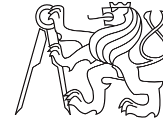
Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science



Master's thesis

Regular tree expression and its derivatives

Bc. Jan Košař

Supervisor: Ing. Radomír Polách

4th May 2015

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 accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 4th May 2015 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Jan Košař. 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

Košař, Jan. *Regular tree expression and its derivatives.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Práce představuje novou metodu pro rozpoznávání stromových (nebo i bezkontextových) jazyků za použití derivací regulárních stromových výrazů. Tato metoda je zobecněním převodu regulárních výrazů na konečný automat pomocí derivací. Práce dále rozebírá další možnosti parsování stromových a obecně bezkontextových jazyků a především srovnává představenou metodu derivací stromových regulárních výrazů s LR parsery.

Klíčová slova parsery, stromové jazyky, bezkontextové jazyky, regulární výrazy, regulární stromové výrazy

Abstract

Thesis proposes a new method for recognition of tree (or generally context-free) languages using derivatives of regular tree expression. This method is a generalization of conversion of a regular expression to a finite state automaton using derivatives. The thesis also discusses other options of parsing tree and generally context-free languages and mainly compares introduced method of derivatives of regular tree expressions with LR parsers.

Keywords parsers, tree languages, context-free languages, regular expressions, regular tree expressions

Contents

Introduction 1
Thesis contributions 1

1 Basic notions 3
1.1 Language 3
1.2 Grammar 3
1.3 Language classes 4
1.4 Language parsing 6
1.5 Tree 7
1.6 Bar notation 8
1.7 Regular expressions 9
1.8 Partial derivatives of regular string expressions 13
1.9 Construction of parser from a regular string expression 15
1.10 LR parser 16

2 Derivatives of regular tree expressions 21
2.1 First version: rules to derivate substitution and tree iteration . 22
2.2 Second version: derivatives generating pushdown automaton . . 23
2.3 Third version: Extension solving nondeterministic behavior . . 30

3 Comparison of methods for parsing tree languages 35
3.1 Similarities between method of derivatives and LR parsers . . . 35
3.2 Grammar classes and parsers 41
3.3 Tree languages 43

Conclusion 45
Method of derivatives 45
Similarity with LR parser 45
Further research 46

Bibliography 49

A Contents of enclosed CD 51

List of Figures

1.1 Example of a grammar with 3 rules 4
1.2 Examples of how a word can be generated by the grammar from figure 1.1 4
1.3 Chomsky hierarchy of formal languages 5
1.4 Language hierarchy relations 5
1.5 Hierarchy of LR and LL languages 6
1.6 Example of a graph 7
1.7 Example of a (rooted) tree 7
1.8 Prefix and postfix notations example 8
1.9 Bar notation example 8
1.10 Substitution symbol in bar notation 9
1.11 Example grammar and a respective graph of LR item sets 17
1.12 Example of construction of item set closure 17
1.13 LR parser from figure 1.11 parsing an input 19

2.1 Notions for substitution and tree iteration derivatives 22
2.2 Example of infinite automaton created by first version of algorithm 23
2.3 Two pushdown automaton states and a transition 23
2.4 Grouping of stack symbols – derivatives of $(A\Box B)^{*,\Box} \cdot_{\Box} C$ 25
2.6 Longer stack symbol – derivation of $(A\Box BCDEF)^{*,\Box} \cdot_{\Box} X$ 26
2.5 Grouping of different stack symbols – derivatives of $(A\Box B{+}C\Box D)^{*,\Box} \cdot_{\Box} E$ 27
2.7 Longer stack symbol – derivatives of $(AB\Box C(D+E))^{*,\Box} \cdot_{\Box} F$ . . . 28
2.8 Example of derivation of expression with ambiguous stack operations 31
2.9 Example of derivation of stack symbol containing ε 32

3.1 First example – derivatives of $(A\Box B)^{*,\Box} \cdot_{\Box} C$ 36
3.2 First example – LR parser for $(A\Box B)^{*,\Box} \cdot_{\Box} C$ 36
3.3 Second example – derivatives of $AB\Box CD\Box EF \cdot_{\Box} XY$ 37
3.4 Second example – LR parser for $AB\Box CD\Box EF \cdot_{\Box} XY$ 38

3.5 Third example – derivatives of $(AB\square CD)^{*,\square} \cdot_\square XY$ 38
3.6 Third example – LR parser for $(AB\square CD)^{*,\square} \cdot_\square XY$ 39
3.7 Third example – LR parser for $(A\square B + C\square D)^{*,\square}$ (equivalent regular expression derivation is in figure 2.5) 39
3.8 Derivatives of $(A\square_a B\square_b C)^{*,\square_a} \cdot_{\square_a} E \cdot_{\square_b} D$ 40
3.9 LR parser for $(A\square_a B\square_b C)^{*,\square_a} \cdot_{\square_a} E \cdot_{\square_b} D$ 40
3.10 Hierarchy of LR and LL languages 41
3.11 Hierarchy of languages parsable using regular tree expression derivatives 42

Introduction

This thesis intends to broaden knowledge of computer science in domain of formal language parsers.

Chomsky hierarchy states four basic types of formal languages: *regular languages*, *context-free languages*, *context-sensitive languages* and *recursively enumerable languages*. This thesis focuses only on *regular languages* and *context-free languages*. We further consider *tree languages* which are subset of *context-free languages*. Analysis of *tree languages* will be done later in this thesis.

Parsing of regular languages is quite explored field of study today. Creation of finite-state machine from grammar of a regular language is straightforward – typically done using regular expressions, or through construction of nondeterministic finite state automaton and its determinization.

Parsing of context-free languages is slightly more interesting. There are several subclasses inside the class of grammars of *context-free languages* and there are several methods for parsing these subclasses. Each method is suitable for parsing of different language subclass.

Thesis contributions

Derivation of regular tree expressions

First contribution of this work to mentioned theories will be a proposition of a new method for parsing of context-free languages – using derivatives of regular tree expressions. This method is based on regular expressions for regular languages, but is generalized so it can be used for parsing of context-free languages with emphasis on tree languages.

Simple version of this method is only suitable for specific subset of context-free languages, however its extension that provide it with the ability to parse all context-free languages will be proposed too.

This will be broadly explained in chapter 2.

Comparison of methods for parsing tree languages

To better understand how tree languages and their derivatives work, it would be appropriate to compare the language to other languages and derivation method for their parsing with other parsing methods.

Pushdown automaton created using method of derivatives of regular tree expressions and LR parsing will be compared.

This issue will be covered in chapter 3.

Chapter 1

Basic notions

In this chapter all basic notions will be properly defined. One does not need to read it too thoroughly since most mentioned notions are generally known as part of formal language parsing theory.

1.1 Language

A *language* is a set of *words*. That means, that for a given *language*, there is a given set of *words*, that can be formed within this language. A *word* is a set of *symbols* from an *alphabet*.

The notion *word* is slightly misleading here. It can be for example a whole source code which is given as input to a compiler. Or it can be whole text that is passed to a regular expression matcher.

Particular *language* is not usually specified as a listing of valid *words*. *Language* is typically defined by grammar rules, which describe, how the words are formed from basic components into whole words. It can also be specified by a regular expression.

As mentioned above, each language has an *alphabet*. An *alphabet* is a set of *symbols* from which the words of a language may be formed. The alphabet is usually denoted with upper-case sigma: Σ.

While *alphabets* usually have finite number of symbols, *languages* can have infinite number of valid *words*.

1.2 Grammar

Grammar is a set of rules that together define, how the words in a language are composed. Each rule consists of left hand side and right hand side.

Both left and right hand sides are strings of symbols from sets Σ and N. Σ contains so called terminal symbols – symbols that will make up the final word – it is the *alphabet* of the target language. N contains nonterminal symbols –

used to construct the string. Nonterminal symbols are usually denoted with upper-case letters while terminal symbols are usually lower-case letters.

Left hand side of rules in grammars of context-free languages consist only from one nonterminal symbol (symbol from set N).

$$\begin{gathered}\mathrm{S} \rightarrow \mathrm{E} \\ \mathrm{E} \rightarrow \mathrm{a\ E\ b} \\ \mathrm{E} \rightarrow \mathrm{c}\end{gathered}$$

Figure 1.1: Example of a grammar with 3 rules

$$\begin{gathered}S => E => c \\ S => E => aEb => acb \\ S => E => aEb => aaEbb => aaaEbbb => aaacbbb\end{gathered}$$

Figure 1.2: Examples of how a word can be generated by the grammar from figure 1.1

Grammar describes how the resulting word is constructed. Initially the word only contains one nonterminal symbol S. Grammar rules are then applied to the word repeatedly until the word does not contain any nonterminal symbols.

Applying the grammar rule to the word means, that a single occurrence of left hand side of the rule in the word is replaced with right hand side of that rule. Rule can not be applied if word does not contain the left hand side.

1.3 Language classes

1.3.1 Chomsky hierarchy

Formal languages can be assigned to different language classes depending on their complexity and on the way they can be parsed.

Basic hierarchy of languages is so called Chomsky hierarchy. It assigns languages into 4 classes. These 4 classes relate to grammar form and the type of automaton that recognizes it.

1.3.2 Parsing of context-free languages

Because tree languages are subset of context-free languages, we should look closer on classification of context-free languages. Although all context-free languages can be easily parsed using nondeterministic pushdown automaton, we are actually more interested in construction of deterministic automatons.

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A\beta \rightarrow \alpha\gamma\beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ $A \rightarrow aB$

Figure 1.3: Chomsky hierarchy of formal languages

Regular $\subset$ *Context-free* $\subset$ *Context-sensitive* $\subset$ *Recursively enumerable*

Figure 1.4: Language hierarchy relations

There are various methods for construction of deterministic pushdown automatons from language grammars and these methods do not usually work for every context-free language.

Some of the methods are *LL* (top-down) parser, *LR* (bottom-up) parser and its version (*SLR*, *LR(n)*, *SLR*, *LALR*). Context-free languages and their grammars can be separated in subclasses (LL languages, LR languages, SLR languages etc.) depending on which method is able to turn them into pushdown automaton.

Figure 1.5 shows some grammar types and explains relations between these types.

1.3.3 Regular tree expressions

Regular tree expressions can express languages from all classes mentioned in 1.3.2. Tree languages have nevertheless some limitations. For example, there is no left or right recursion in grammars of tree languages. Wider explanation of attributes of regular tree expressions will be discussed in section 3.3.

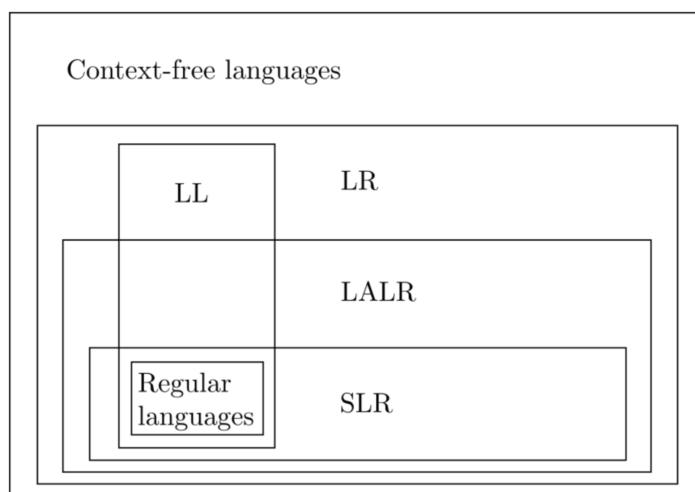


Figure 1.5: Hierarchy of LR and LL languages

1.4 Language parsing

To make computer understand input language – for example to compile program source code or to perform search in text using regular expression – we need to create *parser* for given language.

Typical way to do it is to first take the language specifications and transform it into an automaton.

Most typical way to define particular language is using language *grammar* (1.2). There also exist other methods to specify a language, including regular expressions. Regular expressions (both regular string expressions and regular tree expressions) can be transformed into language grammar and vice versa.

Different types of automatons will be created depending on the input language, grammar type, and parser creation method. This thesis takes into account only *pushdown automatons* and *finite state automatons*.

Regular string expressions can be parsed using only *finite state automatons*, but to parse languages defined with *regular tree expressions* we will need to use *pushdown automatons*.

There are plenty of methods of parsing regular languages. They typically belong to two categories: regular expression derivation and automaton composition. Derivation of regular expressions is explained later in the text (section 1.8).

The methods of automaton composition are usually used to search patterns in text. These methods typically build automatons accepting some patterns (some text with possible don't-care symbols) and then join these automatons together. Sometimes, determinization of the resulting automaton is required.

1.5 Tree

A *tree* is a special case of a *graph*. *Graph* is a data structure that consists of *nodes* and *edges*.

Node is a field that has typically a label and can contain various data. Two *nodes* can by connected together with an *edge*. Single *node* can be connected with multiple other *nodes* using respective number of *edges*. *Nodes* and *edges* can therefore create complex structure of relations between *nodes* which looks like a network.

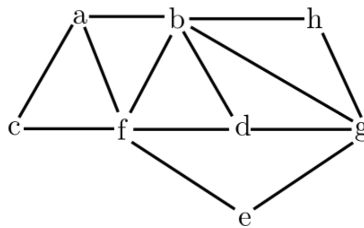


Figure 1.6: Example of a graph

As mentioned above, a *tree* is a special case of a *graph*. *Tree* is a *graph* that does not contain any cycles. It means that from each *node* in the *tree*, there is only one path to every other *node*.

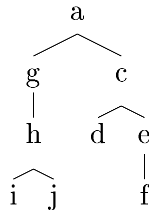


Figure 1.7: Example of a (rooted) tree

When we talk about a *tree*, we usually mean (as in this thesis) a *rooted tree*. A *rooted tree* is a *tree*, where one node is marked as *root node*. Any *tree* can be turned into *rooted tree* by marking any of its nodes as *root node*. *Root* is something like main node of the *tree*, direct neighbour of the *root* are its children which have their own children and so on.

1.6 Bar notation

The design of language grammars suggests and methods of converting grammars into parsers expects, that a *word* is a string. In our case of tree languages we want to *encode* the trees into strings.

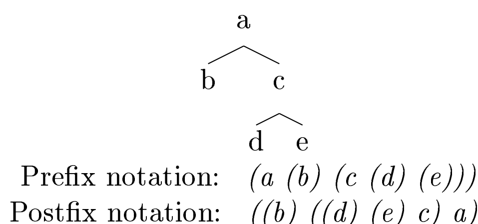


Figure 1.8: Prefix and postfix notations example

Usual method to serialize trees is to use *prefix*, *infix*, or *postfix* notations. Generally, nodes of a tree can have arbitrary number of children, therefore we can not use *infix* notation, we also need to use explicit parentheses in *prefix* or *postfix* notations.

We will define a specific prefix and postfix notation – each node will have its own pair of parentheses – so nodes without children will have its own parentheses, for example *(a)*.

When we look on the examples of our prefix and postfix notations, we can observe an interesting fact – each letter representing a node has exactly one left parenthesis on the left in prefix notation and right parenthesis on the right in the postfix notation. There is the same count of left parentheses (or right parentheses respectively) as nodes.

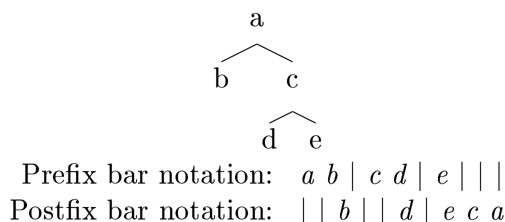


Figure 1.9: Bar notation example

So we can drop the usage of symbol '*(*' in prefix notation and symbol '*)*' in postfix notation. The second parenthesis will be replaced with a bar symbol: '|'. This allows us to use parentheses for other purposes in regular expressions.

Symbols downarrow ($\downarrow$) for prefix notation and uparrow ($\uparrow$) for postfix notation are sometimes used instead of the bar symbol ($|$).

In this thesis, we will use postfix notation and bar symbol instead of right parenthesis. Everything should be similar if prefix notation was to be used.

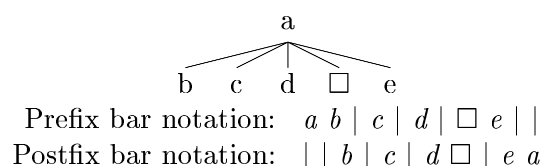
1.6.1 Substitution symbol

The substitution symbols $\square_i$ (introduced in section 1.7.2) only occur in trees as leafs. By saying, that a node is a leaf, we mean that it has no children.

There is always a '|' right before (in postfix notation) or right after (in prefix notation) every leaf.

The substitution symbol marks position, where another tree can be placed. Because this another tree already contains its '|' symbol, we should not write '|' before (after) the substitution symbol.

Figure 1.10 illustrates this. Note that in prefix notation, there is '|' after each child of root 'a' except for the substitution symbol $\square$. Equivalently in postfix notation.



Prefix bar notation: $a\ b \mid c \mid d \mid \square\ e \mid \mid$

Postfix bar notation: $\mid \mid b \mid c \mid d\ \square \mid e\ a$

Figure 1.10: Substitution symbol in bar notation

1.7 Regular expressions

A regular expression is a string that is aside from language grammar another way to specify a language. Regular expressions are basically strings consisting of symbols of language alphabet connected with binary operators. These operators specify, how the resulting word can be constructed from given symbols.

Basic regular expressions contain only 3 basic operations These 3 operators allow regular expressions to recognize regular languages (alternation, concatenation, iteration). Regular expressions are often extended with more operations to provide better user experience or to extend set of recognizable languages.

Regular *tree* expressions use trees instead of strings as operands and use 3 different operators (alternation, substitution, tree iteration). In this section, we consider operands to be trees expressed as strings using bar notation (section 1.6). This assumption allows us to consider regular tree expressions to be extension of regular string expressions – to be string expressions with 5 operators (alternation, concatenation, iteration, substitution, tree iteration). These 5 operators allow us to recognize all context-free languages.

1.7.1 Regular string expressions

- **alternation** – $A + B$

  Alternation denotes something like *OR* operator. To accept the word, it has to be described either by regular expression A or regular expression B.

  Example: $abc + cd$ – Resulting language contains words abc and cd, so the parser should accept input abc or input cd.

- **concatenation** – $A \cdot B$ or AB

  Concatenation operation concatenates two regular expressions A and B. First regular expression A have to be accepted and then regular expression B have to be accepted.

  Example: abc – Resulting language contains only word abc. Parser needs to read a, then b and then c.

- **iteration** – A^*

  Iteration of regular expression A means that there should be arbitrary number (including 0) of occurrences of A concatenated together. This operation typically creates languages with infinite number of words.

  Example: a^* – Resulting language contains words consisting of arbitrary number of symbols a, for example a, aa, aaa, $aaaa$, etc.

Parts of regular expressions can be enclosed into parentheses ('(', ')') to denote order of operations.

Example: $abc(d+e)*$ – Resulting language contains words beginning with string abc followed by string of arbitrary length consisting of symbols d and e, for example abc, $abce$, $abcd$, $abceededde$, $abceeeed$, $abcdddd$, etc.

Basic notions:

- Σ – alphabet of a language
- $E_i \in (\Sigma \cup \{\varepsilon, \emptyset, +, \cdot, ^*\})^*$ – some regular expression
- $w \in \Sigma^*$ – word is a string of alphabet symbols
- $L_i \subset \Sigma^*$ – language is a *set* of words
- $l(E) = L$ – L is a language defined by regular expression E

Operations:

- $l(E_1 \cdot E_2) = l(E_1) \cdot l(E_2)$
- $l(E_1) \cdot l(E_2) = L_3 \mid a \in E_1 \wedge b \in E_2 \implies ab \in L_3$
- $l(E_1 + E_2) = l(E_1) \cup l(E_2)$
- $l(E^*) = l(\varepsilon + E \cdot E^*)$

Basic relations:

- $l(\emptyset) = \emptyset$
- $l(\varepsilon) = \{\varepsilon\}$
- $l(a) = \{a\} \mid a \in \Sigma$
- $l(E_1 \cdot \varepsilon) = l(E_1)$
- $l(E_1 + \emptyset) = l(E_1)$
- $l(E_1 \cdot \emptyset) = \emptyset$

The regular expressions can contain a special symbol ε which means *no symbol* (string of length 0). This basically means that regular expression $\varepsilon aa\varepsilon a$ describes the same language as aaa. Symbol ε can be really useful sometimes.

Another special symbol used in regular expressions is $\emptyset$. This symbol denotes empty language $\{\}$ and is usually used as result of an invalid operation (mainly in derivatives when we try to derivate expression in respect to a symbol that this expression can not be derived in respect to).

1.7.2 Tree operations

Regular tree expressions basically extend string expressions with two operations – substitution and tree iteration. Symbols from new set K now can also be part of regular tree expressions. These symbols are denoted $\square_i$ and they indicate location for substitution or tree iteration operations to take place.

- **substitution** – $A \cdot_{\square_1} B$

  Substitution operator replaces all occurrences of symbol $\square_1$ in expression A with expression B.

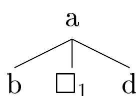
- **tree iteration** – $A^{*,\square_1}$

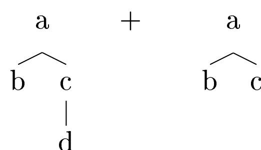
  Tree iteration denotes arbitrary number of applications of substitution operator on self. This operation typically creates language with infinite number of words. Additionally, this operation causes language described by the regular tree expression to be context-free language.

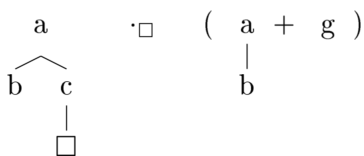
Formal definition:

- $E_i \in (\Sigma \cup K \cup \{\varepsilon, \emptyset, +, \cdot, ^*, \cdot_{\Box_i}, ^{*,\Box_i}\})^* \mid \Box_i \in K$ – regular expressions may now contain boxes, substitution operator and tree iteration operator
- $l(E_1 \cdot_{\Box_1} E_2) = l(E_3)$ where E_3 is E_1 with all occurrences of symbol $\Box_1$ replaced with expression E_2
- $l(E^{*,\Box_1}) = l(\Box_1 + E \cdot_{\Box_1} E^{*,\Box_1})$

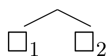
1.7.3 Examples of trees and regular tree expressions

- empty tree: ε
- only substitution symbol as root: $\Box_1$
- ordinary tree: 

- alternation: 

- substitution: 

- iteration: ($)^{*,\Box}$ $\cdot_{\Box}$ b

- another iteration: ( $)^{*,\Box_1} \cdot_{\Box_1} b \cdot_{\Box_2} c$

1.8 Partial derivatives of regular string expressions

Method for derivation of regular *tree* expressions, which will be proposed in chapter 2, is based on derivatives of regular *string* expressions. In this section, the generally known method for derivation of regular string expressions and its notions will be presented.

Derivatives of regular expressions are basically partial derivatives. So when we want to derivate expression E_1, we need to say in respect to which symbol we want to derivate it.

Lets say we want to derivate expression E_1 in respect to symbol a. The symbol has to be in the alphabet of the respective language. Expression E_1 defines a language L_1 – that basically means, that we will read a sequence of input symbols (input word) and we need to decide, if this sequence belongs to language L_1. This can be done by reading first symbol (in our case symbol a) and then constructing another regular expression E_2 which will describe, which symbols can be read next. We call this operation *partial derivation* and E_2 is *partial derivative* of expression E_1 *in respect to* symbol a.

This new expression E_2 defines another language L_2 which contains all words from L_1 that are beginning with symbol a but in this new language, the initial a is skipped:

- $aX \in l(E) \implies X \in l(\frac{dE}{da})$

In general, derivative of expression E_1 in respect to a symbol from an alphabet can result in 3 types of regular expressions E_2:

- $\emptyset$:

  This result means that given expression E_1 can not be derived in respect to given symbol.

- expression E_2 for which applies $\varepsilon \in l(E_2)$:

  If resulting expression contains ε, it means that if no more symbols are in the input, then a word from given language was successfully accepted.

- other expressions :

  If the derivative is an expression that is not $\emptyset$ and does not contain ε, then more input symbols have to be read.

1.8.1 Algorithm for partial derivation of a regular string expression

There is a simple algorithm how to derivate a regular expression. For each operator (alternation, concatenation and iteration) and some basic regular expressions there is defined equation describing how it can be derived.

The equation describing derivation of concatenation operator requires knowledge, if the language of the first operand contains ε. Therefore it is required to know, how to find out, if the language of a given regular expression contains ε. The equations to find it out are included below.

Operators:

- $\frac{dE_1 + E_2}{da} = \frac{dE_1}{da} + \frac{dE_2}{da}$
- $\frac{dE_1 \cdot E_2}{da} = \frac{dE_1}{da} \cdot E_2 \qquad \mid \varepsilon \notin l(E_1)$
- $\frac{dE_1 \cdot E_2}{da} = \frac{dE_1}{da} \cdot E_2 + \frac{dE_2}{da} \qquad \mid \varepsilon \in l(E_1)$
- $\frac{dE^*}{da} = \frac{dE}{da} \cdot E^*$

Basic equations:

- $\frac{da}{da} = \varepsilon$
- $\frac{da}{db} = \emptyset \mid b \neq a$
- $\frac{da}{d\varepsilon} = a$
- $\frac{d\varepsilon}{da} = \emptyset$
- $\frac{d\emptyset}{da} = \emptyset$

Epsilon in a language

- $\varepsilon \in l(E_1 + E_2) \Leftrightarrow \varepsilon \in l(E_1) \vee \varepsilon \in l(E_2)$
- $\varepsilon \in l(E_1 \cdot E_2) \Leftrightarrow \varepsilon \in l(E_1) \wedge \varepsilon \in l(E_2)$
- $\varepsilon \in l((E_1)^*) \Leftrightarrow \top$
- $\varepsilon \in l(\varepsilon) \Leftrightarrow \top$, $\varepsilon \in l(a) \Leftrightarrow \bot$, $\varepsilon \in l(\emptyset) \Leftrightarrow \bot$

1.9 Construction of parser from a regular string expression

How to parse input string W using regular expression E? We can just read input word W symbol by symbol and continuously derivate regular expression in respect to input symbols. At the end of the input, we will check if resulting regular expression contains ε and in that case, we will say that the input string W belongs to the language defined by regular expression E. Otherwise (typically when the last derivative is $\emptyset$) we would say that the input string W does not belong to given language.

To make this method more efficient and stable (in terms of time complexity), we will create automaton doing it. Each state of the automaton represents one derivative and transitions represent derivations. If there is transition a from state A to state B, then regular expression represented by B is a derivative of regular expression represented by A in respect to input symbol a.

1.9.1 Algorithm for construction of finite state automaton using derivatives of a regular string expression

To do this, we will take given regular expression E and derivate it consecutively in respect to each input symbol for which the result is not $\emptyset$. The original regular expression E represents initial state of the finite state automaton, its derivatives represent other states of the finite state automaton. When one regular expression E_1 has derivative E_2 in respect to symbol a, then transition for symbol a from state represented by expression E_1 to state represented by expression E_2 will be added to the finite state automaton. One should then repeatedly derivate expressions representing new states, which will add another new states and so on. Considering regular languages, this should stop after some time – the finite state automaton is done.

For formal definition of the algorithm, we need to first define finite state automaton.

Finite state automaton is defined as $(\Sigma, S, s_0, \delta, F)$, where:

- Σ – input alphabet
- S – finite set of states
- s_0 – initial state
- $\delta : S \times \Sigma \rightarrow S$ – transition function
- F – set of final states

Formal definition: We are constructing finite state automaton $(\Sigma, S, s_0, \delta, F)$ from regular expression E:

- $s_0 = E$
- $E_i \in S \wedge \frac{dE_i}{dx} \neq \emptyset \wedge x \in \Sigma \implies \frac{dE_i}{dx} \in S \wedge (E_i, x, \frac{dE_i}{dx}) \in \delta$
- $E_i \in S \wedge \varepsilon \in l(E_i) \implies E_i \in F$

1.10 LR parser

LR parser is an usual method of parsing context-free languages. In chapter 3, the method of derivatives of regular tree expressions be compared with LR parsers.

There are many different versions of LR parser, mainly SLR, LALR, full LR parser etc. To fully understand, how LR parsers work, I would suggest to read something else [1]. Only basics needed to understand the comparison with method of derivatives will be covered in this thesis.

The algorithm for construction of graph of LR item sets and the way how to use this graph to do the parsing will be explained.

1.10.1 How to construct graph of LR item sets

Before we can create LR item sets, we need the language grammar (section 1.2). A LR item is a grammar rule with a dot symbol . somewhere in the right-hand side of the rule (for example $S \rightarrow a.bc$). LR item set is a set of LR items. There are transitions (directed edges) between item sets. Each transition is labeled with an input symbol with an index (for example A_1).

Each item set has kernel items and closure items.

Item set, that represents the initial state, will contain all the rules that have initial nonterminal symbol S on the left-hand side of the rule. These items will have dot on the beginning of the right-hand side of the rule. For example, if there is only one rule having initial nonterminal symbol S on the left-hand side which is $S \rightarrow abc$, then the item set representing the initial state will have only one kernel item – $[S \rightarrow .abc]$.

Each item set, including the item set representing the initial state, will have closure items, that are created from kernel items and other closure items.

Closure items

A closure item will be added to the item set if the item set contains an item, that has the dot symbol right before a nonterminal symbol (for example item $[S \rightarrow a.Ab]$). If an item set contains item with the dot symbol exactly before nonterminal symbol A, all rules from the grammar that have symbol A on

Grammar: $S \to aA$
$A \to bbBb$
$B \to c$
$B \to d$

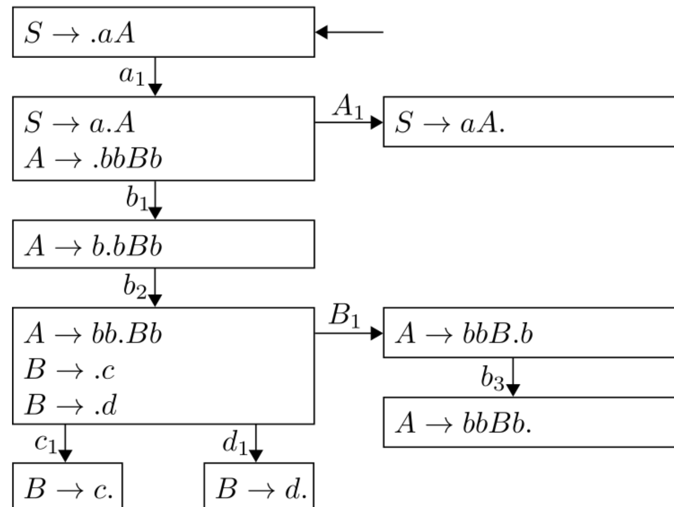


Figure 1.11: Example grammar and a respective graph of LR item sets

left-hand side of the rule will be added to the item set. The new items will have dot symbol on the beginning of the right-hand side of the rule.

Closure items should be added even when other, just created closure item contains dot symbol before a nonterminal symbol.

Grammar: $S \to Ab$
$A \to ab$
$A \to BabC$
$B \to d$
$C \to c$

Initial LR item set:

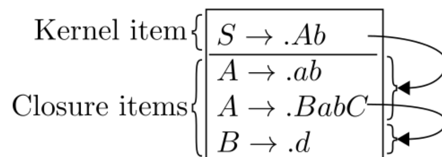


Figure 1.12: Example of construction of item set closure

1.10.2 Constructing other item sets

For each item that has the dot symbol somewhere else than at the end of the right-hand side of the rule, we will create an edge (transition) to another item set. This transition will be labeled with the symbol that is right after the dot symbol. For example if we have an item $[S \to a.Ab]$, we will create transition labeled with symbol A.

Kernel items of the new item set will be created by taking the original item and shifting the dot symbol by one symbol to the right. If we create new item set from an old item set using transition labeled A, we have to include all items from the old item set that have dot right before the symbol A.

After kernel items are created, the closure of this item set will be created.

If we would create exactly the same item set as an already existing item set, we do not create it again, but we just create the transition leading to this item set.

1.10.3 How to use graph of LR item sets to parse the input

The parser reads input symbols, stores them on the stack and then applies rules on them, which changes set of terminals and nonterminals into one terminal. The right-hand side of the rule is popped from the stack while the left-hand side (single nonterminal) is pushed to the stack. This is called bottom-up parsing.

Different type of parsing is top-down parsing, used by LL parsers. Top-down parsing expands nonterminal symbols (pops left-hand side of the rule and pushes right-hand side of the rule) and reads the terminals from input.

Nevertheless, the LR parser starts in initial state, which is denoted with symbol $\#$. The state of the automaton is given by the top symbol on the stack. So stack starts only with symbol $\#$.

The parser can do two actions: *shift* and *reduce*.

The *shift* operation reads next input symbol. The automaton changes state – from current state traverses to next state using transition that is labeled with the read input symbol. New state is put onto stack.

The *reduce* operation applies a grammar rule. First thing is to decide, what grammar rule to apply. Different versions of LR parsers use different methods to determine, which rule should be applied and whether shift or reduce operation should be preformed.

So the *reduce* operation pops one stack symbol for each symbol in the selected rule in the reverse order (the items on the right side of the right-hand side of the rule is popped first). This causes the automaton to use transitions in the opposite direction (automaton is returning to states where it was before). When the whole right-hand side of the rule is popped, the symbol on the left-hand side of the rule is pushed onto the stack in the same way the *shift* operation does it.

The automaton configuration is represented by couple: (stack, input). The # symbol denotes bottom of the stack. The input to be read is $abbcb$. State of the automaton is defined by the symbol on top of the stack.

$(\#, abbcb)$	*shift* symbol a (automaton moves to state a_1)
$\rightarrow (\#a_1, bbcb)$	*shift* symbol b
$\rightarrow (\#a_1b_1, bcb)$	*shift* symbol b
$\rightarrow (\#a_1b_1b_2, cb)$	*shift* symbol c
$\rightarrow (\#a_1b_1b_2c_1, b)$	*reduce* with rule $B \rightarrow c$
$\rightarrow (\#a_1b_1b_2B_1, b)$	*shift* symbol b
$\rightarrow (\#a_1b_1b_2B_1b_3, \varepsilon)$	*reduce* with rule $A \rightarrow bbBb$
$\rightarrow (\#a_1A_1, \varepsilon)$	*reduce* with rule $S \rightarrow aA$
$\rightarrow (\#S, \varepsilon)$	

Figure 1.13: LR parser from figure 1.11 parsing an input

Chapter **2**

Derivatives of regular tree expressions

In this chapter, method for derivation of regular tree expressions will be proposed. This method reads regular tree expression and creates deterministic pushdown automaton accepting language defined by that regular tree expression.

Three versions of this method will be presented. Each version is an extension of the previous one and is capable of processing wider set of languages than the previous version.

First version extends basic method for creation of finite state automaton from regular string expression. This version creates finite state automatons and will create automatons of infinite size when processing regular tree expression describing non-regular language.

Second version improves the first method by adding ways to create pushdown automatons. This will always create finite state automatons or pushdown automatons of finite size, but it may produce nondeterministic pushdown automatons for some context-free languages. Regular tree expressions describing tree languages will always create deterministic pushdown automaton.

The third version solves the problem with non-determinicity as it should always create pushdown automaton that is deterministic.

Generalization to context-free languages

Tree languages are languages (section 1.1) where each word is a tree (section 1.5). We can specify a tree language using a regular tree expression.

Trees can be expressed as strings using bar notation (section 1.6). In that case, the tree languages are a subset of context-free languages.

In this chapter, we will consider regular tree expressions to be capable of describing all context-free languages, not only tree languages. All statements

about regular tree expressions presented in this chapter are meant to apply to this generalized definition of regular tree expressions.

2.1 First version: rules to derivate substitution and tree iteration

Lets say we have a regular expression E and we want to derivate it. Regular expressions generally consist of alphabet symbols connected with operators.

There can be 5 operators: alternation, concatenation, iteration, substitution and tree iteration. The first tree operations – alternation, concatenation and iteration – will be derived in the same way as in regular string expressions (section 1.8).

Now we just need to write similar equations describing how to derivate substitution and tree iteration.

- $\dfrac{de_1 \cdot_{\square_1} e_2}{dx} = \dfrac{de_1}{dx} \cdot_{\square_1} e_2 + \dfrac{de_2(\frac{de_1}{d\square_1} \cdot_{\square_1} e_2)}{dx} \qquad | \; x \neq \square_1$
- $\dfrac{de_1 \cdot_{\square_1} e_2}{d\square_1} = \dfrac{de_2(\frac{de_1}{d\square_1} \cdot_{\square_1} e_2)}{d\square_1}$
- $\dfrac{de^{*,\square_1}}{dx} = \dfrac{de}{dx} \cdot_{\square_1} e^{*,\square_1} \qquad | \; x \neq \square_1$
- $\dfrac{de^{*,\square_1}}{d\square_1} = (\dfrac{de}{d\square_1})^* \cdot_{\square_1} e^{*,\square_1}$

Figure 2.1: Notions for substitution and tree iteration derivatives

Application of these equations will result in valid derivatives. So the equations are valid. There is however a problem. If we try to use this method of derivatives to construct an automaton from a regular tree expression which describes language that is not regular, it will create finite state automaton of infinite size (so it will not really be a *finite* state automaton).

It is apparent that we need to generalize the derivatives to be able to create pushdown automatons.

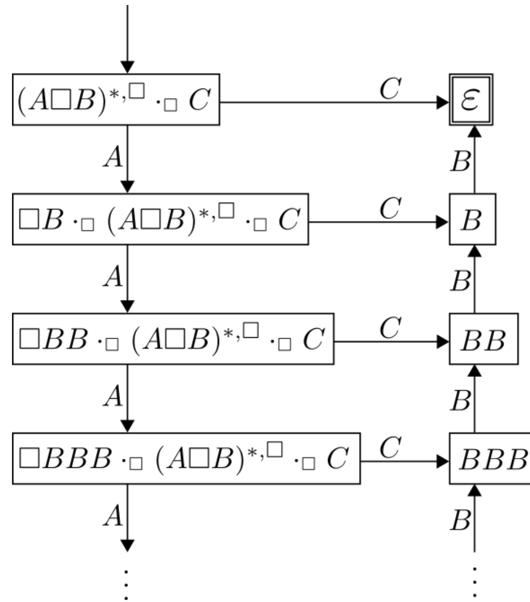
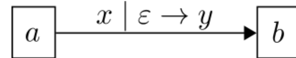


Figure 2.2: Example of infinite automaton created by first version of algorithm

2.2 Second version: derivatives generating pushdown automaton

While creating finite state automaton from a regular string expression, transition labeled 'a' leading from state E_1 to E_2 is created when E_2 is a derivative of E_1 in respect to input symbol a. Transitions in pushdown automatons have to be labeled not only with input symbol, but with triplet *(input symbol | popped symbol → pushed symbol)*.

When the transition is used by pushdown automaton, one input symbol is read from input, one stack symbol may be popped from stack and one stack symbol may be pushed to stack. If no symbol is to be pushed to stack or popped from stack, the symbol ε is used instead of an ordinary stack symbol.



This transition reads symbol x from input and puts symbol y on stack.

Figure 2.3: Two pushdown automaton states and a transition

To create pushdown automatons using derivatives, we have to derivate not only in respect to an input symbol but in respect to the triplet *(input symbol | popped symbol → pushed symbol)*. That means that derivative of an expression E in respect to $(a \mid b \to \varepsilon)$ will be different than the derivative in respect to $(a \mid \varepsilon \to \varepsilon)$ because while using the transition corresponding to the

first derivation, the automaton modified stack, but the second derivation does not modify stack.

Four types of transitions (derivatives) can be distinguished (ordered by their priority – see next subsection 2.2.1):

1. $a \mid \varepsilon \rightarrow b$ – This transition reads input symbol a from input and pushes stack symbol b on stack.

   This transition type shows up when deriving *substitution* operation.

2. $a \mid b \rightarrow \varepsilon$ – Automaton reads input symbol a from input and pops stack symbol b from stack.

   This is complement of previous transition type. Rest of the right operand to *substitution* operation is read.

3. $a \mid \varepsilon \rightarrow \varepsilon$ – This transition reads input symbol a from input and stack is not changed.

   This is equivalent to transitions in finite state automatons and is used usually by equations derivating basic operations (concatenation, alternation, iteration) as presented is section 1.8.1.

4. $\varepsilon \mid \varepsilon \rightarrow c$ – Automaton pops stack symbol c from stack without reading any input.

   This transition will be used in the third version of this algorithm. It is used only when a regular expression belonging to the stack symbol c contains ε – therefore it can be popped without reading any input.

2.2.1 Transition ambiguity – implicit transition priorities

There are some expressions that can be derived both using transitions modifying stack and transitions not modifying stack. In this situation, two problems arise:

- Automaton has to decide, which transition to use.
- Transitions that do not modify stack would create automaton of infinite size (as in figure 2.2).

Both these problems will be solved by giving implicit priorities to transitions. Each type of transition mentioned above (section 2.2) will have its priority. When an expression has for example two transitions, first of type $a \mid \varepsilon \rightarrow \varepsilon$ and second of type $a \mid \varepsilon \rightarrow b$, the second transition will be used.

Only reachable states (taking these priorities into account) should be included in resulting automaton.

Implicit transition priorities can cause another problem – automaton that does not accepts whole language described by regular expression might be

created sometimes. This is caused by automaton not knowing in some cases, if it should pop stack symbol immediately or if it should do it later.

This problem does not occur in tree languages (only in context-free languages). Version three of this algorithm solves the problem (section 2.3).

2.2.2 Stack symbols

To be able to derivate regular expressions in such way that it results in creation of pushdown automaton, we will use stack symbols as part of regular expressions apart from just symbols of input alphabet.

Each stack symbol represents a regular expression that should be read after that stack symbol is popped. Position of stack symbol in a regular expression marks position, where this stack symbol can be popped. If the stack symbol is popped, automaton will read expression associated with that stack symbol.

Notion: $\overline{\underline{E}}$

Example: $\overline{\underline{abc + ef}}$

2.2.3 Grouping of identical stack symbols

The reason, why second version of this method constructs automatons of finite size while first version constructs infinite automatons, is that we can group identical stack symbols together in such way that whenever there is a concatenation of two or more identical stack symbols, we can turn them into one. Original number of these symbols is saved in the stack anyway.

We state the equation: $\overline{\underline{E_i}}\ \overline{\underline{E_i}} \Leftrightarrow \overline{\underline{E_i}}$.

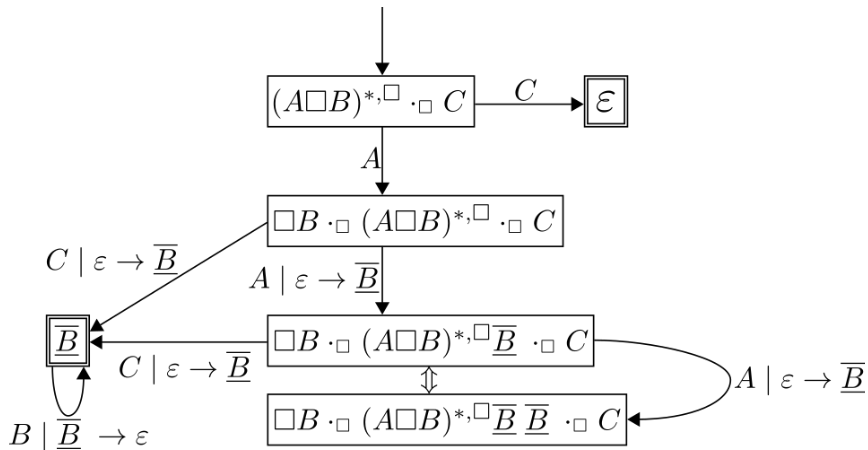


Figure 2.4: Grouping of stack symbols – derivatives of $(A\square B)^{*,\square} \cdot_\square C$

2.2.4 Grouping of different stack symbols

Some regular expression such as $(A\square_1 B + C\square_1 D)^{*,\square_1} \cdot_{\square_1} E$ need stack symbols to group even if the symbols are different. In this particular case, concatenation of multiple $\overline{\underline{B}}$ and $\overline{\underline{D}}$ symbols will be generated.

The concatenation of multiple different stack symbols is equivalent to concatenation of stack symbols such that each stack symbol is there only once. Again, stack will store information about number and order of stack symbols.

$\overline{\underline{E_1}}\ \overline{\underline{E_2}} \Leftrightarrow \overline{\underline{E_2}}\ \overline{\underline{E_1}}$

2.2.5 Stack symbols with longer associated regular expressions

So far, only simple stack symbols consisting only from one input symbol were show as examples. This example shows how a regular tree expression with more complex associated expression is derived.

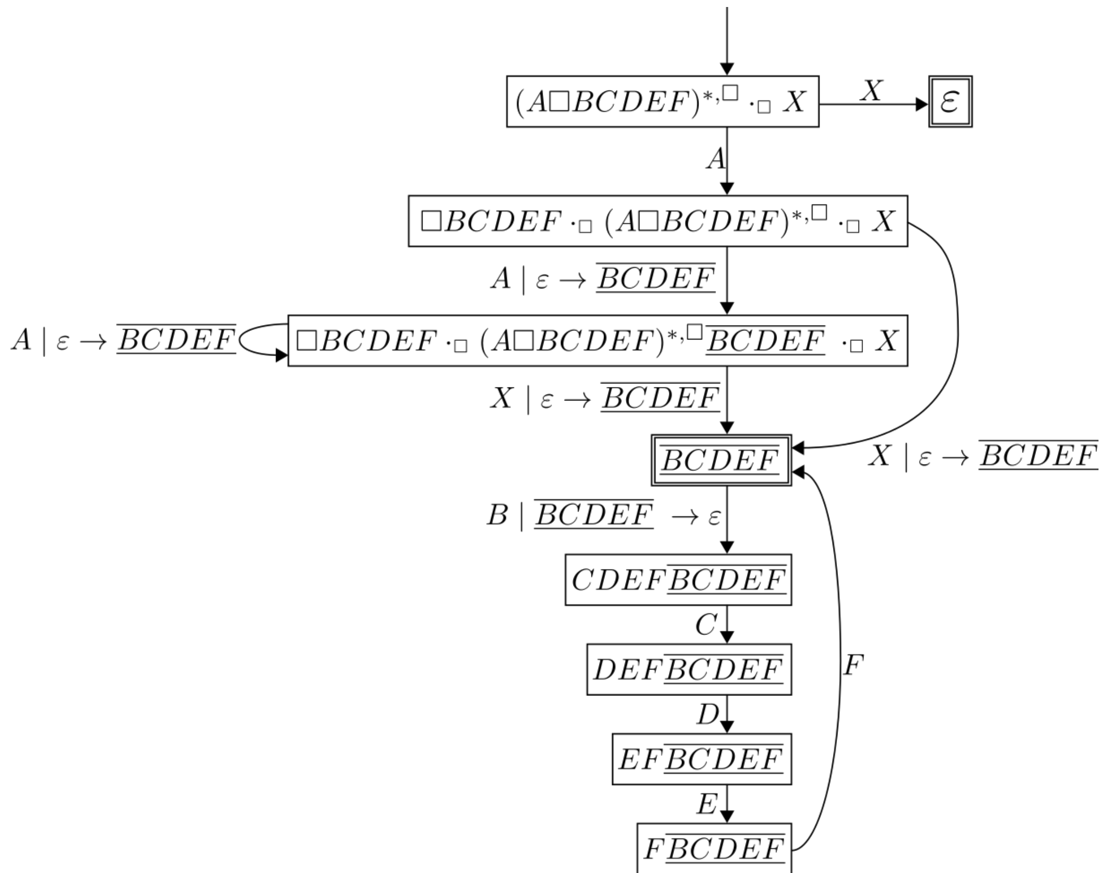


Figure 2.6: Longer stack symbol – derivation of $(A\square BCDEF)^{*,\square} \cdot_\square X$

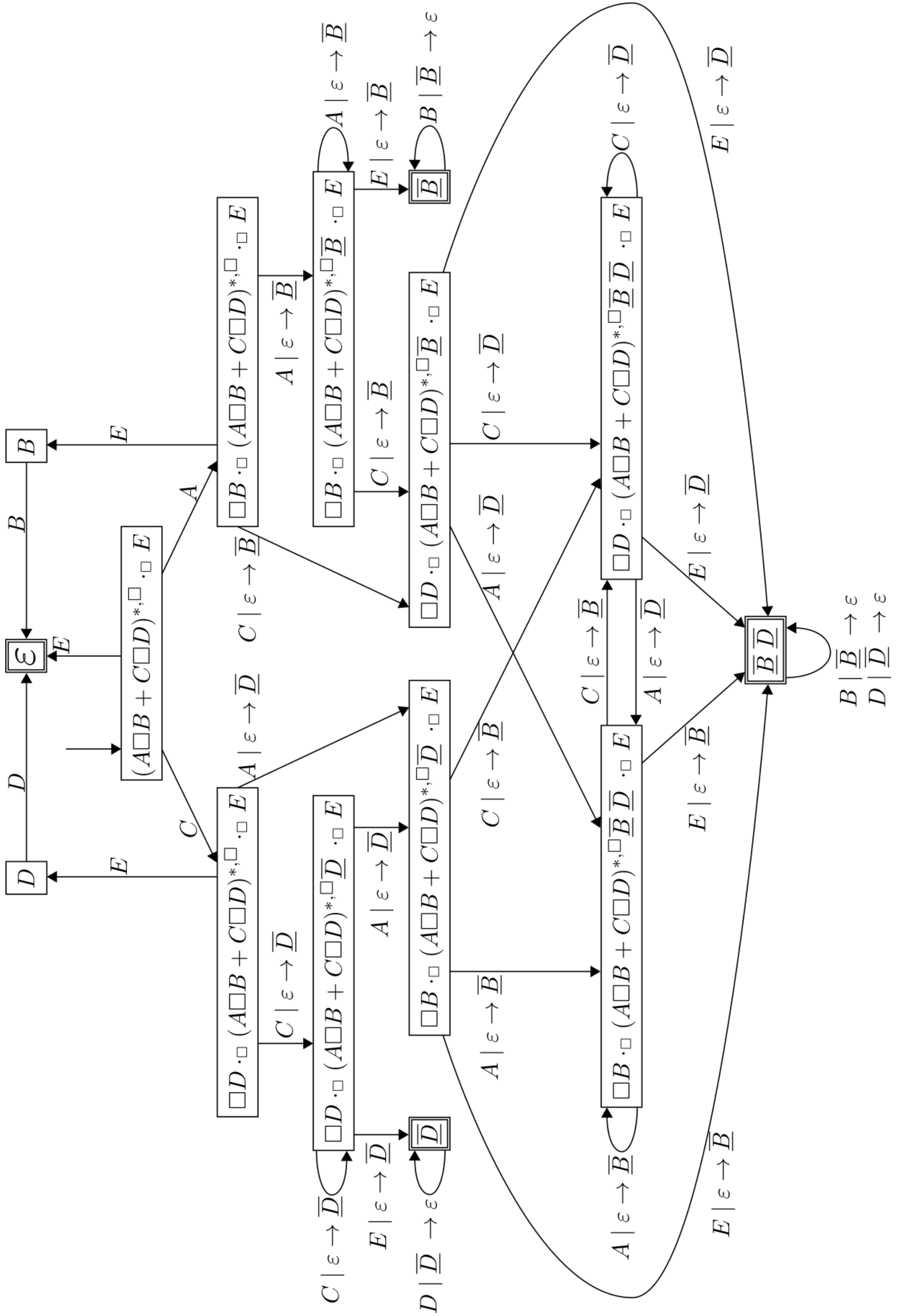


Figure 2.5: Grouping of different stack symbols – derivatives of $(A\square B + C\square D)^{*,\square} \cdot_\square E$

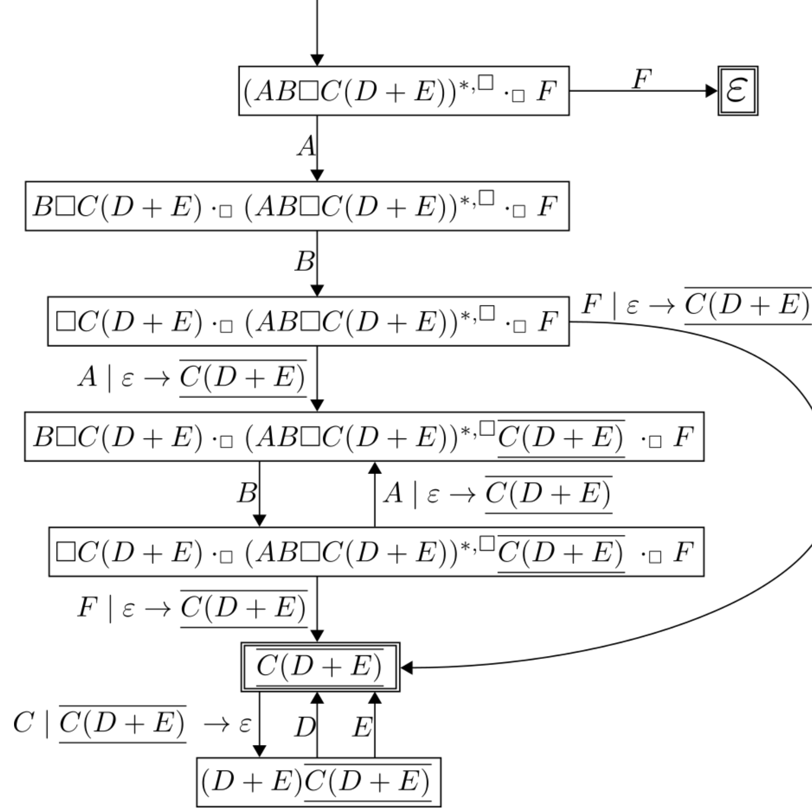


Figure 2.7: Longer stack symbol – derivatives of $(AB\square C(D+E))^{*,\square} \cdot_{\square} F$

2.2.6 Modified concatenation operator

Sometimes it is necessary to keep stack symbols in regular expression while deriving it. When there is for example regular expression $\overline{\underline{a}}\ \overline{\underline{b}}\ c$ and we derive it in respect to $b \mid \overline{\underline{b}} \to \varepsilon$ it should result in the same expression $\overline{\underline{a}}\ \overline{\underline{b}}\ c$ (the $\overline{\underline{a}}$ should remain in the new expression).

It would be hard to achieve this with the usual derivation of concatenation operator. Therefore the concatenation operator has to be redefined. The new definition is listed in following text (section 2.2.7).

The definition contains a new function denoted with letter *sigma*: σ. There are 3 similar σ functions (σ_{ε}, σ_{push}, σ_{pop}). Particular sigma function is selected according to performed stack operation (no operation, push operation or pop operation).

These functions are defined in section 2.2.7. Equations describing σ (without a subscript) apply on all three σ functions.

Input to this function is a regular expression and result is typically $\emptyset$ or ε. ε is returned if language of the regular expression contains ε and $\emptyset$ is returned otherwise.

In some cases when stack symbols are in the regular expression, something more expressive may be returned. This happens in those cases, when an expression can not be just skipped as ε. So when the expression is to be skipped as ε, the result of σ function is placed in its place.

2.2.7 Equations

Substitution:

- $$\frac{dE_1 \cdot_{\Box_1} E_2}{dx \mid \varepsilon \to \overline{\underline{\frac{dE_1}{d\Box_1} \cdot_{\Box_1} E_2}}} = \frac{dE_2 \overline{\underline{\frac{dE_1}{d\Box_1} \cdot_{\Box_1} E_2}}}{dx}$$

- $$\frac{dE_1 \cdot_{\Box_1} E_2}{dx \mid \varepsilon \to \overline{\underline{y}}} = \frac{dE_2 (\frac{dE_1}{d\Box_1 \mid \varepsilon \to \overline{\underline{y}}} \cdot_{\Box_1} E_2)}{dx}$$

- $$\frac{d\overline{\underline{E_1}}}{dx \mid \overline{\underline{E_1}} \to \varepsilon} = \frac{dE_1 \overline{\underline{E_1}}}{dx}$$

- $$\frac{d\overline{\underline{E_1}}}{db \mid \varepsilon \to \varepsilon} = \emptyset$$

- $\overline{\underline{a}}\,\overline{\underline{a}} \Leftrightarrow \overline{\underline{a}}$

- $\overline{\underline{a}}\,\overline{\underline{b}} \Leftrightarrow \overline{\underline{b}}\,\overline{\underline{a}}$

Alternation and iteration:

- $$\frac{dE_1 + E_2}{dx \mid y \to z} = \frac{dE_1}{dx \mid y \to z} + \frac{dE_2}{dx \mid y \to z}$$

- $$\frac{dE_1^*}{dx \mid y \to z} = \frac{da}{dx \mid y \to z} \cdot a^*$$

Concatenation:

- $$\frac{dE_1 \cdot E_2}{dx \mid \varepsilon \to \varepsilon} = \frac{dE_1}{dx \mid \varepsilon \to \varepsilon} \cdot E_2 + \sigma_{\varepsilon}(E_1) \cdot \frac{dE_2}{dx \mid \varepsilon \to \varepsilon}$$

- $$\frac{dE_1 \cdot E_2}{dx \mid \varepsilon \to y} = \frac{dE_1}{dx \mid \varepsilon \to y} \cdot E_2 + \sigma_{push}(E_1) \cdot \frac{dE_2}{dx \mid \varepsilon \to y}$$

- $$\frac{dE_1 \cdot E_2}{dx \mid y \to \varepsilon} = \frac{dE_1}{dx \mid y \to \varepsilon} \cdot E_2 + \sigma_{pop}(E_1) \cdot \frac{dE_2}{dx \mid y \to \varepsilon}$$

Sigma functions:

- $\sigma(x) = \emptyset$ $\quad | \; x \in \Sigma$
- $\sigma(\varepsilon) = \varepsilon$
- $\sigma(\emptyset) = \emptyset$
- $\sigma(E_1 + E_2) = \sigma(E_1) + \sigma(E_2)$
- $\sigma(E_1 \cdot E_2) = \sigma(E_1) \cdot \sigma(E_2)$
- $\sigma(\square_1) = \square_1$ $\quad | \; \square_1 \in K$
- $\sigma(E_1 \cdot_{\square_1} E_2) = \sigma(E_1) \cdot_{\square_1} \sigma E_2$
- $\sigma(E_1^{*,\square_1}) = \sigma(E_1)^{*,\square_1}$

- $\sigma_\varepsilon(\underline{\overline{E_1}}) = \varepsilon$
- $\sigma_{push}(\underline{\overline{E_1}}) = \varepsilon$
- $\sigma_{pop}(\underline{\overline{E_1}}) = \underline{\overline{E_1}}$

2.3 Third version: Extension solving nondeterministic behavior

Previous (second) version does not create working automatons for some regular tree expressions. These wrong automatons do not accept whole language described with the regular expression.

The core of the problem is that the created pushdown automaton might get into situation, when it does not know, when to pop the stack symbols. Our pushdown automatons will always pop stack symbols as soon as possible because of the transition priorities presented in section 2.2.1. This may be sometimes the wrong decision.

This problem probably does not occur with regular tree expressions that are constructed only from trees in postfix or prefix bar notation. This third version of algorithm is only required if we generalize regular tree expressions to context-free languages, where any string can be operand to a regular tree expression.

There are two basic types of regular expressions in relation to stack symbols:

- **Regular expression that begin with a stack symbol**

  They have to be derived using transition of type $a \mid b \to \varepsilon$ – given stack symbol is popped.

  For example $\underline{\overline{AB}}\, CD$ will be derived in respect to $A \mid \underline{\overline{AB}} \to \varepsilon$).

- **Regular expression that begin with an input symbol**

  These expressions will be derived using transition of type $a \mid \varepsilon \to \varepsilon$ – stack is unchanged.

We will have problem if regular expression contains alternation or ε symbols in such way that the regular expression begins with both stack symbol and input symbol.

There are two particular cases of this problem:

1. Stack symbol can be popped too soon
2. Stack symbol can be popped through ε transition

2.3.1 Stack symbol may be popped too soon

This situation arises when there is an alternation in the regular expression in such way that an input symbol can be accepted both with and without popping a stack symbol.

To solve this, we will pop the stack symbol and put there alternation where one possibility is that it should have been popped initially and the other possibility is that should be popped later. There is of course possibility, that there is not the desired stack symbol on the stack and in that case, nothing will be popped and the part of regular expression that does not need to pop the stack symbol will be used.

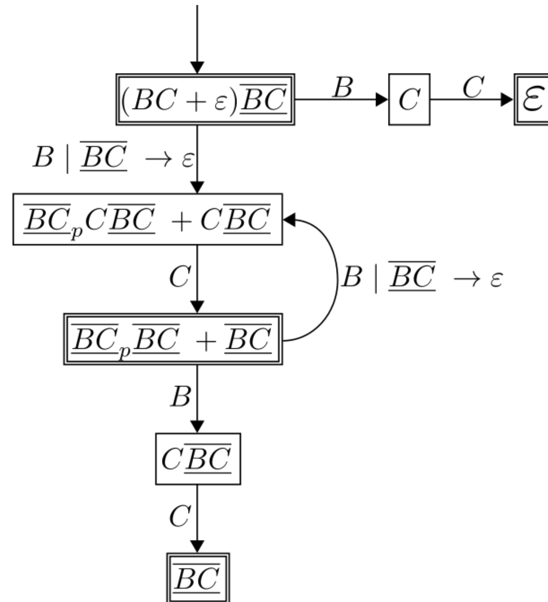


Figure 2.8: Example of derivation of expression with ambiguous stack operations

An example of such regular expression: $(BC + \varepsilon)\overline{\underline{BC}}$.

We introduce new unary operator which can be added to a regular expression: $\overline{\underline{a}}_p$. This operator denotes position in regular expression, where the stack symbol $\overline{\underline{a}}$ was popped, but it did not came into effect yet.

This operator will stay in regular expression, when using transition that does not change stack and will be removed when using transition that changes stack.

2.3.2 Stack symbol can be popped through ε transition

This situation may only occur if language described by regular expression associated with a stack symbol contains ε. Automaton might not know, if it should pop some of these stack symbol using ε transitions or if it should wait for them to be accepted as usually – with transitions that read input.

Example of regular expression with that problem: $\overline{\underline{B+\varepsilon}}\, B$. This regular expression already contains stack symbol. We can pretend that it is a derivative of some bigger regular expression and this part is used only to give clear example.

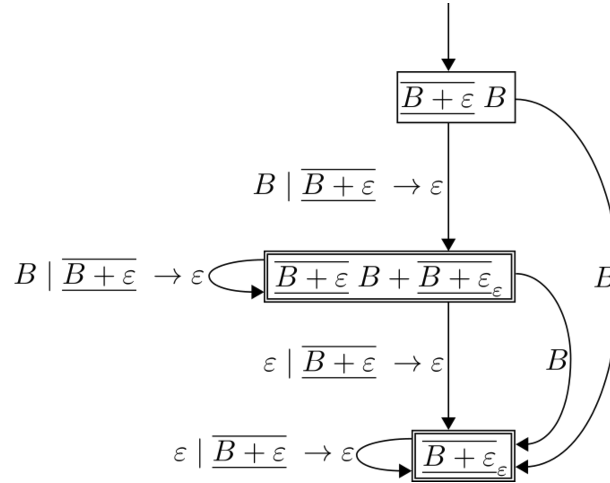


Figure 2.9: Example of derivation of stack symbol containing ε

To solve this, we introduce yet another unary operator: $\overline{\underline{a}}_\varepsilon$. This operator marks position, where the stack symbol $\overline{\underline{a}}$ can be popped using only ε transition.

This operator will stay in regular expression, when using transition that does not change stack and will be removed when using transition that changes stack.

This also means that we have to create $\varepsilon \mid \overline{\underline{E_1}} \to \varepsilon$ transitions from states that begin with stack symbol which contains ε.

2.3.3 Equations

We use all the notions from section 2.2.7 and add some new ones:

Sigma functions:

- $\sigma_{\varepsilon}(\overline{\underline{E_1}}_p) = \overline{\underline{E_1}}_p$
- $\sigma_{push}(\overline{\underline{E_1}}_p) = \emptyset$
- $\sigma_{pop}(\overline{\underline{E_1}}_p) = \emptyset$

- $\sigma_{\varepsilon}(\overline{\underline{E_1}}_{\varepsilon}) = \overline{\underline{E_1}}_{\varepsilon}$
- $\sigma_{push}(\overline{\underline{E_1}}_{\varepsilon}) = \varepsilon$
- $\sigma_{pop}(\overline{\underline{E_1}}_{\varepsilon}) = \varepsilon$

- $\sigma_{\varepsilon}(\overline{\underline{E_1}}\ ) = \overline{\underline{E_1}}_{\varepsilon} \qquad \mid \varepsilon \in l(E_1)$

Epsilon transitions:

- $\dfrac{d\overline{\underline{E_1}}}{d\varepsilon \mid \overline{\underline{E_1}} \ \to \varepsilon} = \overline{\underline{E_1}} \qquad \mid \varepsilon \in l(E_1)$
- $\dfrac{d\overline{\underline{E_1}}_{\varepsilon}}{d\varepsilon \mid \overline{\underline{E_1}} \ \to \varepsilon} = \overline{\underline{E_1}}_{\varepsilon} \qquad \mid \varepsilon \in l(E_1)$

Early stack pop:

- $\dfrac{dE_1}{dx \mid \overline{\underline{E_2}} \ \to \varepsilon} = \dfrac{d\overline{\underline{E_2}}_p \cdot E_1}{dx}$
- $\dfrac{d\overline{\underline{E_1}}_p \cdot E_2}{dx \mid \varepsilon \to \varepsilon} = \dfrac{dE_1 \cdot E_2}{dx \mid \varepsilon \to \varepsilon} \qquad \mid \dfrac{dE_2}{d\overline{\underline{E_1}} \ \mid \overline{\underline{E_1}} \ \to \varepsilon} \neq \emptyset$

Chapter **3**

Comparison of methods for parsing tree languages

This chapter intends to cover characteristics of some methods that can be used to recognize tree languages with some generalizations to context-free languages. Characteristics of tree languages and the way they affects parsing process will be discussed.

3.1 Similarities between method of derivatives and LR parsers

In this section, automatons created using regular tree expression derivatives and automatons created using a common parsing method – LR parsing – will be compared.

Basic similarities and differences of the two approaches will be shown on few examples.

We will compare pushdown automaton created by method of regular tree expression derivatives and graph of LR items of LR parser for the same language.

To create LR parser, one needs grammar for particular language. It is usually quite simple to convert regular tree expression into a grammar describing the same language. Formal definition of this conversion will not be presented in this thesis though (because formal definition would not be that simple).

3.1.1 First example – simple automaton

This is example of a simple regular tree expression $(A\Box B)^{*,\Box} \cdot_{\Box} C$. Derivation parser is in figure 3.1, LR parser is in figure 3.2.

As seen on the examples, there are some similarities. Particularly, the triangle containing edges C_1 and C_2 is similar in both graphs.

This similarity occurs because both methods work the same way when parsing single grammar rule (single substitution or tree iteration in regular tree expression). As seen in next example, the difference appears when we will try to create parser from grammar with more grammar rules (more substitutions in regular tree expression).

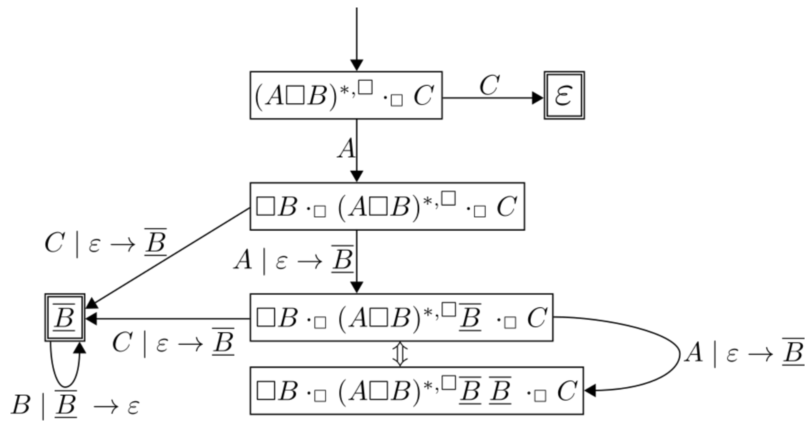


Figure 3.1: First example – derivatives of $(A\Box B)^{*,\Box} \cdot_{\Box} C$

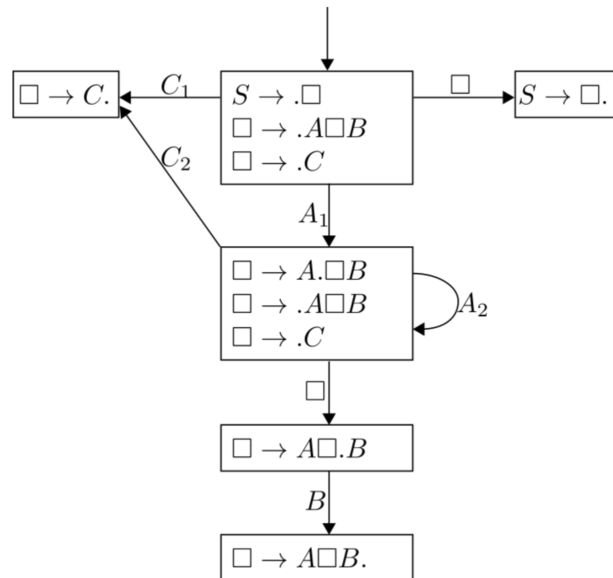


Figure 3.2: First example – LR parser for $(A\Box B)^{*,\Box} \cdot_{\Box} C$

3.1.2 Second example – differences in substitution

The main difference between LR parser and derivation method can be seen on example of substitution operation on longer string.

Regular tree expression $AB\Box CD\Box EF \cdot_{\Box} XY$ and corresponding grammar $S \rightarrow AB\Box CD\Box EF; \Box \rightarrow XY$ are turned into parsers in figures 3.3 and 3.4.

As seen on the images, both parsers create the same automaton for derivation of strings *AB*, *CD*, *EF* and XY. There is however difference in the way they are joined.

Method of derivatives joins these parts in the order in which they are read. This causes that each of both sequences of XY is read by different part of the automaton.

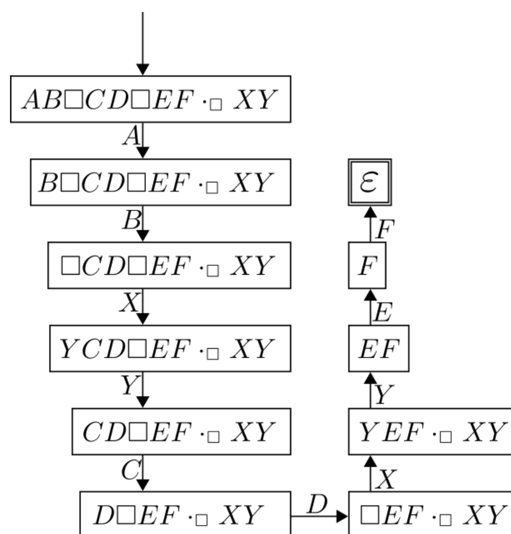


Figure 3.3: Second example – derivatives of $AB\Box CD\Box EF \cdot_{\Box} XY$

The LR parser on the contrary has different approach based on understanding the language as a grammar. Grammar has rules and nonterminal symbol from which the language is composed. This corresponds with the ability of the parser to *go back* in the graph when reduction (applying a grammar rule) is performed. This ability can also sometimes work as a cycle.

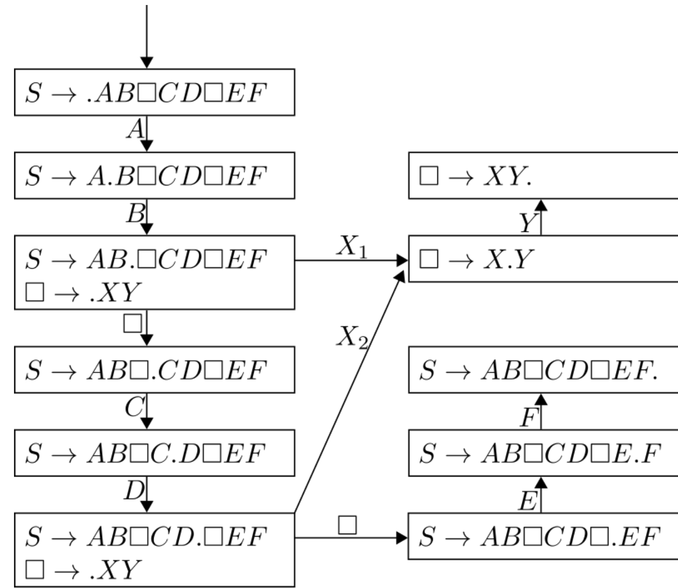


Figure 3.4: Second example – LR parser for $AB\square CD\square EF \cdot_{\square} XY$

3.1.3 Third example – tree iteration and cycles

On example of regular tree expression $(AB\square CD)^{*,\square} \cdot_{\square} XY$ and its corresponding grammar, we can see that method of derivatives creates more cycles in the graph than the method of LR parsing.

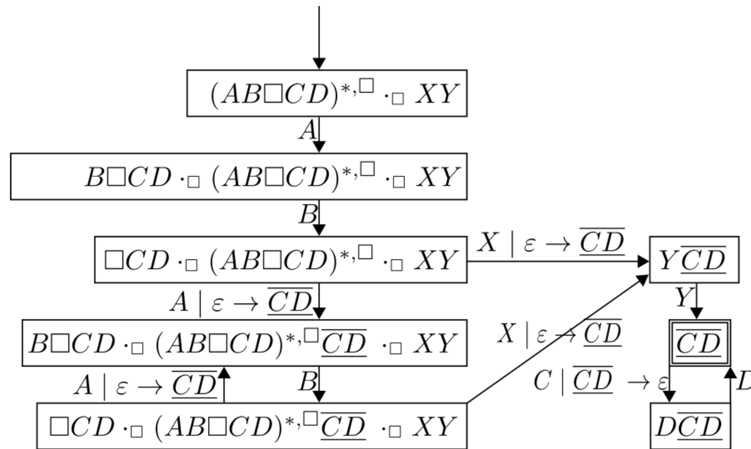


Figure 3.5: Third example – derivatives of $(AB\square CD)^{*,\square} \cdot_{\square} XY$

In this example, the parser first needs to read arbitrary number of left parts of recursion AB, then it reads symbols XY and then it reads the same number of right parts of recursion CD.

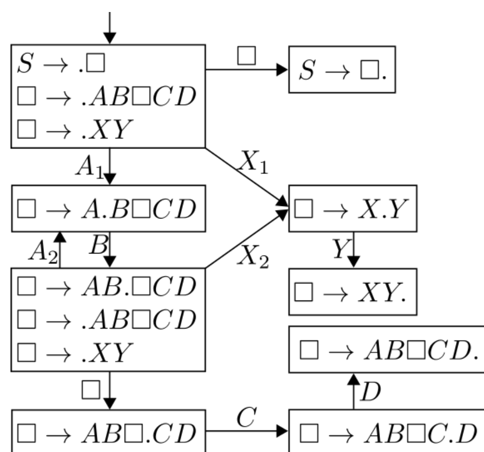


Figure 3.6: Third example – LR parser for $(AB\square CD)^{*,\square} \cdot_{\square} XY$

It is interesting to look closer on the difference in how both methods handle this situation and why LR parser needs only one cycle in graph while method of derivatives needs two cycles.

The LR parser first reads all occurrences of AB in the input using the cycle in the graph of LR items. Then it reads input symbols XY and then it comes to read all the right parts of recursion CD. The parser each time recalls, what it has read earlier (by applying the grammar rule $\square \to AB\square CD$) and then it reads what is required by applied rule.

In this case it either reads next CD or it stops the parsing (accepts the input). This is more apparent in example of figures 2.5 and 3.7.

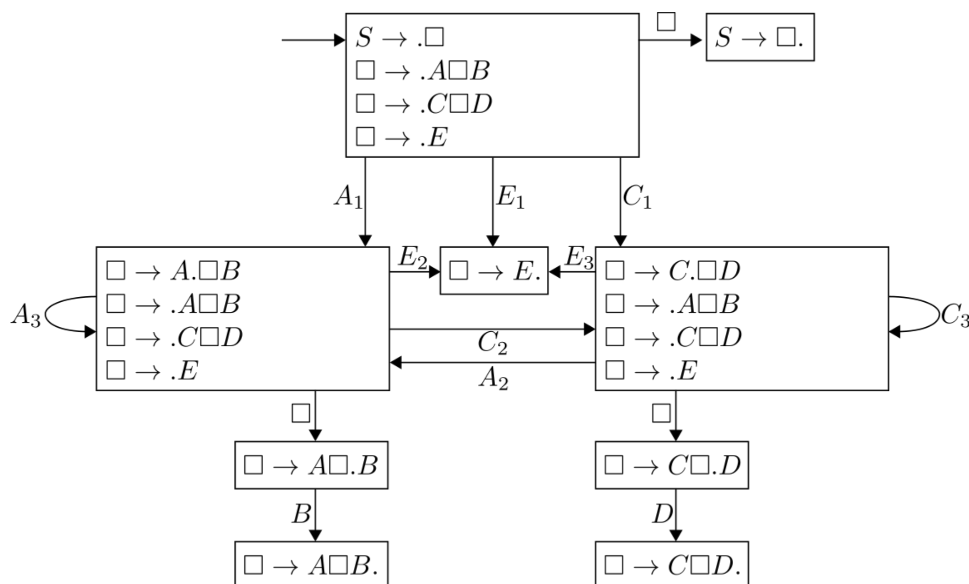


Figure 3.7: Third example – LR parser for $(A\square B + C\square D)^{*,\square}$ (equivalent regular expression derivation is in figure 2.5)

3.1.4 Conclusion

The important thing to note is, that the LR parser uses stack to store information about what has been read, while derivatives automaton uses stack to store what should be read in the future.

This means that LR parser needs to go back in the graph sometimes (while doing reduction) to *recall* what he has read and then he can continue reading. Derivation automaton does not have to return back because it has the information about what should be read on the stack.

3.1.5 Extra example: two different substitution operators

This is just another example comparing derivatives and LR parsers, this time for more complex language. Regular tree expression contains two different substitution operators.

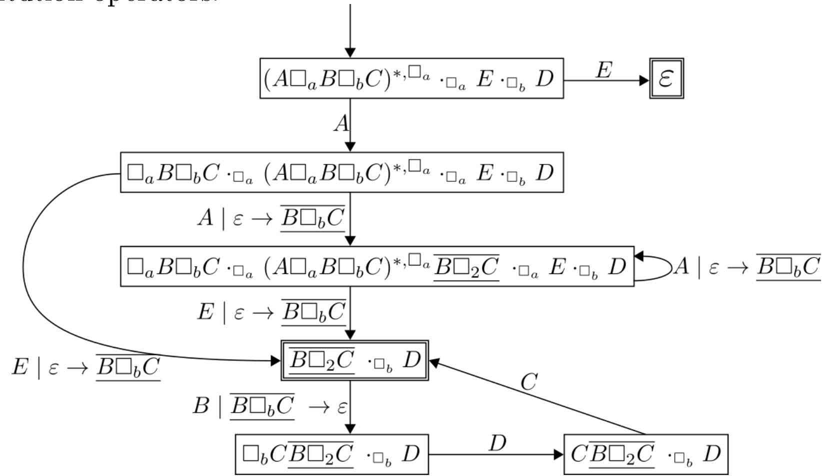


Figure 3.8: Derivatives of $(A\square_a B\square_b C)^{*,\square_a} \cdot_{\square_a} E \cdot_{\square_b} D$

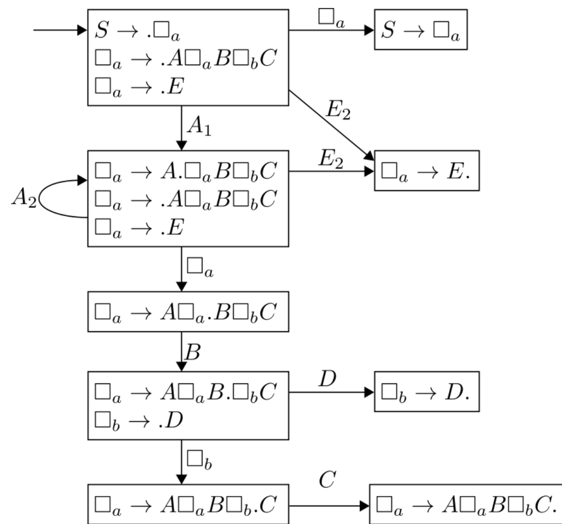


Figure 3.9: LR parser for $(A\square_a B\square_b C)^{*,\square_a} \cdot_{\square_a} E \cdot_{\square_b} D$

3.2 Grammar classes and parsers

This section should review methods for parsing context-free languages, classify languages by their ability to be processed by these parsers and provide most accurate relations between these classes.

We will in fact not classify *languages* but *grammars* or *regular tree expressions* by their ability to be parsed by different parsers. Most languages can be described with different grammars from different classes (or with different regular tree expressions).

Methods for parsing context-free languages

- LL parsing – top down application of grammar rules
- LR parsing – bottom up parser using language grammar
- regular tree expression derivatives – method proposed in chapter 2
- regular tree expression automaton composition – method proposed in citation [2]

3.2.1 LR and LL parsers

Following figure shows relations of some LR and LL parsing methods.

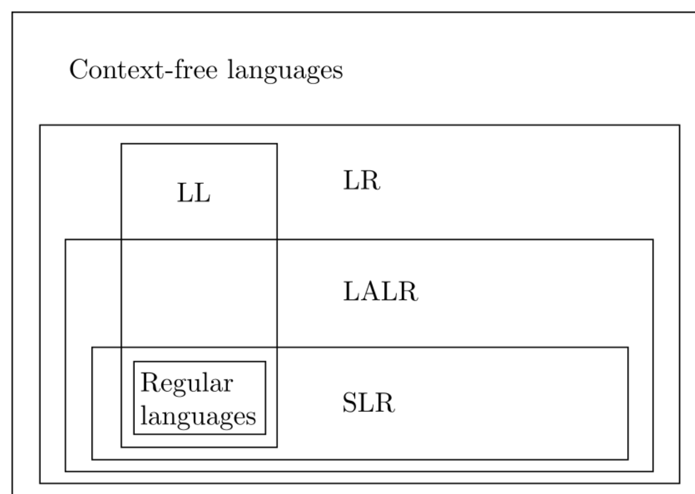


Figure 3.10: Hierarchy of LR and LL languages

3.2.2 Regular tree expression derivatives

This figure shows relations between three versions of algorithm creating deterministic pushdown automatons from regular tree expression as described in chapter 2.

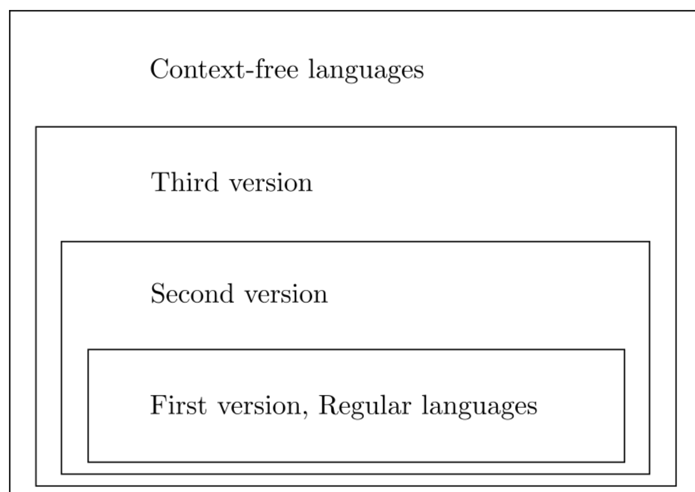


Figure 3.11: Hierarchy of languages parsable using regular tree expression derivatives

3.2.3 Regular tree expression automaton composition

This method will not be deeply covered in this thesis. Regular tree expressions are usually smaller trees connected with operators alternation, substitution and tree iteration. This method constructs automatons that accept these small trees and then connects the automatons together according to used operators. This produces nondeterministic automatons which needs to be determinized afterwards.

This method should be able to parse all languages that can be described using regular tree expressions.

3.2.4 Relations between LR languages and regular tree expression derivatives

There seems to be no simple relation between languages that can be parsed by different LR or LL parsers and languages that can by parsed using regular tree expression derivatives.

As we know, languages can belong to any subset of LR languages but knowing it will give us no information about which version of algorithm using derivatives should be used and vice versa.

3.3 Tree languages

In this section, we discuss attributes of tree languages – languages that are composed of trees connected with alternation, substitution and tree iteration operators.

3.3.1 Language (grammar) class according to method of derivatives

The second version of algorithm using derivatives (section 2.2) should be probably enough to parse all tree languages.

There are two situations, when a regular tree expression can not be parsed only using second version of mentioned algorithm and the third version is required. The third version is needed if one of these two patterns appear in some point during construction of the automaton:

- $(A + \varepsilon)[A]$ – expression may be derived using the same input symbol both with and without popping the stack

  This expression can result for example from initial expression $\mid \Box a \cdot_{\Box} (\mid a+ \mid)$. This is not realy a tree expression because one of it operands $(\mid)$ is not a tree.

- $[A + \varepsilon]A$ – stack symbol may be popped without reading input symbol

  This results for example from $(\mid \Box a+ \mid \Box)^{*,\Box}$. The operand $\mid \Box$ is not a valid tree.

  This pattern can not appear in a true regular tree expression, because tree in postfix bar notation always has at least one symbol after $\Box$ (unless it is only $\Box$ as a root, but that is not a problem either).

So the second pattern can not appear in tree languages and we think that the second pattern probably neither. Tree languages should be parsable by second version of algorithm of regular tree expression derivatives (section 2.2).

3.3.2 Right and left iteration

If we restrict regular tree expressions to contain only trees in postfix/prefix notation as operands, then the languages described by these regular tree expressions can not contain right or left iteration.

Example of a right iteration: $(AB\Box)^{*,\Box}$, left iteration: $(\Box AB)^{*,\Box}$.

This would require tree to have substitution symbol $\Box$ entirely on the right or on the left of the bar notation representation of the tree. The only tree, that can have that is tree that has the substitution symbol as a root of the tree: $\Box$.

If the tree has root other than the substitution symbol $\square$, for example A, then it has the bar symbol $|$ on one end of the string and name of the root, for example A, on the other side.

Prefix bar notation: $A \ldots |$

Postfix bar notation: $| \ldots A$

Conclusion

Tree languages have many interesting characteristics. We went through some approaches to tree and context-free languages recognition, mainly *LR parser* and *method of derivatives*, which is a new algorithm proposed in this thesis.

Method of derivatives

Method of derivatives which was proposed in the thesis is a new algorithm for recognition of tree and context-free languages. This method is based on derivations of regular tree expressions, which is a generalization of method used for parsing of regular languages. This generalization allows us to create pushdown automatons.

The method of derivatives seems to be simple enough to be considered useful in some cases. The main disadvantage of this method when comparing to LR parser is that it does not fully recognizes the structure of input word – it only tells us, if the word belongs to given language defined by regular expression or not.

The main advantage of method of derivatives is that it behaves like usual regular expression matcher. It can be modified to save specific parts of input word while matching it to regular expression. Parts of this method can also be used to enhance capabilities of simple regular expression matchers.

Similarity with LR parser

Pushdown automatons created with the method of derivatives are similar to graphs of LR item sets (part of LR parsers), but they also differ in some important points. It would be probably not simple, neither perspective to try to convert graph of LR items to deterministic automaton similar to the one created by method of derivatives.

Difference between LR parser and the method of derivatives

The difference is in the way these two parsers use stack.

LR parser uses stack to store information about history – what was read in the past. So when the LR parser wants to read next input, it will first recall what was read before (with *reduction* action) and then read the input.

On the other side, the automaton resulting from method of derivatives of regular tree expression uses stack to store information about what should be read. Every time the automaton pops a stack symbol, a part of regular expression specified by the stack symbol is added to the beginning of the regular expression and then the automaton reads it.

Further research

There are some topics that might be considered for further research. They are related to each other and are generally meant to broaden knowledge of regular expressions and context-free languages.

Better formalization of method of derivatives

The method of derivatives of regular tree expression proposed in this thesis could surely be improved. We tried to provide formal definitions of three different versions of algorithm of derivatives in similar way, in which common regular expressions are defined.

It might be beneficial to try to clean and simplify the definitions. Choosing different approach than the one used by common regular expressions might help.

Implementation of proposed method of derivatives

It was initially intended to implement the proposed method as part of the thesis. It however showed up, that it is harder to recognize context-free languages than expected, so there was no time left to implement the algorithm.

A sample implementation should help to improve the algorithm, because some inaccuracies in presented equations might be found.

Conversion between language grammar and regular tree expression

It is usually fairly easy to convert language grammar to regular tree expression and vice versa just by hand. It would be probably more complicated to create an algorithm that can do this. Construction of such algorithm could help to explain more deeply, how regular expressions work.

Modifying the method of derivatives to recognize structure of input word

Method of derivatives can recognize given context-free languages, but it is not good at recognizing the structure of particular word.

Output of LR parser and LL parser is a sequence of grammar rules in the order they would be applied to form the input word. This gives us a good way to understand the structure of the word. Method of derivatives only tells us, if the input word is in language defined by the regular expression or not.

It might be beneficial to find out, if there are any ways to get more information about input word, than just *accepted/unaccepted*.

One way to get some information about input word could be to use match groups similar to regex implementations in many programming languages. Parts of the regular expression would be enclosed in parentheses and matcher then returns array containing which input symbols were used to match these parts of the regular expression.

Bibliography

[1] Melichar, B.; Janoušek, J.; Vagner, L. *Syntaktická analýza a překlad*. Czech Technical University in Prague, Faculty of Information Technology, 2010.

[2] Polách, R. *Tree Pattern Matching and Tree Expressions*. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2011.

[3] Knuth, D. On the Translation of Languages from Left to Right. *Information and Control*, volume 8, 1965: pp. 607–639.

[4] Brzozowski, J. A. Derivatives of Regular Expressions. *Journal of the ACM*, volume 11, no. 4, Oct. 1964: pp. 481–494.

Appendix **A**

Contents of enclosed CD

```
└── latex-source/ ......... the directory of LaTeX source codes of the thesis
│   ├── images/ ............................. bitmap versions of images
│   ├── svg/ .................................... svg versions of images
│   ├── DP_Košař_Jan_2015.tex ........................ LaTeX source file
│   ├── makefile ...................... makefile build script for the thesis
│   └── README ........... instructions how to build the PDF file from sources
└── DP_Košař_Jan_2015.pdf ................ text of the thesis in PDF format
```