` in two cases. Either if we are in a (sub)expression that is a concatenation of two expressions r and s, or if we are in a (sub)expression that is an alternation of two expressions r and s. Let us now examine what form r and s have, according to Definition 4.2.

It can be ϵ or a terminal symbol, in which case we want to have it directly contained on the right-hand side of a rule. This corresponds to the output on line 25.

Another possibility is that it is a reference of a subpattern $(?i)$, in which case we want to add a corresponding non-terminal symbol to the right-hand side of a rule. This corresponds to the output on line 27.

The last possibility is that it is a composed expression enclosed by a bracket pair, in which case we want to add a non-terminal symbol corresponding to this bracket pair to the right-hand side of a rule. This corresponds to the output on line 30.

Examples

We first show a detailed example of an extended regular expression transformed by Algorithm 2 to a context-free grammar. There, we go through the procedure step by step.

The first example is then followed by another one, where we show equivalence of language matched by a simple extended regular expression E and language generated by a context-free grammar obtained as an output of Algorithm 2 for the expression E.

▶ **Example 4.3.** Consider extended regular expression over alphabet $\Sigma = \{a, b\}$:

$$E' = (((((a(?4))|\epsilon)(?1))(?4))|(b))$$

For clarity, we also show numbering of the bracket pairs:

$$(_1(_2(_3(_4(_5a(?4))_5|\epsilon)_4(?1))_3(?4))_2|(_6b)_6)_1$$

The very first step of Algorithm 2 is adding a bracket pair around the input expression E' and accordingly renumbering the subpattern references. This gives us the expression E and looks like this:

$$E = ((((((a(?5))|\epsilon)(?2))(?5))|(b)))$$

The numbering of brackets is then following:

$$(_1(_2(_3(_4(_5(_6a(?5))_6|\epsilon)_5(?2))_4(?5))_3|(_7b)_7)_2)_1$$

To avoid any confusion, from now on we will always write the numbers of each bracket pair throughout the example.

Additionally, the number of bracket pairs in E is $n = 7$. Then, we may continue in initialization by defining

$$N = \{S, A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$$

and R with one rule

$$R = \{S \to A_1\}.$$

$\mathbf{i = 1}$: Now we can enter the for loop on line 7. Starting with number 1, contents of the first bracket pair are following:

$$c = (_2(_3(_4(_5(_6a(?5))_6|\epsilon)_5(?2))_4(?5))_3|(_7b)_7)_2$$

Bracket pair number 2 encloses the whole expression. That means we are in switch case $c \equiv (r)$. Therefore, we add rule $A_1 \to A_2$ to R. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & \mathbf{A_1 \to A_2}\} \end{aligned}$$

i = 2: We can continue to the following iteration. Contents of the second bracket pair are following:

$$c = ({}_3({}_4({}_5({}_6a(?5))_6|\epsilon)_5(?2))_4(?5))_3|({}_7b)_7$$

This time, we encounter alternation. That means we are in switch case $c \equiv r|s$. We can show the exact values of r and s:

$$\begin{aligned} r &= ({}_3({}_4({}_5({}_6a(?5))_6|\epsilon)_5(?2))_4(?5))_3 \\ s &= ({}_7b)_7 \end{aligned}$$

To determine what rules are to be added, we need to evaluate function `expressionToSymbol`() for both r and s. We can see that both r and s start with an opening bracket. Particularly, r starts with opening bracket 3 and s starts with opening bracket 7. This means that the two rules we add to R are $A_2 \to A_3$ and $A_2 \to A_7$. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & \mathbf{A_2 \to A_3} \\ & \mathbf{A_2 \to A_7}\} \end{aligned}$$

i = 3: We continue to the third iteration. Contents of the corresponding bracket pair are:

$$c = ({}_4({}_5({}_6a(?5))_6|\epsilon)_5(?2))_4(?5)$$

This time, we find concatenation. That means we are in switch case $c \equiv (rs)$. The values of r and s are:

$$\begin{aligned} r &= ({}_4({}_5({}_6a(?5))_6|\epsilon)_5(?2))_4 \\ s &= (?5) \end{aligned}$$

We again need to evaluate function `expressionToSymbol`() for r and s. In the case of r, we get A_4 as a result. This is similar to the previous iteration, as r starts with an opening bracket number 4. However, s is a subpattern reference. This means that we get A_5 because it references the fifth bracket pair. Finally, we can compose the rule we are going to add to R as $A_3 \to A_4A_5$. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & A_2 \to A_3 \\ & A_2 \to A_7 \\ & \mathbf{A_3 \to A_4A_5}\} \end{aligned}$$

i = 4: We can continue to the next iteration. Contents of the fourth bracket pair are:

$$c = ({}_5({}_6a(?5))_6|\epsilon)_5(?2)$$

We find concatenation once again. The procedure is exactly the same as in the previous iteration. Only with different numbers. We get a new rule $A_4 \to A_5A_2$. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & A_2 \to A_3 \\ & A_2 \to A_7 \\ & A_3 \to A_4A_5 \\ & \mathbf{A_4 \to A_5A_2}\} \end{aligned}$$

i = 5: We continue to the iteration 5. Contents of this bracket pair are:

$$c = ({}_6a(?5))_6|\epsilon$$

This time, we encountered alternation. That means we are in switch case $c \equiv r|s$. The exact values of r and s are:

$$\begin{aligned} r &= ({}_6a(?5))_6 \\ s &= \epsilon \end{aligned}$$

We evaluate `expressionToSymbol`() and get A_6 as an output for r because it starts with opening bracket number 6. On the other hand, s is equal to ϵ, so `expressionToSymbol`(s) outputs exactly ϵ. This gives us the two rules we add to R to be $A_5 \to A_6$ and $A_5 \to \epsilon$. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & A_2 \to A_3 \\ & A_2 \to A_7 \\ & A_3 \to A_4A_5 \\ & A_4 \to A_5A_2 \\ & \mathbf{A_5 \to A_6} \\ & \mathbf{A_5 \to \epsilon}\} \end{aligned}$$

i = 6: We continue to the iteration 6. Contents of sixth bracket pair are:

$$c = a(?5)$$

For this concatenation, outputs of `expressionToSymbol`() for r and s are a and A_5. That gives us a new rule $A_6 \to aA_5$. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & A_2 \to A_3 \\ & A_2 \to A_7 \\ & A_3 \to A_4 A_5 \\ & A_4 \to A_5 A_2 \\ & A_5 \to A_6 \\ & A_5 \to \epsilon \\ & \mathbf{A_6 \to aA_5}\} \end{aligned}$$

i = 7: We reached the last iteration of the for loop. Contents of the last bracket pair number 7 are:

$$c = b$$

The seventh bracket pair encloses a terminal symbol. That means we are in switch case $c \in \Sigma \cup \{\epsilon\}$. Therefore, we add rule $A_7 \to b$ to R. Contents of R at this point are:

$$\begin{aligned} R = \{\ & S \to A_1 \\ & A_1 \to A_2 \\ & A_2 \to A_3 \\ & A_2 \to A_7 \\ & A_3 \to A_4 A_5 \\ & A_4 \to A_5 A_2 \\ & A_5 \to A_6 \\ & A_5 \to \epsilon \\ & A_6 \to aA_5 \\ & \mathbf{A_7 \to b}\} \end{aligned}$$

We have shown how does Algorithm 2 work step by step. Particularly, we have shown that for extended regular expression

$$E' = (((((a(?4))|\epsilon)(?1))(?4))|(b)),$$

it produces following context-free grammar:

$$\begin{aligned} G = \{&\{S, A_1, A_2, A_3, A_4, A_5, A_6, A_7\}, \\ &\{a, b\}, \\ &\{S \to A_1, \\ &\ A_1 \to A_2, \\ &\ A_2 \to A_3, \\ &\ A_2 \to A_7, \\ &\ A_3 \to A_4 A_5, \\ &\ A_4 \to A_5 A_2, \\ &\ A_5 \to A_6, \\ &\ A_5 \to \epsilon, \\ &\ A_6 \to a A_5, \\ &\ A_7 \to b\}, \\ &\ S\} \end{aligned}$$

▶ **Example 4.4.** Consider following extended regular expression:

$$E' = ((a((?1)b))|\epsilon)$$

The language matched by E' is context-free and is equal to $L(E) = \{a^n b^n : n \in \mathbb{N}_0\}$. If we input the expression E' into Algorithm 2, we get the following grammar as an output:

$$\begin{aligned} G = \{&\{S, A_1, A_2, A_3, A_4\}, \\ &\{a, b\}, \\ &\{S \to A_1, \\ &\ A_1 \to A_2, \\ &\ A_2 \to A_3, \\ &\ A_2 \to \epsilon, \\ &\ A_3 \to a A_4, \\ &\ A_4 \to A_2 b\}, \\ &\ S\} \end{aligned}$$

Let us now examine what language does G generate.

$$
\begin{aligned}
L(G) &= L(S) \\
&= L(A_1) \\
&= L(A_2) \\
&= L(A_3) && \cup L(\epsilon) \\
&= L(a)L(A_4) && \cup \{\epsilon\} \\
&= \{a\}L(A_2)L(b) && \cup \{\epsilon\} \\
&= \{a\}(L(A_3) \cup L(\epsilon))\{b\} && \cup \{\epsilon\} \\
&= \{a\}L(A_3)\{b\} && \cup \{a\}\{\epsilon\}\{b\} && \cup \{\epsilon\} \\
&= \{a\}L(a)L(A_4)\{b\} && \cup \{ab\} && \cup \{\epsilon\} \\
&= \{a\}\{a\}L(A_2)\{b\}\{b\} && \cup \{ab\} && \cup \{\epsilon\} \\
&= \{aa\}L(A_2)\{bb\} && \cup \{ab\} && \cup \{\epsilon\} \\
&= \{aaa\}L(A_2)\{bbb\} && \cup \{aabb\} && \cup \{ab\} && \cup \{\epsilon\} \\
&\vdots \\
&= \{a^n b^n : n \in \mathbb{N}_0\}
\end{aligned}
$$

Clearly, in the presented case, language matched by the input expression is equal to the language matched by the output grammar of Algorithm 2.

The first example was very detailed and aimed to demonstrate how Algorithm 2 works step by step. In the subsequent example, we assumed that the construction of the output grammar by Algorithm 2 is already evident. Instead, we focused on the language generated by the output grammar. We have shown that, at least in this simple case, the language matched by the input regular expression is equal to the language generated by the output grammar.

Correctness

We showed an example of a successful transformation of expression to grammar. Succesful in this case means that the language matched by input expression is equal to the language generated by the output grammar. However, we need to show that such transformation using Algorithm 2 will be successful for any given extended regular expression as an input.

▶ **Theorem 4.3** (Equivalence of generated languages)**.** *Set of words matched by extended regular expression E and set of words generated by context-free grammar G constructed from expression E by Algorithm 2 are equal.*

Proof. Consider an extended regular expression E' with features from Definition 4.2 and strictly all brackets. Create a new extended regular expression E by adding a bracket pair around E', that is $E = (E')$. Then consider a context-free grammar $G = \{N, \Sigma, R, S\}$ created from E by Algorithm 2. Let L_E be the language matched by E' and L_G the language generated by G.

During the construction of the output grammar G, we create a non-terminal symbol for each bracket pair in the expression E. Corresponding bracket pairs and non-terminal symbol share the same number (ith bracket pair corresponds to non-terminal symbol A_i).

We now inductively show that language matched by subexpression enclosed in ith bracket pair is the same as the language generated by non-terminal symbol A_i. We use rules in Definition 2.5 to obtain language matched by some (sub)expression and rules in Definition 4.1 to obtain language matched by some word $\in (N \cup \Sigma)^*$.

The base cases are:

- the ith bracket pair is equal to (ϵ):
  - **matched language**: $L(\epsilon) = \{\epsilon\}$
  - **corresponding rule**: $A_i \to \epsilon$
  - **generated language**: $L(A_i) = \{\epsilon\}$
- the ith bracket pair is equal to (x), where $x \in \Sigma$:
  - **matched language**: $L(x) = \{x\}$
  - **corresponding rule**: $A_i \to x$
  - **generated language**: $L(A_i) = \{x\}$

We will now assume that the language of each subexpression enclosed in ith bracket pair is equal to the language generated by the non-terminal symbol A_i. That is:

$$L({}_i(r)_i) = L(A_i),$$

where $a \in \Sigma \cup \{\epsilon\}$.

We now follow with the inductive steps. They are:

- the ith bracket pair is equal to (rs):
  - **matched language**: $L(rs) = L(r)L(s)$
  - **corresponding rule**: $A_i \to \alpha\beta$
    where $\alpha, \beta \in \Sigma \cup \{\epsilon\} \cup N \backslash \{S\}$
  - **generated language**: $L(A_i) = L(\alpha)L(\beta)$
    - if α (or β) is a terminal symbol or ϵ, then it is equal to $L(\alpha) = L(r)$ (or $L(\beta) = L(s)$)
    - else α (or β) is equal to some A_j corresponding to r (or s). But in this case, we know that $L(\alpha) = L(A_j) = L(r)$ (or $L(\beta) = L(A_j) = L(s)$)
  - **generated language after adjustment**: $L(A_i) = L(r)L(s)$
- the ith bracket pair is equal to $(r|s)$:
  - **matched language**: $L(r|s) = L(r) \cup L(s)$
  - **corresponding rules**: $A_i \to \alpha$ and $A_i \to \beta$
    where $\alpha, \beta \in \Sigma \cup \{\epsilon\} \cup N \backslash \{S\}$
  - **generated language**: $L(A_i) = L(\alpha) \cup L(\beta)$
    - if α (or β) is a terminal symbol or ϵ, then it is equal to $L(\alpha) = L(r)$ (or $L(\beta) = L(s)$)
    - else α (or β) is equal to some A_j corresponding to r (or s). But in this case, we know that $L(\alpha) = L(A_j) = L(r)$ (or $L(\beta) = L(A_j) = L(s)$)
  - **generated language after adjustment**: $L(A_i) = L(r) \cup L(s)$

- the ith bracket pair is equal to $((?j))$:
  - **matched language**: $L((?j)) = L((_j r)_j)$
  - **corresponding rule**: $A_i \to A_j$
  - **generated language**: $L(A_i) = L(A_j)$
    - We know that $L(A_j) = L((_j r)_j)$ therefore we can write $L(A_i) = L(A_j) = L((_j r)_j)$
  - **generated language after adjustment**: $L(A_i) = L((_j r)_j)$
- the ith bracket pair is equal to $((r))$:
  - **matched language**: $L((r)) = L(r)$
  - **corresponding rule**: $A_i \to A_{i+1}$
  - **generated language**: $L(A_i) = L(A_{i+1})$
    - We know that (r) is enclosed by ith bracket pair, so r is enclosed by $(i+1)$th bracket pair. Thus, we know that $L(A_{i+1}) = L(r)$.
  - **generated language after adjustment**: $L(A_i) = L(r)$

We have shown that the language matched by any subexpression enclosed in a bracket pair is equal to the language generated by the corresponding non-terminal symbol in our grammar G. Specifically, A_i generates the same language as the subexpression inside ith bracket pair matches.

Therefore we can point out this particular case:

$$L(A_1) = L((_1 r)_1). \tag{1}$$

Recall that in the beginning, we created E by adding a bracket pair around the input expression E'. This means that the subexpression inside bracket pair number 1 is exactly E' and therefore:

$$L(E) = L((_1 r)_1). \tag{2}$$

Let us now take a look at the language generated by grammar G. According to Definition 4.1, it is:

$$L(G) = L(S). \tag{3}$$

Since there is only one rule with S on the left-hand side $S \to A_1$, we can easily see that:

$$L(S) = L(A_1). \tag{4}$$

Putting the equalities together, we get:

$$L_E = L(E) \overset{(2)}{=} L((_1 r)_1) \overset{(1)}{=} L(A_1) \overset{(4)}{=} L(S) \overset{(4)}{=} L(G) = L_G$$

and therefore we can finally write:

$$L_E = L_G$$

◀

Results

With the results achieved in this chapter, we can finally present the main contribution of the thesis. Using Theorem 4.1 and Theorem 4.3, we can formulate and, most importantly, prove the main result, Theorem 4.4.

▶ **Theorem 4.4** (Main theorem)**.** *Language L is context-free if and only if L is matched by some extended regular expression E.*

Proof. According to Theorem 4.1, we know that for any context-free grammar generating language L, there is an extended regular expression matching language L exactly.

$$\exists \text{ CFG generating } L \Rightarrow \exists \text{ ERE matching } L$$

Similarly, according to Theorem 4.3, we know that for any extended regular expression matching language L, there is a context-free grammar generating exactly language L.

$$\exists \text{ ERE matching } L \Rightarrow \exists \text{ CFG generating } L$$

Putting the two implications together, we get:

$$\exists \text{ CFG generating } L \Leftrightarrow \exists \text{ ERE matching } L$$

Additionally, we present an elementary characteristic of context-free languages. The following well known statement:

$$L \text{ is CFL} \Leftrightarrow \exists \text{ CFG generating } L$$

Putting the two equivalences together, we get:

$$L \text{ is CFL} \Leftrightarrow \exists \text{ ERE matching } L$$

We have come to the desired result in the last equivalence. This concludes proof of Theorem 4.4 and we can indded state that:

Language L is context-free if and only if L is matched by some extended regular expression E.

◀

Chapter 5

Conclusion

In this thesis, we studied regular expressions with subpattern recursion. We researched already known results and presented some of them. Particularly, we showed results regarding backreferences, μ-regular expressions, and lookahead assertions.

We selected the PCRE library to work with for the next part because it is widely used and provides a broad range of features. We provided detailed definitions of used expressions and language matched by them. We then studied the expressive power. We came up with an algorithm for transforming, both to and from context-free grammar. Additionally, we showed that iteration is replaceable by particular usage of subpattern recursion. As the main contribution, we have shown that regular expressions with subpattern recursion describe exactly the context-free languages.

Bibliography

[1] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003. doi:10.1142/S012905410300214X.

[2] Bart Gruppen. From μ-regular expressions to context-free grammars and back. Bachelor's thesis, Radboud University, 2018.

[3] Hans Leiß. Towards kleene algebra with recursion. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, and Michael M. Richter, editors, *Computer Science Logic, 5th Workshop, CSL '91, Berne, Switzerland, October 7-11, 1991, Proceedings*, volume 626 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 1991. doi:10.1007/BFb0023771.

[4] University of Cambridge. Specification of the regular expressions supported by pcre2, the html documentation for pcre2. http://www.pcre.org/current/doc/html/pcre2pattern.html, 2020. Accessed: 2021-03-17.

[5] Nikita Popov. The true power of regular expressions. https://nikic.github.io/2012/06/15/The-true-power-of-regular-expressions.html, 2012. Accessed: 2021-03-08.

[6] Peter Thiemann. Partial derivatives for context-free languages – from µ-regular expressions to pushdown automata. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures – 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 248–264, 2017. doi:10.1007/978-3-662-54458-7_15.

Contents of Enclosed CD

```
readme.txt ..... contents description
src ..... the directory of source codes
    thesis ..... the directory of LaTeX source codes of the thesis
text ..... the thesis text directory
    thesis.pdf ..... the thesis text in PDF format
```