Representation of Regular Formal Languages

by

Aslam Safla

Submitted in fulfillment of the academic

requirements for the degree of

Master of Science

in the

School of Mathematics, Statistics and Computer Science

University of KwaZulu-Natal

Pietermaritzburg

2014

Abstract

This dissertation presents three different approaches to representing Regular Formal languages, i.e., regular expressions, finite acceptors and regular grammars. We define how each method is used to represent the language, and then the method for translating from one representation to another of the language. A toolkit is then presented which allows the user to input their definition of a language using any of the three models, and also allows the user to translate the representation of the language from one model to another.

Preface

The research described in this dissertation was carried out in the School of Computer Science, University of KwaZulu-Natal, Pietermaritzburg, from May 1999 to July 2013, under the supervision of Professor Hugh Murrell.

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any University. Where use has been made of the work of others it is duly acknowledged in the text.

____

Student's Signature

____

Supervisor's Signature

Acknowledgements

I would first like to thank my wife, Sumaiyah, for all her encouragement and support over these last few years. I would also like to thank my parents, Suleman and Rabiya, for their support.

Thanks must also go to Professor Yuri Velinov for all his assistance and to Professor Hugh Murrell for taking over from Professor Velinov, and for all the advice that he has given me.

Table of Contents

Table of Contents 5
Table of Figures 9
Chapter 1 11
Introduction 11
1.1. The Aim of the Study 11
1.2. Structure of this Dissertation 13
Chapter 2 15
Preliminaries 15
2.1. Definitions 15
2.1.1. Some concepts [44] 15
2. 1.2.Operations on formal languages [44] 17
2. 1.3. Regular languages [24] 18
2. 1.4. Regular Expressions 20
2. 1.5. Regular Expressions, Formal Grammars and Finite Acceptors 21
2.2. Generative Grammar. Chomsky Hierarchy 25
2.2.1. Main definitions 25
2.2.2. Classification 26
2.2.3 Redundant symbols 29
Chapter 3 34
Related software 34
3.1. Complete Toolkits 34
3.2. Bi-Model Toolkits 35

3.3. Single Model Toolkits 38

Chapter 4 40

Translation between models 40

4.1 Translating from Regular Expression to Nondeterministic Finite Acceptor 40

4.1.1 Base Algorithm 42

4.1.2 Shortcomings 46

4.1.3 Proposed Algorithm 46

4.1.4 Advantages of proposed algorithm 51

4.2 Translation from Nondeterministic Finite Acceptor to Deterministic Finite Acceptor 55

4.2.1 Base Algorithm 56

4.2.2 Compatibility 58

4.3 Minimising the Number of States of a Deterministic Finite Acceptor 61

4.3.1 Base Algorithm 63

4.3.2 Application 65

4.4 Translating from Deterministic Finite Acceptor into a Regular Expression 67

4.4.1 Base Algorithm 67

4.4.2 Application 69

4.5 Translating from Regular Expression into a Regular Grammar 71

4.5.1 Base Algorithm 72

4.5.2 Application 72

4.6 Translating from Regular Grammar into a Regular Expression 74

4.6.1 Base Algorithm 74

4.6.2 Application 76

4.6.3 Possible Problem Areas 78

4.7 Translating from Regular Grammar into a Nondeterministic Finite Acceptor 79

4.7.1 Base Algorithm 79

4.7.2 Application 80

4.8 Translating from Deterministic Finite Acceptor into a Regular Grammar 82

4.8.1 Base Algorithm 82

4.8.2 Application 83

4.9 Conclusions 84

Chapter 5 85

Implementation 85

5.1 Programming Languages 85

5.2 Model Designs 86

5.2.1 The Regular Expression Editor 86

5.2.2 The Finite Acceptor Editor 87

5.2.3 The Regular Grammar Editor 89

5.3 Translation techniques 91

5.3.1 Translating from Regular Expression to Nondeterministic Finite Acceptor 91

5.3.2 Translating from Nondeterministic Finite Acceptor to Deterministic Finite Acceptor 93

5.3.3 Minimizing the Deterministic Finite Acceptor 94

5.3.4 Translating from Regular Grammar to Deterministic Finite Acceptor 94

5.3.5 Translating from Deterministic Finite Acceptor to Regular Grammar 95

5.3.6 Translating from Regular Expression to a Regular Grammar 95

5.3.7 Translating from Regular Grammar to a Regular Expression 96

5.3.8 Translating from Deterministic Finite Acceptor to Regular Expression 97

5.4 Test case examples 99

5.5 Content of accompanied CD 103

Chapter 6 105

Conclusion 105

6.1 Summary 105

6.2 Future Improvements 106

References 107

Appendix 116

Appendix A: User Documentation 116

Table of Figures

Figure 1: Relationship between different classes of languages 28

Figure 2: NFA representing the language λ.. 42

Figure 3: NFA representing the language *a* 43

Figure 4: NFA representing the language N(*s/t*) 43

Figure 5: NFA representing the language N(*st*).. 44

Figure 6: NFA representing the language N(*s**)... 45

Figure 7: NFA representing the language *a* 47

Figure 8: λ-free NFA representing the language N(*s/t*)... 48

Figure 9: λ-free NFA representing the language N(*st*).. 49

Figure 10(a) : λ-free NFA representing the language N(*s**), where the number of states of

N(*s**) is greater than two... 50

Figure 11: λ-free NFA representing the language a.b.c/(d.e)* 52

Figure 12: λ-free NFA representing the language (d.e)* / (d.e)*.a.b.c 53

Figure 13(a): λ-free NFA representing the language a.b.c.d*/a.b.d* 54

Figure 14: Transition table for the DFA representing the language (a.b.c.d*/a.b.d*)............. 60

Figure 15: State diagram for the DFA representing the language (a.b.c.d*/a.b.d*)................ 60

Figure 16: Transition table for the minimal DFA representing the language (a.b.c.d*/a.b.d*) 66

Figure 17: State diagram for the minimal DFA representing the language (a.b.c.d*/a.b.d*)... 66

Figure 18: State diagram for the DFA representing the language (a.b)*.a.b.b 81

Figure 19: State diagram for the DFA representing the language a.b.(c/d*)........................... 83

Figure 20: Design of the regular expression editor... 87

Figure 21: Design of the finite acceptor editor 88

Figure 22: Design of the regular grammar editor 90

Figure 23: Regular expression editor for (a.b.c.d*/a.b.d*) 99

Figure 24: Finite acceptor editor for (a.b.c.d*/a.b.d*) 100

Figure 25: NFA editor for (a.b.c.d*/a.b.d*) 101

Figure 26: Regular Grammar for (a.b.c.d*/a.b.d*) translated from a NFA 101

Figure 27: NFA for (a.b.c.d*/a.b.d*) translated from a regular grammar 102

Figure 28: DFA for (a.b.c.d*/a.b.d*) 102

Figure 29: Regular grammar translation of DFA for (a.b.c.d*/a.b.d*) 103

Chapter 1

Introduction

In this chapter, an introduction to the contents and structure of this dissertation will be presented.

1.1. The Aim of the Study

In the representation of languages, it is likely that the language is infinite. There are three approaches to describe these languages in some finite way.

- The first approach is to use string operations to combine individual symbols and simpler strings.
- The second approach is to specify a mechanism for *recognizing*, in an algorithmic sense, strings in a language.
- The third approach is to specify simple recursive *grammar* rules, by which strings in the language can be generated.

To recognize the language is to have a method to determine whether or not any string is in the language. The simplest language is the *regular* language. This important class of languages and the results we derive play an important role in applications such as compilers, spelling checkers, and web browsers. Some of the tools we develop to study these languages such as finite automata play an important role in the modeling of a wide variety of systems such as digital logic controllers, vending machines, and biological processes.

If a language can be described as one that can be generated from the null string and the individual symbols in the alphabet, using a finite number of applications of a certain standard operation, then this will satisfy the first approach. A notational device for representing the symbols and the operations is a *regular expression*. A regular language is one that can be described by a regular expression.

It is often necessary to “remember”, in order to recognize a language, especially when one is restricted to a single pass through the string in processing the string. The device that would be most appropriate here would be the *finite automaton*. A language is regular if a finite automaton can describe the language. This approach also gives us more insight into what it means for a language *not* to be regular. The finite automaton can also be generated directly so as to handle the other abstract machines we shall consider.

In order to generate the elements of a regular class, a grammar is used. Context-free grammars provide rules which are reminiscent of the grammar rules in programming languages and other “formal” languages. Although it does not make it immediately obvious that every regular language is of this type, it is not difficult to find a subclass of the context-free grammar corresponding precisely to the class of regular languages. This subclass is that of regular grammars.

The study will address the representation of the three models and the translation from one model to another. A software toolkit will also be presented, which will demonstrate how each model can be represented, and how translations from one model to another can be accomplished. In this dissertation, a brief description of the models for representing regular

languages is first given, followed by an introduction to the different algorithms that allow us to translate one description of a language into another description of a different type. A taxonomy of all of the known algorithms concerned in this study will not be presented; however, a few different approaches are given, followed by the algorithm that will be used in the toolkit.

In the following section, we present a broad overview of the structure of this dissertation.

1.2. Structure of this Dissertation

This dissertation is divided into six chapters.

- Chapter 1 contains the introduction (this chapter).
- Chapter 2 contains the preliminaries, where definitions of different concepts are given.
- Chapter 3 looks at related software that exists which addresses the methods of translating from one model to another.
- The different algorithms that are studied are addressed in Chapter 4. For each algorithm, a brief introduction to the algorithm is first given, followed by the algorithm, and then by an example of how this algorithm is implemented.
- Chapter 5 presents a toolkit that implements most of the algorithms presented in Chapter 4. An introduction to the design is firstly given, followed by a practical performance of many of the algorithms presented in Chapter 4.

- Chapter 6 lists the general conclusions, and some of the improvements that could be made.

Chapter 2

Preliminaries

In this chapter, a number of definitions and properties required for the reading of this dissertation are presented. The reader that is well versed in these matters may skip this chapter and refer to it while reading the rest of the dissertation.

2.1. Definitions

2.1.1. Some concepts [44]

Symbols are initial objects in the linguistic construction each one considered as indivisible.

A finite set of symbols is called an *alphabet*.

A *string* (sentence, word) over an alphabet is an object obtained by attachment of several symbols consecutively to each another. It can be considered as a finite sequence of symbols.

A *phrase* is a string over an alphabet such that external meaning is associated with it (that is a meaningful string).

A *language* over an alphabet is a set of phrases together with the external meaning associated with them. The construction of the phrases from symbols or other phrases together with the rules of this construction is called the *syntax* of the language. The external meaning associated

with the phrases and the way in which they are constructed is called the *semantic* of the language.

A *formalized* language is a language in which the syntax is described independently of its semantic.

A *formal language* over an alphabet is just a set of strings over the alphabet. The set may be empty, finite or infinite.

Because a language is a set of strings, the words language and set are often used interchangeably in talking about formal languages.

L(M) is the notation for a language accepted by a machine M.

The machine M accepts a certain set of strings, and rejects others.

L(G) is the notation for a language defined by a grammar G.

The grammar G generates a certain set of strings, thus a language.

M(L) is the notation for a machine that accepts a language.

The language L is a certain set of strings.

G(L) is the notation for a grammar that generates a language.

The language L is a certain set of strings.

A *formal grammar* is a derivation system which produces the strings of a formal language.

2. 1.2.Operations on formal languages [44]

- The *union* of of two languages L_1 and L_2 (written $L_1 \cup L_2$) is the language

$$L_1 \cup L_2 = \{s \mid s \in L_1 \vee s \in L_2\}$$

- The concatenation of two languages L_1 and L_2 (written $L_1 \bullet L_2$ or simply $L_1 L_2$) is the language

$$L_1 \bullet L_2 = \{s \mid s = s_1 s_2 \,\&\, s_1 \in L_1 \,\&\, s_2 \in L_2\}$$

- The Kleene closure of a language L (written L^*) is the language

$$L^* = \left\{ s \mid \exists i \in \mathbf{N} \left(s \in L^i \right) \right\}$$

or in another way $L^* = \bigcup_{i=0}^{\infty} L^i = \lambda \cup L \cup L^2 \cup ... \cup L^i \cup ...$, where λ denotes the empty string.

Intuitively, L^* denotes "zero or more concatenations of the elements of L".

- The positive Kleene closure of a language L (written L^+) is the language

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L \cup L^2 \cup ... \cup L^i \cup ...$$

2. 1.3. Regular languages [24]

Let A be an alphabet. Any language over A is a subset of A^*. The set A^* together with the operations union, concatenation and Kleene closure, and the constant sets $\varnothing$, $\{\lambda\}$ are a prototype for an algebraic system that can be specified as a tuple: $\langle A, \cup, \bullet, {}^*, \varnothing, \{\lambda\}\rangle$ where A is a set of objects which fulfil the following axioms in relation to the operations $\cup$, $\bullet$, * and the constants $\varnothing$, $\{\lambda\}$:

1. $L \cup M = M \cup L$

2. $(L \cup M) \cup N = L \cup (M \cup N)$

3. $(L \bullet M) \bullet N = L \bullet (M \bullet N)$

4. $L \bullet (M \cup N) = (L \bullet M) \cup (L \bullet N)$

5. $(M \cup N) \bullet L = (M \bullet L) \cup (N \bullet L)$

6. $L \cup L = L$

7. $L \cup \varnothing = L$

8. $\varnothing \bullet L = L \bullet \varnothing = \varnothing$

9. $\{\lambda\} \bullet L = L \bullet \{\Lambda\} = L$

10. $L^* = L \cup L^*$

11. $(L^*)^* = L^*$

12. $\varnothing^* = \{ \lambda \}$

13. $L^* = \{ \lambda \} \cup L \cup L^2 \cup \ldots \cup L^{k-1} \cup L^k$

14. $L^* = \{ \lambda \} \cup L^+$

for every $L, M, N \in A^*$

Notice that axiom 13 describes an infinite family of equations. Some additional properties that can be proved from the axioms are:

15. $\{ \lambda \}^* = \{ \lambda \}$

16. $L^* = (\{\lambda \} \cup L)^*$

17. $L^* \bullet L^* = L^*$

18. $(L \bullet M)^* \bullet L = L \bullet (M \bullet L)^*$

19. $(L \cup M)^* = (L^* \cup M^*)^* = (L^* \cup M)^* \bullet L^*$

for every $L, M, N \in A$.

From the point of view of formal languages, we will concentrate only on the algebraic language where, given an alphabet A, any subset of A^*, denoted by $P(A^*)$, is a language over A and $\{\lambda\}$ is the one element set containing the empty string λ .

DEFINITION 1 [3]
The algebraic system

$$\langle P(A^*), \cup, \bullet, *, \varnothing, \{\lambda\}, (\{a_i\}) \rangle$$

will be called the **regular algebra** over the alphabet $A=\{a_1,a_2,\ldots,a_m\}$.

Subsets of A^* which can be generated from the elements of A using the operations union, concatenation or Kleene closure, are called regular languages. They can be defined more precisely by the following inductive definition.

DEFINITION 2 [3]

- $\varnothing$ is a regular language;
- $\{\lambda\}$ is a regular language;
- $\{a_i\}$ is a regular language for every $a_i \in A$;
- if L_1 and L_2 are regular languages so is $L_1 \cup L_2$;
- if L_1 and L_2 are regular languages so is $L_1 \bullet L_2$;
- if L is a regular languages so is L^*
- only the subsets of A^* which can be obtained by the above rules are regular languages

In order to define a regular language, a regular expression is used. Finite acceptors are acceptors used to determine whether a string belongs to the language or not. Definitions of these representations will now be presented.

2. 1.4. Regular Expressions

Regular languages in A^* where A is an alphabet can be described by a (meta) language called the **language of regular expressions** for the alphabet A.

DEFINITION 3 [3]

The **language of regular expressions** for the alphabet $A = \{a_1, a_2, \ldots, a_m\}$ is built over the symbols $A \cup \{/, \bullet, *, \lambda, \varnothing, (,)\}$. Its phrases are defined inductively as follows:

- λ and $\varnothing$ are regular expressions;
- $a_1, a_2, \ldots, a_m$ are regular expressions;
- if R_1 and R_2 are regular expressions then (R_1 / R_2) is a regular expression;
- if R_1 and R_2 are regular expressions then $(R_1 \bullet R_2)$ is a regular expression;
- if R is a regular expression then R^* is a regular expression;
- only the strings which can be obtained by the above rules are regular expressions.

A finite set of regular expressions $\{R_1, R_2, \ldots, R_k\}$ also describes a regular language - the union of the particular languages determined by the elements. Indeed, this language is defined by $R_1 / R_2 / \ldots / R_k$.

ADDITIONAL NOTATIONAL CONVENTIONS

- Usually we skip the symbol $\bullet$.
- R^+ denotes $R \bullet R^*$.
- $[a-z]$ denotes $a/b/.../z$.

2. 1.5. Regular Expressions, Formal Grammars and Finite Acceptors

Regular languages on an alphabet can be represented by three different equivalent methods: using regular expressions, using finite acceptors which are partial case of transition systems, and using type 3 formal grammars .

Regular definitions combine the regular expressions approach and the formal grammar approach. The initial idea is simply to denote a regular expression by a symbol and then use the symbol in other definitions. Allowing recursive definitions can extend the idea further.

DEFINITION 4 [2]

If ***A*** is an alphabet and $\{D_1 D_2 \ldots D_n\}$ is an additional set of symbols then a **regular definition** is a sequence of definitions of the form

$$D_1 \rightarrow R_1 \text{ , } D_2 \rightarrow R_2 \text{ ,...,} D_n \rightarrow R_n$$

where $R_1, R_2, ..., R_n$ are regular expressions over $A \cup \{D_1 D_2 \ldots D_n\}$

Restricted regular definitions use on the right side of a pseudoproduction only previously defined additional symbols.

DEFINITION 5 [22]

A ***deterministic finite acceptor*** is a system $\langle A, S, \delta, s_0, F \rangle$, where

- A is a set of input symbols;
- S is a set of states;
- $s_0 \in S$ is an initial state and $F \subseteq S$ a set of final states;
- $\delta : AxS \rightarrow S$ is a transition function

DEFINITION 6 [22]

A ***nondeterministic finite acceptor*** is a system $\langle A, S, \delta, \mathbf{I}, \mathbf{F} \rangle$, where

- A is a set of input symbols;
- S is a set of states;
- $\mathbf{I} \subseteq S$ is a set of ***initial states*** and $\mathbf{F} \subseteq S$ a set of ***final states***;
- $\delta : AxS \rightarrow P(S)$, where $P(S)$ is the power set of **S**, i.e. all subsets of **S**, is a transition function.

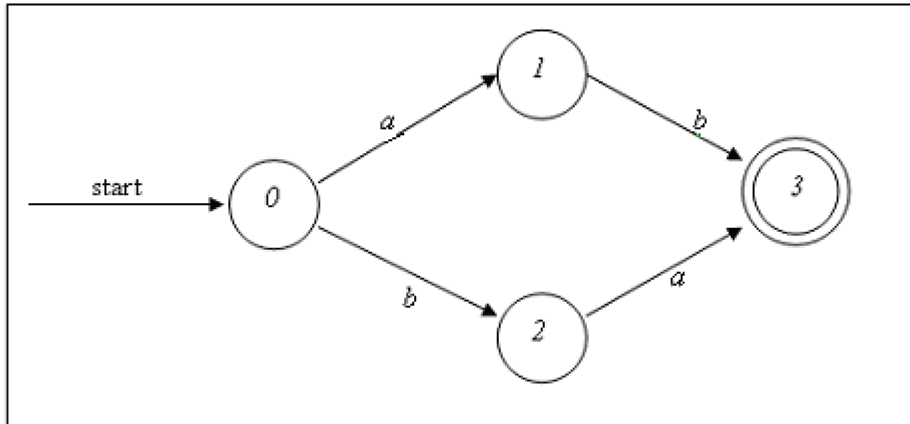
It should be noted that the difference between a NFA and a DFA is, in a DFA the transition function gives exactly one state, when applied from some state on an input symbol. In the NFA there can be several possible next states, and an interpreter will traverse all possible paths until either one of the paths ends in an accept state, or all paths have been traversed. Hence, the DFA is a particular case of the NFA.

A Deterministic Finite Acceptor such that every state is reachable from the start state and such that a final state is reached eventually, is called a reduced Deterministic Finite Acceptor. A reduced Deterministic Finite Acceptor may not be a complete Deterministic Finite Acceptor.

When drawing the state graph corresponding to a finite acceptor, we adopt the following conventions:

- All states are drawn as circles or ovals.
- Transitions are drawn as labeled (with λ or alphabet symbol $a \in A$) directed edges between states.
- Start states have a transition into a state without a source state (the transition does not come from another state).
- Final states are drawn as two concentric circles or ovals.
- All states are numbered.

For example, the finite acceptor below has four states, with state zero being the start state, and state 3 being the final state, with a transition on input *a* from state zero to state one, a transition on input *b* from state one to state three, a transition on input *b* from state zero to state two, a transition on input *a* from state two to state three:



The finite acceptor could also be represented using a **transition table**.

- Each row of the table is the name of one of the states of the finite acceptor.
- Each column of the table is a letter of the input alphabet.
- The entries inside the table are the transition states.

The state graph above can be represented using the transition table below:

State	Input Symbol	
	a	b
0	1	2
1		3
2	3	
3		

2.2. Generative Grammar. Chomsky Hierarchy

Generative grammars are derivation systems of a specific kind. They were introduced as a tool for studying the syntactical properties of the natural languages and are also called phrase structure grammars referring to their origin.

2.2.1. Main definitions

DEFINITION 7 [3]

A generative grammar is a system with main components presented by a tuple

$$\mathbf{G} = \langle \mathsf{A}, \mathsf{T}, \mathsf{R}, S \rangle$$

where

- $\mathsf{A} \equiv \mathsf{Alph}(\mathbf{G})$ is a finite set of symbols called the *alphabet* of the grammar;
- $\mathsf{T} \equiv \mathsf{Talph}(\mathbf{G})$ is a subset of A called the *terminal alphabet* of the grammar, with elements called *terminal symbols*;
- S is an element of $\mathsf{A} \setminus \mathsf{T}$ called the *initial nonterminal* or *start symbol* of the grammar.
- $\mathsf{R} \equiv \mathsf{Prod}(\mathbf{G})$ is a finite set of productions. Each production is of the form $u \to v$ for two strings $u \in S, v \in A^*$;

The set $\mathsf{N} \equiv \mathsf{Nalph}(\mathbf{G}) = \mathsf{A} \setminus \mathsf{T}$ is called the *nonterminal alphabet* or the *classifiers alphabet* of the grammar and its elements are called *nonterminal symbols* or *classifiers*.

A string $w \in \mathsf{A}^*$ is called a terminal string of the grammar if it contains only terminal symbols.

Productions of the form $u \to \lambda$ where λ is the empty string are called *λ-productions* or *empty productions*.

The main purpose of a generative grammar is the production of strings over its terminal alphabet. The set R of productions of a formal grammar $\mathbf{G}$ together with the initial nonterminal S constitute the set of derivation rules of the grammar. The derivation rules are used to produce string by derivations.

DEFINITION 8 [19]

A sequence of strings

$$w_1, w_2, \ldots, w_n$$

over the alphabet of a grammar $\mathbf{G}$ is a *derivation* (of length n) of w_n from w_1 in $\mathbf{G}$ iff for $i = 1,2,\ldots,n-1$ each w_{i+1} is obtained by rewriting a substring of w_i by another string corresponding to it according to a production of the grammar.

A string u is derivable from a string v in $\mathbf{G}$ if there is a derivation of u from v in $\mathbf{G}$.

DEFINITION 9 [19]

The set $\mathsf{Lang}(\mathbf{G})$ of all terminal strings derivable in a formal grammar $\mathbf{G}$ is called the *language generated by* $\mathbf{G}$ or simply the *language of* $\mathbf{G}$.

Two grammars $\mathbf{G}_1$ and $\mathbf{G}_2$ are *equivalent* iff $\mathsf{Lang}(\mathbf{G}_1) = \mathsf{Lang}(\mathbf{G}_2)$ i.e. if they generate the same language.

2.2.2. Classification

Properly imposed restrictions on the form of the used productions makes it possible to classify the generative grammars as well as the languages which they generate. A widely accepted

classification is known as the Chomsky Hierarchy. To generate the elements of a regular language, a subset of the Chomsky Hierarchy, the regular grammar, is used.

DEFINITION 10 [12]

- A grammar with no restrictions on the form of productions is called a *grammar of general type* or a grammar of *type 0*.

- A grammar such that all its productions are of the form

$$u\alpha v \rightarrow u\eta v, \quad u,v \in \mathsf{A}^*, \alpha \in \mathsf{N}, \eta \in \mathsf{A}^+ \text{ or } S \rightarrow \lambda$$

  if S is not at the right side of any other production is called a *context-sensitive grammar* or a grammar of *type 1*.

  A grammar, which differs from a context-sensitive grammar only in a more loose condition - $\eta \in \mathsf{A}^*$ imposed on η is called an *unrestricted context-sensitive grammar*.

- A grammar such that all its productions are of the form

$$\alpha \rightarrow \eta, \quad \alpha \in \mathsf{N}, \eta \in \mathsf{A}^*$$

  is called a *context-free grammar* or a grammar of *type 2*.

  A grammar of this type with an additional restriction - $\eta \neq \lambda$ is called a λ*-free* context-free grammar.

- A grammar such that all its productions are of the form

$$\alpha \rightarrow u\beta \text{ or } \alpha \rightarrow u, \ \alpha, \beta \in \mathsf{N}, u \in \mathsf{T}^*$$

$$(\alpha \rightarrow \beta u \text{ or } \alpha \rightarrow u, \ \alpha, \beta \in \mathsf{N}, u \in \mathsf{T}^*)$$

  is called a *right (or left) linear grammar* or a grammar of *type 3*. A grammar of this type with an additional restrictions that $u \in \mathsf{T}^+$ is called a λ*-free* right (or left) linear grammar.

The introduced types of grammars are the main types in the Chomsky Hierarchy. **Regular grammars** are used to generate elements of regular languages. There are two types of regular grammars, namely, right-linear grammar and left-linear grammar.

The diagram below represents the relations between the different classes of languages. Every arrow in the diagram represents an inclusion.

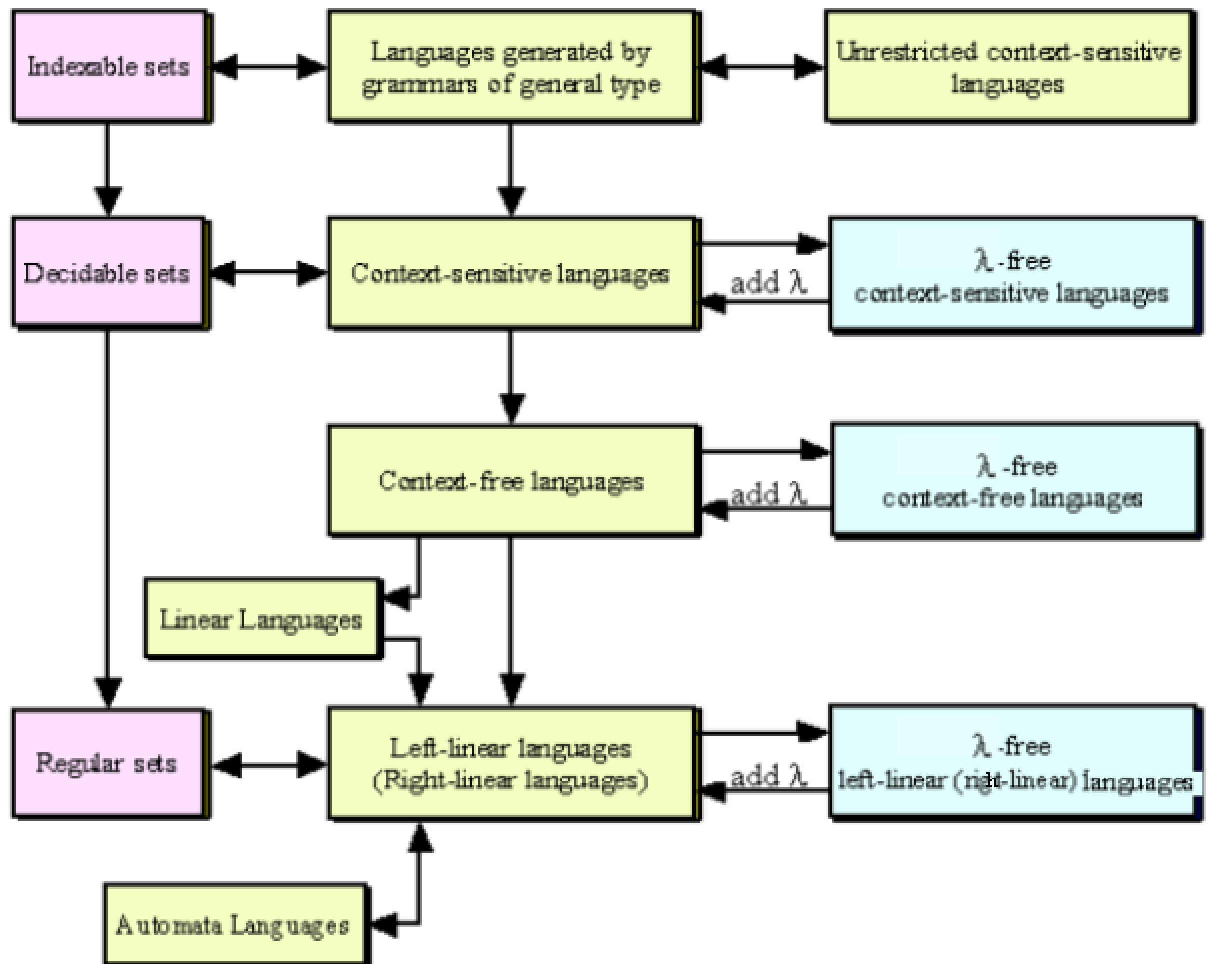


Figure 1: Relationship between different classes of languages

Some of the relations in the diagram follow directly from the definitions. Every left or right linear grammar is a linear grammar, every linear grammar is a context free grammar and every

context-free grammar is an unrestricted context-sensitive grammar. Further, every λ-free left or right linear grammar is a λ-free context-free grammar and every λ-free context-free grammar is a λ-free context-sensitive grammar. Finally, every grammar is a grammar of general type. Observe further, that in a context-sensitive language with λ-production if we eliminate λ only the empty string will be missing from the generated language. The rest of the relations shown in the diagram are based on the theorems below.

THEOREMS [3]

1. Every unrestricted context-sensitive grammar is equivalent to a grammar of general type.
2. Every context-sensitive language over an alphabet A is a decidable subset of A^*. The inverse is not true.
3. Every length increasing grammar is equivalent to a λ-free context sensitive grammar.
4. For every context-free grammar $\mathbf{G}$ with λ-productions a λ-free context-free grammar exists.
6. Every right linear grammar is equivalent to a left linear grammar.
7. For every right linear grammar $\mathbf{G}$ with λ-productions there is effectively a λ-free right linear grammar $\mathbf{G'}$.

2.2.3 Redundant symbols

A symbol in a Context Free Grammar is a redundant symbol if it can be removed from the alphabet of the grammar together with the productions in which it takes part without changing the language of the grammar. A symbol in a Context Free Grammar can be redundant for two

reasons: either it does not appear in any derivable string, hence the symbol is called an unreachable symbol, or even if it appears in some derivable string it does not play a part in the derivation of any terminal string, hence the symbol is called nonterminating symbol. Unreachable and nonterminating symbols are the redundant ones.

To determine the set of the terminating symbols in a Context Free Grammar $G=<A,T,R,S>$ we can use a procedure based on the following transitive property: if there is a derivation $\alpha \vdash w$ of length greater than 1, such that w is terminal then there is a production $\alpha \longrightarrow u$ and a derivation of lesser length $u \vdash w$. In this case all the symbols in u are terminating. Consider the following inductively defined sets.

$T_0 = T$

$T_1 = \{\alpha \in A \mid \alpha \vdash w \text{ is in } R \text{ for some } w \in T^* \} \cup T$

....

$T_i = \{\alpha \in A \mid \alpha \vdash w \text{ is in } R \text{ for some } w \in (T \cup T_{i-1})^* \} \cup T_{i-1}$

....

Intuitively, T_i contains all the symbols from which a terminal string can be derived by a derivation tree if height less than or equal to i. From the definition of the sets it follows directly that

$T_0 \subseteq T_1 \subseteq \ldots \subseteq T_i \subseteq \ldots$

On the other hand, since each T_i is a subset of A and A is a finite set the size of T_i cannot increase infinitely and therefore a smallest number k such that $T_k = T_{k-1}$ must exist.

Then the definition of the sets T_i shows that $T_i = T_{i-1}$ for any $i \geq k$. Therefore, T_k contains all terminating symbols.

For grammars of small size a simple process of labeling the symbols in the list of productions of the grammar can be used to determine the set of terminating symbols. To make the process of labeling more transparent and economical we will use two types of tokens: active – '*' and passive - '+'

- label all terminal symbols in the right side of the productions of the list with active tokens;
- label all the symbols on the left side of those productions of the list where all the symbols at the right side are labeled with tokens, with at least one of them being active with active tokens; change all the old tokens to passive;
- label in the right side of the productions of the list all the symbols which appear at the left side of a production labeled with active token; change all the old tokens to passive;
- repeat the last two operations until no active tokens appear;
- take all labeled symbols as terminating.

Similarly, to determine the set of all reachable symbols in a Context Free Grammar $G = <A, T, R, S>$ consider the following inductively defined sets.

$R_0 = \{S\}$

$R_1 = \{\alpha \in A \mid \beta \vdash u\ \alpha\ w$ is in R for some $\beta \in R_0$ and $u, w \in A^*\} \cup R_0$

....

$R_i = \{\alpha \in A \mid \beta \vdash u\ \alpha\ w$ is in R for some $\beta \in R_{i-1}$ and $u, w \in A^*\} \cup R_{i-1}$

....

Intuitively, R_i contains all the symbols reachable by a derivation of length less than or equal to i . From the definition of the sets it follows directly that

$$R_0 \subseteq R_1 \subseteq \ldots \subseteq R_i \subseteq \ldots$$

On the other hand, since each R_i is a subset of A and A is a finite set the size of R_i cannot increase infinitely and therefore there must be a smallest number k such that $R_k = R_{k+1}$. Then the definition of the sets R_i shows that $R_i = R_{i+1}$ for any $i \leq k$. Therefore, R_k contains all reachable symbols.

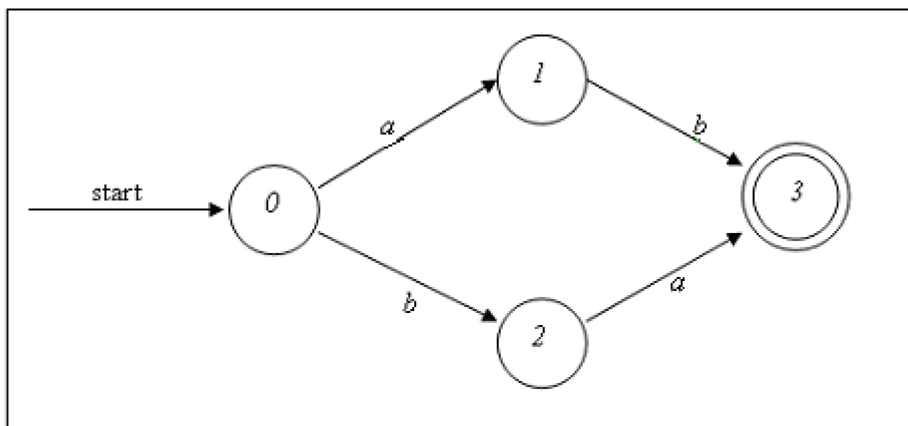
Again, a simple process of labeling of the symbols in the list of the grammar productions can be used to determine the set of reachable symbols for grammars of small size. We will use two types of tokens: active - '*' and passive - '+'. Notice, that the direction in which the tokens spread is the opposite of the direction in the previous procedure.

- label the initial nonterminal of the grammar with an active token;
- label in the right side of the productions of the list all the symbols which appear at the left side of a production labeled with an active token; change all the old tokens to passive;
- label all the symbols in the left side of those productions of the list all the symbols at the right side of which are labeled with tokens at least one of which is active; change all the old tokens to passive;

- repeat the last two operations until no active tokens appear;
- take all the labeled symbols as reachable.

Knowing the sets R of all reachable symbols and T of all terminating symbols the sets A\R of all unreachable symbols and the set A\T of all nonterminating symbols can be determined. Eliminating them and the productions in which they take part from the grammar we obtain an equivalent grammar with no redundant symbols.

As an example, the grammar corresponding to the finite acceptor:



can be represented using a table of production rules as follows:

Nonterminal	Production Rules	
0	a1	b2
1	b3	
2	a3	
3	λ	

Chapter 3

Related software

Many tools have been developed that allow the user to experiment with the representation of regular languages. Most of these toolkits are designed to simulate one of the three representations of Regular Languages, i.e., regular expressions, finite automata, or regular grammars. Another popular model is the Turing Machine, which is used as acceptors for recursively enumerable languages, of which regular languages are a subset. We will now present a list of some of the toolkits that have been created. The toolkits will be grouped into three categories, namely: complete toolkits (Section 3.1), which simulate all three models (and more), bi-model toolkits (Section 3.2), which simulate two of the models, and single model toolkits (Section 3.3), which simulate only the finite automaton.

3.1. Complete Toolkits

JFLAP [5][29][33][34][35] (Java Formal Languages and Automata Package) is a package of graphical tools that can be used both in teaching and learning the basic concepts of automata theory. It allows designing and running several variations of automata, such as deterministic finite automata (DFA), nondeterministic finite automata (NFA), regular expressions(RE), regular grammar (RG) and Turing Machines (TM), as well as conversions between the different representations.

Language Emulator [39], written in Java, allows the manipulation of regular expressions, regular grammars, deterministic finite automata, non-deterministic finite

automata with and without lambda transitions, as well as other machines such as the Moore and Mealy machines, which are transducers, and not related to regular languages at all.

3.2. Bi-Model Toolkits

Deus Ex Machine [36], developed by Nick Savoiue, comprises simulations of seven computation models. It was designed to be used in parallel with the book “Models of Computation and Formal Languages”. It provides a generic multi-purpose platform for designing and simulating different kinds of automata, including deterministic finite automata (DFA), nondeterministic finite automata (NFA). Nowadays, there are versions available for the Windows operating system only.

FIRE Engine is a finite automata toolkit, written in C++ by Bruce Watson [44]. It provides implementations of finite automata and regular expression algorithms. Several finite automata minimization algorithms have been implemented, including Hopcroft's O(n log n) algorithm. Both deterministic and non-deterministic automata are supported, and it has been used for compiler construction, hardware modeling, and computational biology applications. It is strictly a computing engine, and does not provide a graphical user-interface.

The *HaLeX* [48] library introduces a number of Haskell datatypes and the respective functions that manipulate them, providing the following facilities:

- The definition of deterministic finite automata, non-deterministic finite automata, and regular expressions directly and straightforwardly in Haskell.

- The definition of the acceptance functions for all those models.
- The transformation from regular expressions into non-deterministic finite automata (NFA) and from NDFA into deterministic finite automata (DFA).
- The minimization of the number of states of deterministic finite automata.
- Determines the equivalence of regular expressions and finite automata.
- The graphical representation of finite automata.
- The definition of Lex-like facilities, for example, it generates from a regular expression an efficient recognizer for the language: a Haskell program based on the equivalent minimized DFA.
- The automatic animation of the acceptance function of finite automata.

Grail+ [31] is a symbolic computation environment for finite-state machines, regular expressions, and other formal language theory objects. Using Grail+, one can input machines or expressions, convert them from one form to the other, minimize them, make them deterministic, find the complement, and perform many other operations. Grail+ enables you to manipulate parameterizable finite acceptors, parameterizable regular expressions, and parameterizable finite languages. By `parameterizable', we mean that the alphabet is not restricted to the usual twenty-six letters and ten digits. Instead, all algorithms are written in a type-independent manner, so that any valid C++ base type and any user-defined type or class can define the alphabet of a finite-state machine or regular expression.

AMoRE [49] is a program for the computation of **A**utomata, **Mo**noids, and **R**egular **E**xpressions. **AMoRE** is an implementation of automata theoretical algorithms. Sample functions are:

- conversion of regular expression into finite automata,
- determinisation and minimization of automata (including a heuristic minimization of nondeterministic automata),
- language operations (e.g. boolean and regular operations, quotients, shuffle product),
- tests (e.g. for set inclusion, nonemptiness),
- computation of the syntactic monoid and its algebraic decomposition,
- display of small automaton graphs on the screen.

Automata Editor [59]: is a toolkit for working with deterministic and non-deterministic finite automata, as well as (classic) regular expressions. Features include manually or programmatically creating and edit DFA's and NFA's, converting from NFA to DFA, viewing the created automata as a graph in real time as you edit, optimizing deterministic automata to the minimal number of states, converting from regular expressions to automata and constructing the equivalent regular expression from any NFA.

DFA to Regular Grammar [50] is a small Perl script that converts Deterministic Finite Automata into a Regular Grammars, showing states, and transitions.

3.3. Single Model Toolkits

The Java Computability Toolkit (JCT) [32] is a graphical environment to create and simulate deterministic finite automata (DFA), non-deterministic finite automata (NFA) and Turing Machines (TM). It was developed in Java, and it allows the user to minimize the DFA and to convert a NFA into a DFA. As in JFLAP, automata can be drawn on a canvas by positioning states and transitions. Finite automata (FA) can be executed on arbitrary input strings and executed in step-by-step fashion with immediate visual feedback.

FSA6: Finite State Automata Utilities Version 6 [47] (manual generated with FSA Utilities version fsa6-268) is a collection of utilities to: **construct** finite automata (from regular expressions), **manipulate** finite automata, **visualise** finite automata, **apply** finite automata.

RegeXeX: An Interactive System Providing Regular Expression Exercises [55] is a graphical tool used to help students learn to write regular expressions. Students are given prose descriptions of regular languages and asked to provide a regular expression that defines the language.

Finite automata tool (FAT) [56]: The FAT is a tool that implements and illustrates various algorithms on deterministic and non-deterministic finite automata including:

- determinization via the powerset construction

- minimization of deterministic finite automata
- intersection, union, and complement

All algorithms are visualized where one can either let the algorithms run automatically or in a step-by-step mode.

Visual Automata Simulator: This application is created by Jean Bovet from the University of San Francisco. According to the home website this simulator is created with the inspiration from an automata and object-orientation course [57]. It supports the construction of NFA’s and DFA’s, as well as the conversion from NFA to DFA.

jFAST [58] is a simple, easy-to-use tool for creating, editing, and simulating finite automata and state machines. It provides a simple and intuitive method of interacting with numerous types of supported finite state machines (FSM), including deterministic finite automata (DFA), and non-deterministic finite automata (NFA) among others. jFAST also provides user's with the ability to check their FSM on different inputs, and provides clear and meaningful messages when mistakes are encountered.

Chapter 4

Translation between models

In order for a language to be regular, it is essential that the language be definable by all these three representations. In chapter 2, definitions were given for a regular expression, a finite acceptor and a regular grammar. Regular expressions are used to define regular languages, finite acceptors are used to determine whether a string belongs to the language or not, and regular grammars are used to generate elements of the regular language. It is also pertinent to note that for a language that is defined in one of these models, there exists an equivalent language defined by any one of the other two models. In this chapter, we shall present techniques for translation from any one of the three models to any other. We shall also present a technique to translate from one sub-model of the Finite State Acceptor to another. For each translation, an algorithm is presented. An example of an implementation of the algorithm is then shown.

4.1 Translating from Regular Expression to Nondeterministic Finite Acceptor

The translation from a regular expression into a finite automaton is one of the oldest and most widely studied areas of computer science. Some of these techniques involve the translation from a regular expression into a Nondeterministic Finite Acceptor with λ transitions, some without λ transitions, while others translate the expression into a Deterministic Finite Acceptor, with some translating it into a minimal Deterministic Finite Acceptor. Thompson's

construction as presented in [37] constructs a possible Nondeterministic Finite Acceptor with λ transitions. Descriptions of this algorithm are given in various texts, such as [2, 3, 19, 44, 45], and these are considered more readable than Thompson's original paper. Berry and Sethi's construction as presented in [6, 15, 25] constructs a Nondeterministic Finite Acceptor. Berry and Sethi [6] relate this algorithm to Brzozowski's construction as presented in [9]. McNaughton, Yamada and Glushkov's construction as presented in [15, 25] produce a Deterministic Finite Acceptor. A mirror image of this construction is presented in [2] by Aho, Sethi and Ullman. Improvements to this algorithm were made by Bruggemann-Klein as presented in [8] and by Chang and Paige as presented in [13]. Brzozowski's construction as presented in [9] produces a Deterministic Finite Acceptor. From the review of the various techniques we concluded that the best method, and the method most used, for translating a regular expression into a finite state acceptor would be to first translate it into a nondeterministic finite acceptor. An algorithm to construct a nondeterministic finite acceptor from a regular expression is now given. The algorithm is syntax-directed in that it uses the syntactic structure of the regular expression to guide the construction process. The rules in the algorithm follow the rules in the definition of a regular expression. The process involves first showing how to construct an acceptor to recognise λ and any symbol in the alphabet. It is then shown how to construct an acceptor for expressions containing an alternation, concatenation, or Kleene closure operation. The construction process used is called the *Thompson's construction* [37].

4.1.1 Base Algorithm

Algorithm 1: (*Thompson's construction*) A nondeterministic finite acceptor from a regular expression

Input: A regular expression r over an alphabet Σ.

Output: A nondeterministic finite acceptor N accepting L (r).

Method: First r is parsed into its constituent sub expressions. Then, using rules (1) and (2) below, nondeterministic finite acceptors are constructed for each of the basic symbols in r. The basic symbols correspond to the first two points in definition 3 (pg. 15) of a regular expression. It must be noted that for each time a symbol a occurs in r, a separate nondeterministic finite acceptor is constructed.

Then, guided by the syntactic structure of the regular expression r, these nondeterministic finite acceptors are combined inductively using rule (3) below until the nondeterministic finite acceptor for the entire expression is obtained.

Rules:

1. For λ, construct the nondeterministic finite acceptor

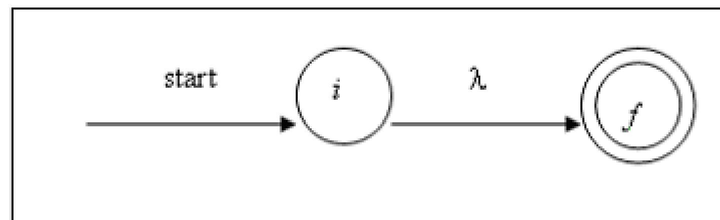


Figure 2: NFA representing the language λ

Here i is the new start state and f a new accepting state. Clearly, this nondeterministic finite acceptor recognises $\{\lambda\}$.

2. For a in Σ, construct the nondeterministic finite acceptor

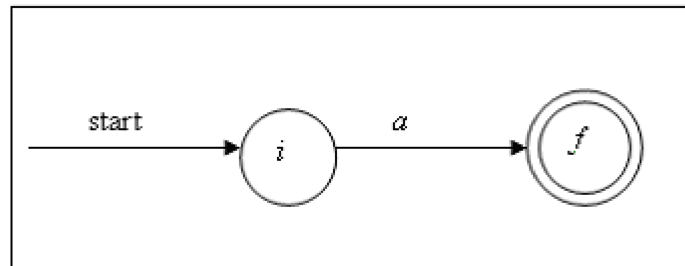


Figure 3: NFA representing the language *a*

Again i is the new start state and f a new accepting state. This machine recognises $\{a\}$ and will be extended to represent the entire regular expression.

3. Suppose N (s) and N (t) are nondeterministic finite acceptors for regular expressions s and t.

   a) For the regular expression s/t, construct the following composite nondeterministic finite acceptor N (s/t):

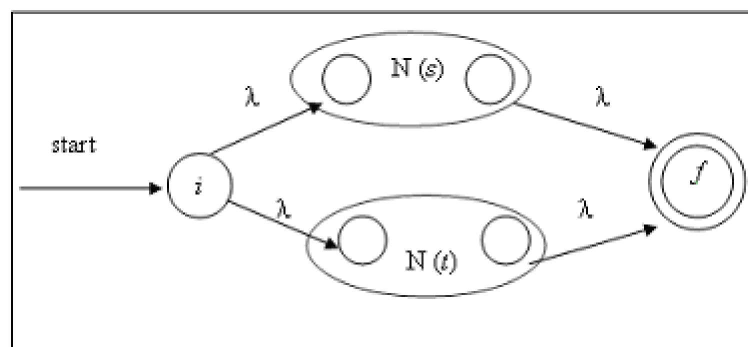


Figure 4: NFA representing the language N(*s/t*)

Here i is a new start state and f a new accepting state. There is a transition on λ from i to the start state of N (s) and N (t). There is a transition on λ from the accepting states

of N (*s*) and N (*t*) to the new accepting state *f*. The start and accepting states of N (*s*) and N (*t*) are not start and accepting states of N (s/t). Note that any path from *i* to *f* must pass through either N (*s*) or N (t) exclusively. Thus, the composite nondeterministic finite acceptor recognises L (*s*) / L (*t*).

b) For the regular expression s.t, construct the composite nondeterministic finite acceptor N(*st*):

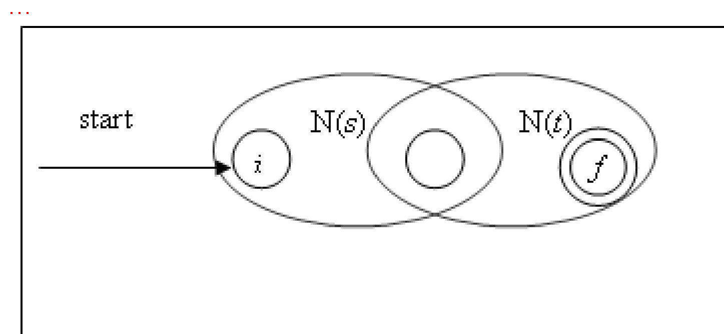


Figure 5: NFA representing the language N(*st*)

The start state of N(*s*) becomes the start state of the composite nondeterministic finite acceptor and the accepting state of N (*t*) becomes the accepting state of the composite nondeterministic finite acceptor. The accepting state of N(*s*) is merged with the start state of N (*t*), that is, all transitions from the start state of N (*t*) become transitions from all the accepting states of N(*s*). The new merged state loses its status as a start or accepting state in the composite nondeterministic finite acceptor. A path from *i* to *f* must go first through N(*s*) and then through N (*t*), so the label of that path will be a string in L(*s*)L (*t*). Since no edges enters the start state of N (*t*) or leaves the accepting state of N(*s*), there can be no path from *i* to *f* that travels from N (*t*) back to N(*s*).

c) For the regular expression *s**, construct the composite nondeterministic finite acceptor N(*s**):

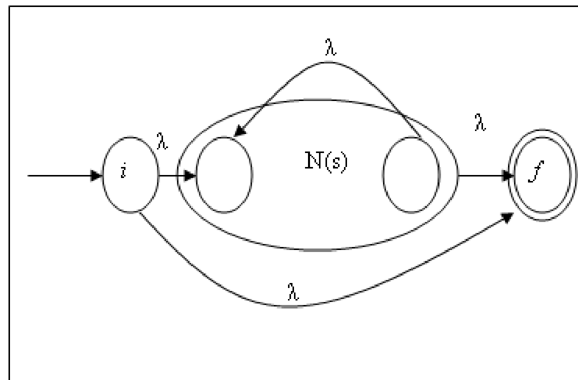


Figure 6: NFA representing the language N(*s)**

Here i is a new start state and f a new accepting state. In the composite nondeterministic finite acceptor, the transition can go from i to f directly, along an edge labelled λ, representing the fact that λ is in (L(*s*))*, or it can go from i to f passing through N(*s*) one or more times. Clearly, the composite nondeterministic finite acceptor recognises (L(*s*))*.

Every time a new state is constructed, it is given a distinct name. Thus, no two states of any component nondeterministic finite acceptor can have the same name. Even if the same symbol appears several times in *r*, for each instance of that symbol a separate nondeterministic finite acceptor with its own states is created. Finally, if more that one accepting state exists, a new state should be created, and a transition on λ from all accepting states to this new state should be created. The new state will then become to new single accepting state.

4.1.2 Shortcomings

Although the algorithm given is efficient in obtaining the desired output, it has a few shortcomings that need to be looked at. These shortcomings will first be listed, and their misgivings will be discussed. Measures to rectify these will then be suggested.

Consider a nondeterministic finite acceptor N(*r*) constructed from a regular expression:

- N(*r*) has at most twice as many states as the number of symbols and operators in *r*. This follows from the fact that for each step of the construction at most two new states are created on input λ.
- Each state of N(*r*) has either one outgoing transition on a symbol in Σ or at most two outgoing λ-transitions.

From the above it is noticed that because of the extensive use of the λ-transition, almost twice as many states as the number of symbols have resulted. This causes a lot of memory wastage. Also, by the definition of a deterministic finite acceptor, it is apparent that the λ-transitions are to be removed in the next stage of the translation so one may ask the question "Why have lambda transitions in the first place?" To answer this question, one needs to look at ways to convert a regular expression into a nondeterministic finite acceptor without the use of lambda transitions. Although no formal algorithm could be found, methods in this light were investigated and an algorithm is now proposed.

4.1.3 Proposed Algorithm

An algorithm is now proposed that is free from any lambda transitions. Like the previous algorithm, this algorithm is also syntax-directed. Since there are no lambda transitions, there is no need for the acceptor to recognise λ. The method for constructing an acceptor to recognise

any symbol in the alphabet will be shown first. Thereafter, methods for expressions containing the alternation, concatenation, or Kleene closure operator are constructed.

Algorithm 2: A nondeterministic finite acceptor from a regular expression

Input: A regular expression r over an alphabet Σ.

Output: A nondeterministic finite acceptor N accepting L (r).

Method: First r is parsed into its constituent subexpressions. Then, using rules (1) below, nondeterministic finite acceptors are constructed for each of the basic symbols in r. The basic symbols correspond to the second point in definition 3 (pg. 15) of a regular expression. Since there are no λ-transitions in the algorithm, the first point in definition 3 (pg. 15) of a regular expression can be ignored. It must be noted that for each time a symbol a occurs in r, a separate nondeterministic finite acceptor is constructed.

Then, guided by the syntactic structure of the regular expression r, the nondeterministic finite acceptors are combined inductively using rule (2) below until the nondeterministic finite acceptor for the entire expression is obtained.

Rules:

1. For a in Σ, construct the nondeterministic finite acceptor

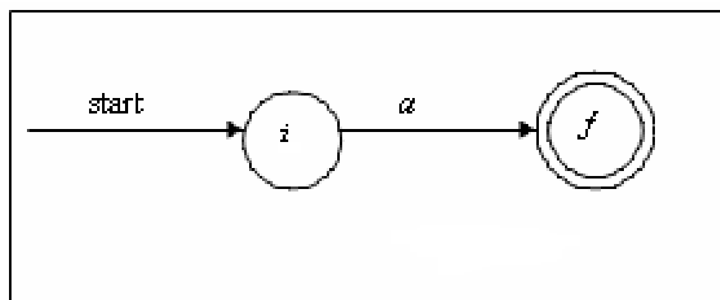


Figure 7: NFA representing the language *a*

Again *i* is the new start state and f a new accepting state. This machine recognises $\{a\}$.

2. Suppose N (*s*) and N (*t*) are nondeterministic finite acceptors for regular expressions *s* and *t*.

   a) For the regular expression *s/t*, construct the following composite nondeterministic finite acceptor N (*s/t*):

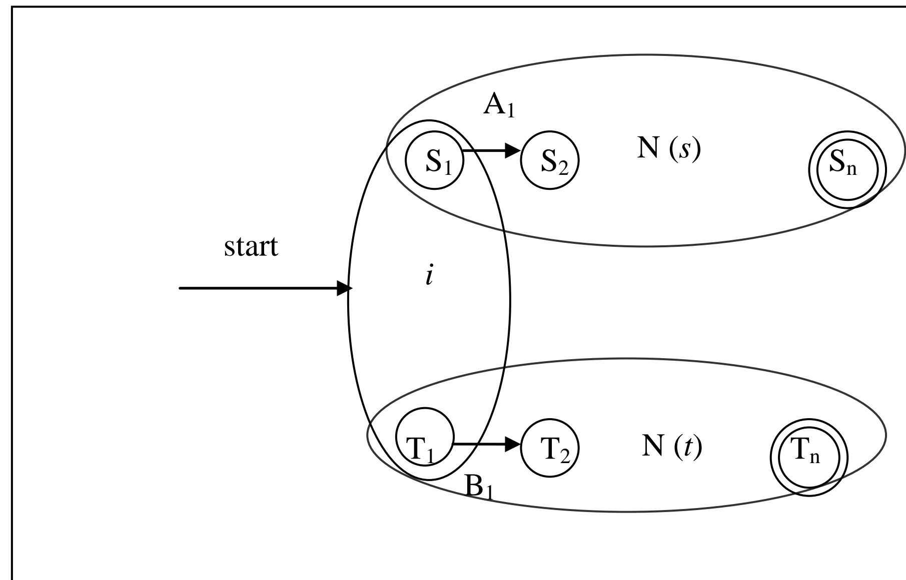


Figure 8: λ-free NFA representing the language N(*s/t*)

Here *i* is a new single start state consisting of both the start states of N(s) and N(t). Let A_1 be the transition from the start state of N(*s*) to the second state or set of second states of N(*s*), and B_1 be the transition from the start state of N(*t*) to the second state or set of second states of N(*t*). If the nondeterministic finite acceptor N(*s*) or N(*t*) is made up of only one state, then there is a transition on A_1 for N(*s*) or a transition on B_1 for N(*t*) from the start state to that state or set of states. This state/states will also be the accepting state for its respective nondeterministic finite acceptor. For a nondeterministic finite acceptor that contains more than one state, there is a transition on A_1 from *i* to the second state or set of second states of N (*s*) and a transition on B_1

from *i* to the second state or set of second states of N (*t*). The accepting states of N(*s*) and N(*t*) cannot be combined to form a single accepting state, and hence will be treated as two separate accepting states. Note that any path from *i* to an accepting state must pass through either N (*s*) or N (t) exclusively. Thus, the composite nondeterministic finite acceptor recognises L(*s*) / L(*t*).

b) For the regular expression *st*, construct the composite nondeterministic finite acceptor N (*st*):

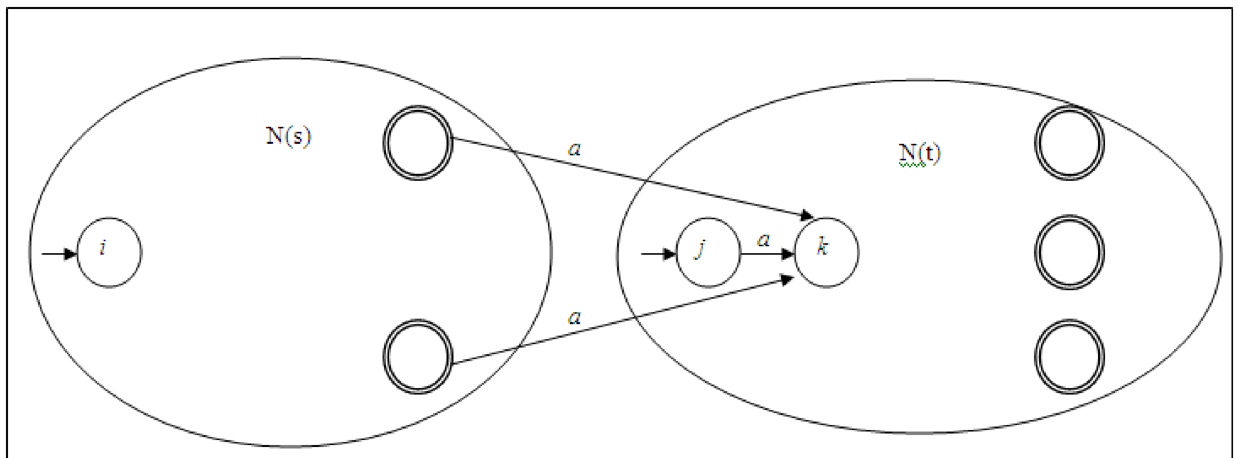


Figure 9: λ-free NFA representing the language N(*st*)

The start state of N(*s*) becomes the start state or set of states of the composite nondeterministic finite acceptor and the accepting state or set of states of N(*t*) becomes the accepting state or set of states of the composite nondeterministic finite acceptor. The accepting state or set of states of N(*s*), is merged with the start state of N(*t*), that is, all transitions from the start state of N(*t*) become transitions from all the accepting states of N(*s*). The new merged state loses its status as a start or accepting state in the composite nondeterministic finite acceptor. A path from *i* must go first through N(*s*) and then through N(*t*), so the label of that path will be a string in L(*s*)L(*t*). Since no

edges enter the start state of N(t) nor leave the accepting state or set of states of N(s), there can be no path from i that travels from N(t) back to N(s). Thus, the composite nondeterministic finite acceptor recognises L(s)L(t).

c) For the regular expression s*, construct the composite nondeterministic finite acceptor N(s*):

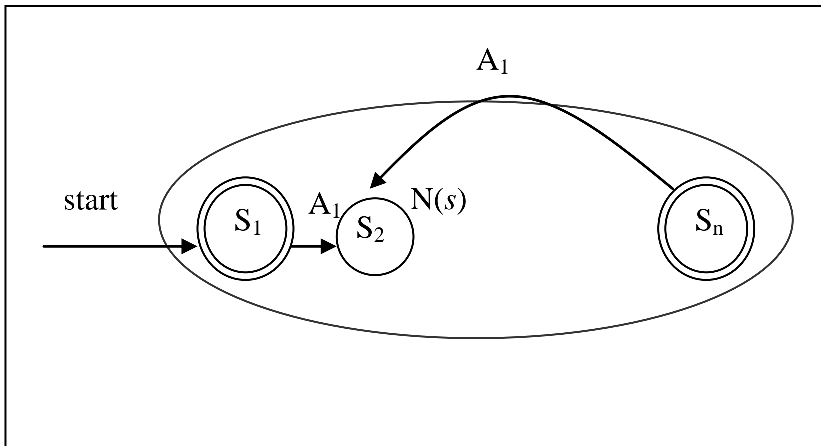


Figure 10(a) : λ-free NFA representing the language N(*s), where the number of states of N(*s**) is greater than two**

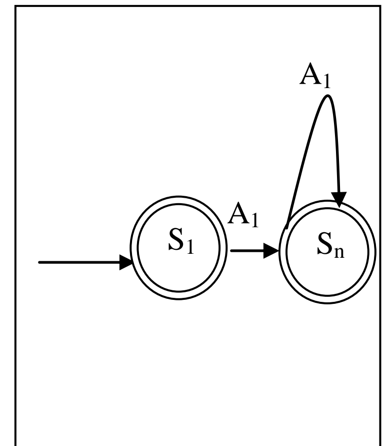


Figure 10(b) : λ-free NFA representing the language N(*s), where the number of states of N(*s**) is equal to two**

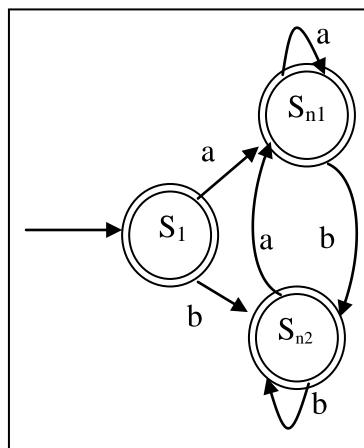


Figure 10(c) : λ-free NFA representing the language N(*(a/b))**

Here S_1 is a start state and an accepting state of N(*s*)*. In the composite nondeterministic finite acceptor, shown in Figure 6, transitions go from *i*, the start state, to *f*, the accepting state, directly, along an edge labelled λ, representing the fact that λ is in (L(*s*))*, or it can go from *i* to *f* passing through N(*s*) one or more times. Here, since λ-transitions are not used, an operation of zero occurrences of N(*s*) merely remains in the start state, which is also the accepting state of N(*s*)*. For the case where one or more occurrences of N(*s*) occur, transitions can go from S_1 to S_n passing through N(*s*), and then back to the second state of N(S), i.e. all states that lead from the start state, on the transition A_1, i.e., the same transition as from the start state S_1 to the second state S_2. Thus it can pass through N(*s*) any number of times. Clearly, the composite nondeterministic finite acceptor recognises (L(*s*))*.

d) For the parenthesised regular expression (*s*), use N(*s*) itself as the nondeterministic finite acceptor.

Every time a new state is constructed, it is given a distinct name. In this way, no two states of any component nondeterministic finite acceptor can have he same name. Even if the same symbol appears several times in *r*, for each instance of that symbol a separate nondeterministic finite acceptor is created with its own states.

4.1.4 Advantages of proposed algorithm

From the above, we notice that there is an addition of a state only for the case when the union operation is performed, see Figure 8 (pg.43). For all the other operations the number of states and the number of symbols are equal. Thus fewer states are being used in this algorithm than

were used in Thompson's algorithm. This will therefore result in a lot less memory being used up. In the case of the union operation, one may ask why the transitions from the start state to the second state of the nondeterministic finite acceptor are being duplicated. This is done because there may be a case where the start state of the nondeterministic finite acceptor will have a transition coming into the state. If this is the case then using this state as the start state, an extra operation will be created, which would be similar to the Kleene closure operation. This is best illustrated in the following example:

Consider the expression (a.b.c/(d.e)*). Using the method proposed, the nondeterministic finite acceptor is as follows:

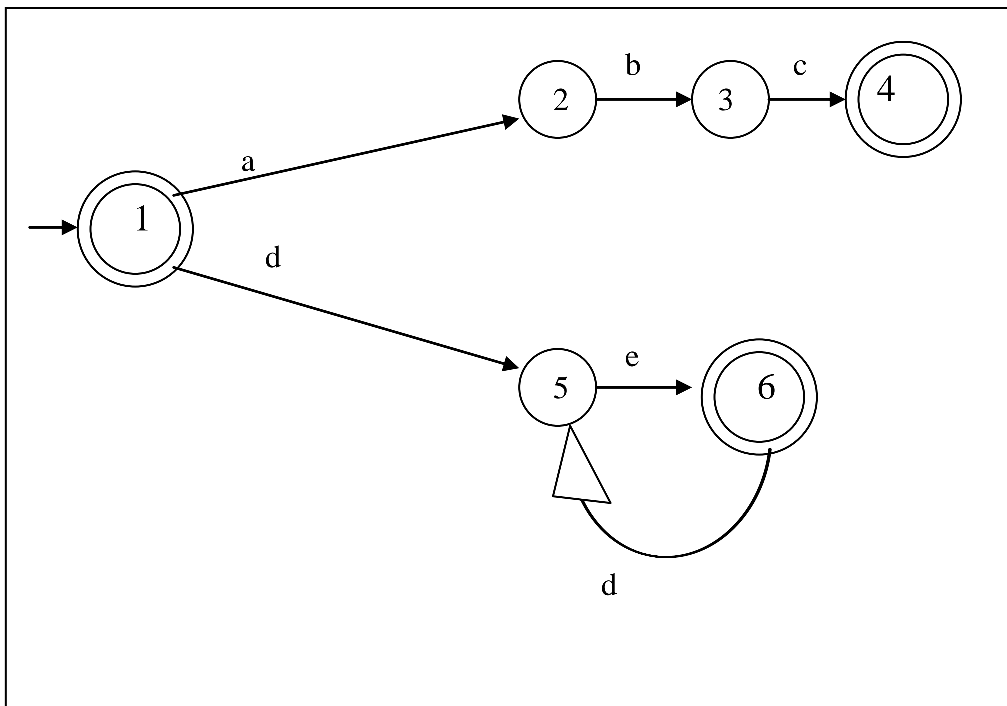


Figure 11: λ-free NFA representing the language a.b.c/(d.e)*

Note that if this "*method of parallelism*" were not used, then the nondeterministic finite acceptor would have looked as follows:

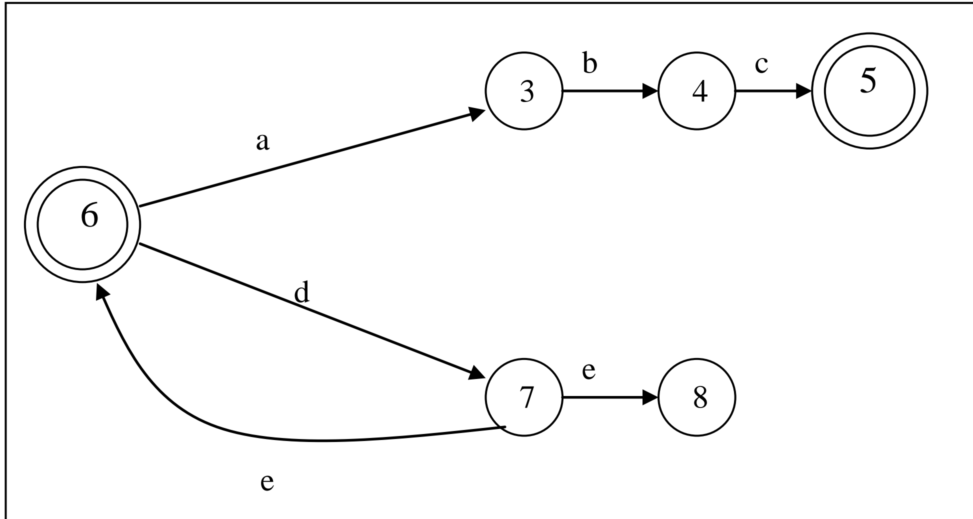


Figure 12: λ-free NFA representing the language (d.e)* / (d.e)*.a.b.c

In this case the nondeterministic finite acceptor defines a language (d.e)* / (d.e)*.a.b.c However this is not the language defined by the regular expression. Thus this is not a correct representation of the original regular expression. The method of parallelism is therefore justified. We also noticed that if cases like the above one do not occur, then the "extra" states will be removed in the next stage, i.e. when the nondeterministic finite acceptor is translated into a deterministic finite acceptor. Another important observation is that the nondeterministic finite acceptor can have a maximum of one start state, but it could have any number of accepting states. This does not cause any problem since by the definition of a nondeterministic finite acceptor F is defined as a set of accepting states. This implies that there could be more than one accepting state.

An example is now given indicating the difference between the two algorithms. Consider the regular expression defined by the language (a.b.c.d*/a.b.d*) The nondeterministic finite acceptor using each of the algorithms are represented below.

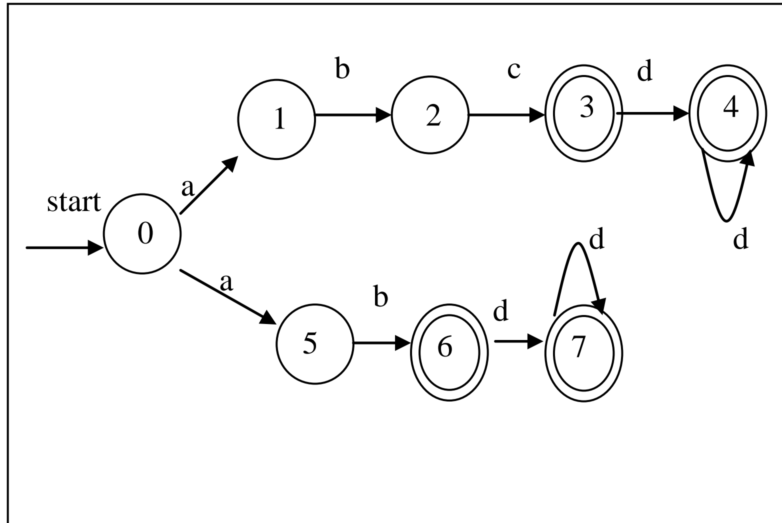


Figure 13(a): λ-free NFA representing the language a.b.c.d*/a.b.d*

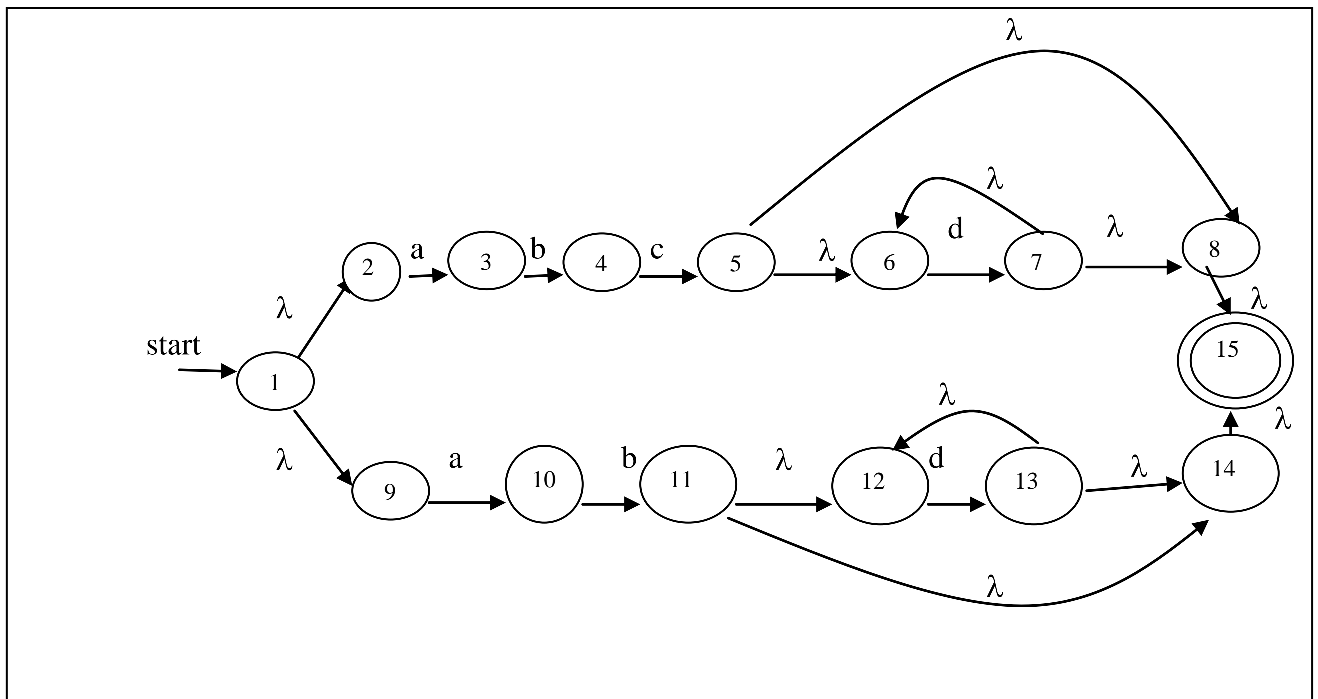


Figure 13(b) : NFA with λ transitions representing the language a.b.c.d*/a.b.d*

The nondeterministic finite acceptor for Algorithm 2 (pg 47) is shown in Figure 13(a) and the nondeterministic finite acceptor of the Algorithm 1 (pg 42) is shown in Figure 13(b).

From the figures it is apparent that the nondeterministic finite acceptor for Algorithm 2 (pg 47) contains half the number of nodes as the nondeterministic finite acceptor for Algorithm 1 (pg 42). Also it should be noted that both the algorithms produce a nondeterministic finite acceptor that recognises the specified language for the regular expression.

From the above we can postulate that all the necessary requirements for the nondeterministic finite acceptor are properly met and that the proposed algorithm will translate from a regular expression into a nondeterministic finite acceptor correctly. One of the reasons for the introduction of lambda transitions in Algorithm 1 is to ensure there is at most one start state and one final state at all times during the conversion process. As will be seen in Algorithm 3 (pg. 56), even though Algorithm 2 produces a nondeterministic finite acceptor with possibly more than one final state, this does not affect the conversion form a nondeterministic finite acceptor to a deterministic finite acceptor. It therefore appears that Algorithm 2 performs more efficiently than the one containing lambda transitions and thus is a better algorithm to use.

4.2 Translation from Nondeterministic Finite Acceptor to Deterministic Finite Acceptor

Two methods for translating a regular expression into a nondeterministic finite acceptor were presented in the previous section. We notice that the nondeterministic finite acceptor has a few states that contain more than one transition from that state on a single input. An example of this is in Figure 13(a) (pg. 53), where state 0 has two transitions from it on input a, and in

Figure 13(b) (pg. 54), state 1 has three transitions on input λ coming out of it. Since these are multivalued transition functions, it is difficult to simulate the nondeterministic finite acceptor in a computer program.

An algorithm is now presented for the construction of a deterministic finite acceptor from a nondeterministic finite acceptor that recognises the same language. This algorithm, often called the *subset construction,* as presented by Rabin and Scott in [30], is useful for simulating a nondeterministic finite acceptor by a computer program. The difference between the nondeterministic finite acceptor and the deterministic finite acceptor is that, firstly, the nondeterministic finite acceptor could contain transitions on λ, but this has been removed in the proposed algorithm, and secondly, in the transition table of a nondeterministic finite acceptor, each entry is a set of states, while in the transition table of a deterministic finite acceptor, each entry is just a single state. The general idea behind the nondeterministic finite acceptor-to-deterministic finite acceptor construction is that each deterministic finite acceptor state corresponds to a set of nondeterministic finite acceptor states. The deterministic finite acceptor uses its state to keep track of all possible states the nondeterministic finite acceptor could be in after reading each input symbol. It should be noted that the number of states for the deterministic finite acceptor can be much larger than the number of states of the nondeterministic finite acceptor, but in practise this worst case rarely occurs.

4.2.1 Base Algorithm

Algorithm 3: *(Subset Construction)* A deterministic finite acceptor from a nondeterministic finite acceptor [30]

Input: A nondeterministic finite acceptor *M* accepting L(*r*)

Output: A deterministic finite acceptor *M'* accepting L (*r*).

Method: The idea of the algorithm is to use a given nondeterministic finite acceptor *M* as the basis for the construction of a new deterministic finite acceptor *M'* that accepts the same language. The essence of the construction of *M'* is in the rules below.

Rules:

1) The states of *M'* will correspond to the nonempty subsets of the "states of *M*" = Q say. In what follows, it will be easiest to merely *identify* the states of *M'* with nonempty sets of states of *M*. With this understanding, the set of states of *M'* will sometimes be written as S_i, S_j, and so on.
2) Where q_0 is the start state of *M*, the singleton $\{q_0\}$ shall be designated as the start state of *M'*.
3) The accepting states of M' will be those sets of states containing at least one accepting state of *M*.
4) Finally, for each set of states *S* given by (1) and for each *s* in *M'* alphabet, draw an arc labelled *s* from the set of states to that state, call it $S_{s\text{-}succ}$, consisting of all the *s*-successors of members of *S*.
5) Eliminate any set of states, as well as all arcs incident upon it, such that there is no path leading to it from $\{q_0\}$.
6) Create a new state called the empty state, and for each state, if there is no transition from that state on an input, create a transition from that state on that input to the empty state.

4.2.2 Compatibility

Now that the algorithm for translating from a regular expression to a nondeterministic finite acceptor has been changed, it is essential that the resultant nondeterministic finite acceptor be compatible with the algorithm above. Probably the best way to describe the compatibility of the above algorithm is to use an example and show how it will work for all cases. Consider the example introduced in the previous section. The finite state acceptor, *M*, accepts the language denoted by (a.b.c.d*/a.b.d*). *M* is the 6-state nondeterministic finite state acceptor of Figure 13(a) (pg. 53). It should be noted that the algorithm above does not accommodate for λ-transitions. For nondeterministic finite acceptors with λ-transitions, the algorithm would first need to compute the λ-closure, i.e. all the states that the current state can reach using the λ-transitions for a given input. It can be seen from this that the methods that use λ-transitions do in fact waste time and memory since λ-transitions are first inserted into the nondeterministic finite acceptor and then deleted from the deterministic finite acceptor. The nondeterministic finite acceptor produced by the proposed algorithm, which does not contain lambda transitions, will therefore be used. The state set is defined by Q= { 0, 1, 2, 3, 4, 5}, with 3 and 5 as accepting states. By rule (1), the states of *M′* will now be the 63 nonempty subsets of *Q*. The first 12 state sets are shown below.

{0}

{1}

{2}

{3} **

{4}

{5} **

{0, 1}

{0, 2}

{0, 3} **

{0, 4}

{0, 5} **

{1, 2}

Similarly, all the state sets for the 63 subsets are defined. By rule (3), each of the subsets marked with ** will be accepting states of M' since states 3 and 5, which is in each one of these subsets, are accepting states of the nondeterministic finite acceptor. Finally, since it is desired that M' be fully defined and deterministic, it is required that the single arc labelled by a and a single arc labelled by b emanate from each of the state-nodes in the state diagram of M'. A few examples of these will be shown. Consider the state {0}. From the state diagram of M it can be seen that 0 has a-successors 1 and 4. So by rule (4), the a-arc from state {0} has an a-successor to state {1,4} in the new state diagram for M'. As for the b-arc from state {0}, it can be seen that 0 has no b-successor. Again by rule (4), {0} has a b-arc to state {Φ}, i.e. the empty state. In a similar manner all the other state transitions are defined. Since the deterministic finite acceptor will have 63 subsets, we cannot layout all the transitions for all of the state sets. Also, by rule (5), state sets that cannot be reached from the start state will be eliminated. A method is now proposed for finding only those state sets that will emanate from the start states, i.e. only those required by the transition diagram. The method is as follows:

1) Start at the start state set.
2) From the current state set find all transitions from this state set using all possible inputs.
3) For each of the new state sets found in step (2), apply step (2) with the new state set as the current state set.

4) Repeat steps (2) and (3) until the transitions for all state sets defined have been found.

Using the example above, the transition table using this method is shown in Figure 14.

State	Input Symbol			
	a	b	c	d
{0}	{1,4}	-	-	-
{1,4}	-	{2,5}	-	-
{2,5}	-	-	{3}	{5}
{3}	-	-	-	{3}
{5}	-	-	-	{5}

Figure 14: Transition table for the DFA representing the language (a.b.c.d*/a.b.d*)

The final deterministic finite acceptor is shown in Figure 15 below.

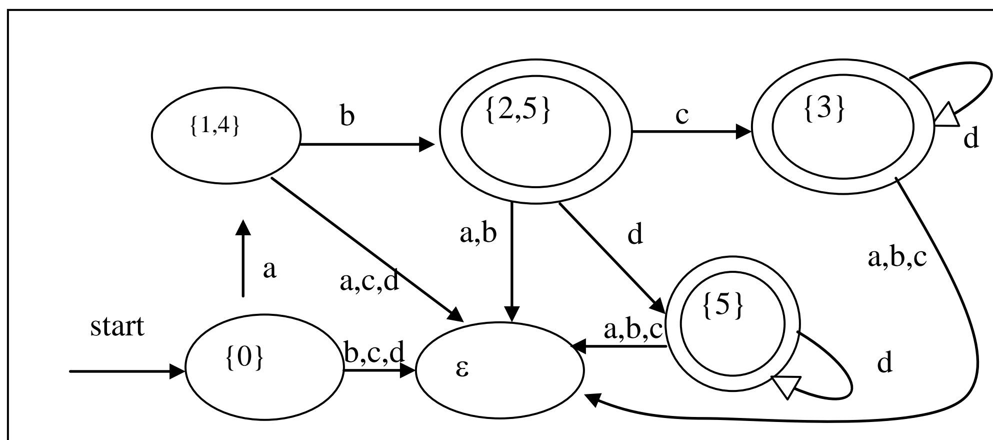


Figure 15: State diagram for the DFA representing the language (a.b.c.d*/a.b.d*)

From the above figures it is apparent that M' is a fully defined deterministic finite acceptor. There is at most one arc for *a,* one arc for *b* from each node, etc. This will be the case for any nondeterministic finite acceptor-to-deterministic finite acceptor transition. Thus, we can conclude that for any nondeterministic finite acceptor obtained using the proposed algorithm, the algorithm given will translate the nondeterministic finite acceptor into the deterministic finite acceptor that defines the language, and only that language that defined the nondeterministic finite acceptor.

4.3 Minimising the Number of States of a Deterministic Finite Acceptor

An important theoretical result is that every regular set is recognised by a minimum-state deterministic finite acceptor that is unique up to state names. In this section, a method for constructing this minimum-state deterministic finite acceptor is given. The number of states in the given deterministic finite acceptor are reduced to a minimum without affecting the language that is being recognised. The problem of minimising a Deterministic Finite Acceptor has been studied since the early 1950's. A number of different algorithms have been presented, with most textbooks presenting their own variation of the algorithms. The algorithm presented by Hopcroft in [18] is considered to be the one with the best running time, but this algorithm is "obscure and difficult to understand" [40]. According to Watson [40], "most textbook authors claim that their minimization algorithm is directly derived from those presented by Huffman [20]. Unfortunately most textbooks present vastly differing algorithms (for example compare [4], [2], [19], and [45]), and only the algorithms presented by Aho and

Ullman and by Wood are directly derived from those originally presented in [20]". Another algorithm was presented by Brzozowski in [9] , which was also presented in [38]. Moore, Wood, Brauer and Urbanek presented a layerwise computation of equivalence class algorithm in [45, 7] respectively. Unordered computation of equivalence algorithms were presented in [2] and [40], with improved unordered computation of equivalence class algorithms also presented in [40]. Huffman and Moore presented algorithms in [20], and Hopcroft and Ullman presented an algorithm based on Huffman and Moore's algorithm in [19]. The best known algorithm for minimization was presented by Hopcroft in [18] and a more simple variation of this algorithm was presented by Gries in [17].

Suppose that a deterministic finite acceptor M with set of states S and input symbol alphabet A is given. It is assumed that every state has a transition on every input symbol. If this is not the case, introduce a new "empty state" Φ, with transitions from Φ to Φ on all inputs, and add a transition from state s to Φ on input a if there is no transition from s on a.
A string w is said to distinguish state s from state t if, by starting with the deterministic finite acceptor M in state s and feeding it input w, an accepting state is reached, but starting in state t and feeding it input w, a nonaccepting state is reached, or vice versa. The algorithm for minimising the number of states of a deterministic finite acceptor works by finding all groups of states that can be distinguished by some input string. Each group of states that cannot be distinguished is then merged into a single state. The algorithm works by refining and maintaining a partition of the set of states. Each group of states within the partition consists of states that have not yet been distinguished from one another, and any pair of states chosen from different groups have been found distinguishable by some input.

Initially the partition consists of two parts, the accepting states and the nonaccepting states. The fundamental step is to take some group of states, say Z = {s_1, s_2,s_3 ,... ,s_k} and some input symbol *a*, and look at what transitions states s_1, …, s_k have on input *a*. If these transitions are to states that fall into two or more different groups of the current partition, *A* must be split up so that the transitions from all subsets of *A* are all confined to a single group of the current transition. Suppose, for example, that s_1 and s_2 go to state t_1 and t_2 on input *a*, and t_1 and t_2 are different groups of the partition. Then *A* must be split up into at least two subsets so that one subset contains s_1 and the other s_2. Note that t_1 and t_2 are distinguished by some string *w*, so s_1 and s_2 are distinguished by string *aw*.

The process of splitting the groups in the current partition is repeated until no more groups need to be split up.

4.3.1 Base Algorithm

Algorithm 4: Minimising the number of states of a deterministic finite acceptor. [2, 27]

Input: A deterministic finite acceptor *M* with set of states *S*, set of inputs *A*, transitions defined for all states and inputs, start state s_0, and set of accepting states *F*.

Output: A deterministic finite acceptor *M'* accepting the same language as *M* and having as few states as possible.

Method:

1) Construct an initial partition Π of the set of states with two new groups, the accepting states *F* and the nonaccepting states *S* - *F*.

2) Apply the following procedure to Π to construct a new partition Π_{new}.

for each group G of Π **do**

partition G into subgroups such that two states *s* and *t* of G are in the same subgroup if and only if for all input symbols *a*, states *s* and *t* have transitions on *a* to a state in the same group of Π;

/* at worst, a state will be in a subgroup by itself */

replace G in Π_{new} by the set of all subgroups formed.

end

3) If $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with $\Pi=\Pi_{new}$.

4) Choose one state in each group of the partition Π_{final} as the *representative* for the group. The representative will be the state of the reduced deterministic finite acceptor *M'*. Let *s* be a representative state, and suppose on input *a* there is a transition of *M* from *s* to *t*. Let *r* be the representative of *t*'s group (*r* may be *t*). Then *M'* has a transition from *s* to *r* on *a*. Let the start state of *M'* be the representative of the group containing the start state s_0 of *M*, and let the accepting states of *M'* be the representative of the states that are in *F*. Note that each group of Π_{new} either consists only of states in *F* or has no states in *F*.

5) If *M'* has an empty state, that is, a state Φ that is not accepting and that has transitions to itself on all input symbols, or an unreachable states, i.e. a state Φ that cannot be reached from the start state, then remove it from *M'*. Any transitions to Φ from other states become undefined.

4.3.2 Application

We now show an example of how this algorithm works. Reconsider the deterministic finite acceptor represented in Figure 15. The initial partition Π consists of two groups, ({2,5}, {3}, {5}), the accepting states, and ({0}, {1,4}), the nonaccepting states. To construct Π_{new}, the algorithm first considers ({2,5}, {3}, {5}). On input d, each of these has a transition to itself or to a state set that is in this subgroup, so they could remain in one group as far as input b is concerned. On input c, only {2,5} has a transition to {3}, while the others have a transition to the empty state, a member of another group. Thus the group ({3}, {5}, {2,5}) must be split up into two new groups ({3}, {5}) and ({2,5}). Π_{new} is thus ({0}, {1,4}) ({2,5}) ({3}, {5}).

The process is now repeated for the new Π. The subgroup for the accepting state remains unchanged as before, thus add this subgroup to Π_{new}. Considering the subgroup ({0}, {1,4}), it is noticed that on input b state set {1,4} has a transition to {2,5}, which is now a member of another subgroup. Thus group ({0}, {1,4}) must be split up into two new groups ({0}) and ({1,4}). Π_{new} now becomes ({0}) ({1,4}) ({2,5}) ({3}, {5}). The process is repeated one more time, but it is found that all the subgroups are reduced to their minimum. Thus Π_{final} will be the subgroups ({0}) ({1,4}) ({2,5}) ({3}, {5}), where the underlined state sets are the representative state sets for each subgroup. The transition table for the minimal deterministic finite acceptor is shown in Figure 16.

State	Input Symbol			
	a	b	c	d
{0}	{1,4}	-	-	-
{1,4}	-	{2,5}	-	-
{2,5}	-	-	{3}	{3}
{3}	-	-	-	{3}

Figure 16: Transition table for the minimal DFA representing the language

(a.b.c.d*/a.b.d*)

The equivalent minimal deterministic finite acceptor is shown in Figure 17 below. From the reduced acceptor above, it is noticed that state {2,5} has a transition on d to state {3}. This is because state {3} is the representative of the group for {5} and there is a transition from {2,5} to {5} on d in the original acceptor. Also, there is no empty state in Figure 17, and all states are reachable from the start state {0}.

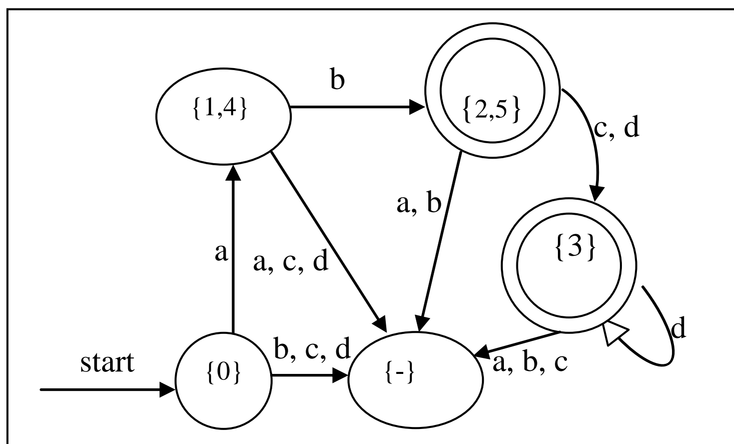


Figure 17: State diagram for the minimal DFA representing the language

(a.b.c.d*/a.b.d*)

4.4 Translating from Deterministic Finite Acceptor into a Regular Expression

The constructions in the previous sections demonstrated that every regular expression is recognised by a finite state acceptor. It was also demonstrated that any nondeterministic finite acceptor can be reduced into a similar deterministic finite acceptor. The deterministic finite acceptor, which is the simplified form of the finite state acceptor, is now to be converted back into a regular expression using the McNaughton and Yamada construction as presented in [25]. For this to be done, the notion of a state diagram needs to be extended.

4.4.1 Base Algorithm

An *expression graph* is a labelled directed graph in which the arcs are labelled by regular expressions. An expression graph, like a state diagram, contains a distinguished start node and a set of accepting nodes.

The state diagram of an acceptor with alphabet Σ can be considered an expression graph. The labels consist of lambda and expressions corresponding to the elements of Σ. Arcs in an expression graph can also be labelled by Φ, i.e. the empty state. Paths in an expression graph generate regular expressions. The language of an expression graph is the union of the sets represented by the accepted regular expressions. The state diagram of a finite state acceptor may have any number of accepting states. Each of these states exhibits the acceptance of a set of strings, the strings whose processing successfully terminates in the state. The language of the machine is the union of these sets. To determine the language of an acceptor, the previous observation allows one to consider the accepting states separately. The

algorithm to construct a regular expression from a state diagram does exactly this, i.e. it builds an expression for the set of strings accepted by each individual accepting state. The label of an arc from node *i* to node *j* is denoted $w_{i,j}$. If there is no arc from node *i* to *j*, $w_{i,j} = \Phi$.

Algorithm 5: Constructing a regular expression from a deterministic finite acceptor. [2, 25]

Input: A state diagram G of a finite acceptor, with nodes numbered 1,2,…,*n*.

Output: A regular expression L (G).

Method: Let the number of accepting states in the deterministic finite acceptor be *m*. Make *m* copies of G, each of which has one accepting state. Call these graphs G_1, G_2, … , G_m. Each accepting node of G is the accepting node of G_t, for some $t = 1, 2, \ldots, m$.

For each G_t repeat the following procedure:

1. choose a node *i* in G_t that is neither the start nor the accepting node of G_t.
2. delete the node *i* from G_t according to the following rules:

   for every *j, k* not equal to *i* (including when $j = k$) **do**

   repeat

   i) **if** $w_{j,i} \neq \Phi$, $w_{i,k} \neq \Phi$, and $w_{i,i} = \Phi$, **then** add an arc from node *j* to node *k* labelled $w_{j,i}w_{i,k}$.

   ii) **if** $w_{j,i} \neq \Phi$, $w_{i,k} \neq \Phi$, and $w_{i,i} \neq \Phi$, **then** add an arc from node *j* to node *k* labelled $w_{j,i}(w_{i,i})^*w_{i,k}$.

   iii) **if** nodes *j* and *k* have arcs labelled w_1, w_2, … , w_s connecting them **then** replace them by a single arc labelled w_1 / w_2 / … / w_s, where "/" implies union.

   iv) Remove the node *i* and all incident to it in G_t.

   until the only nodes in G_t are the start node and the single accepting node.

Determine the expression accepted by G_t.

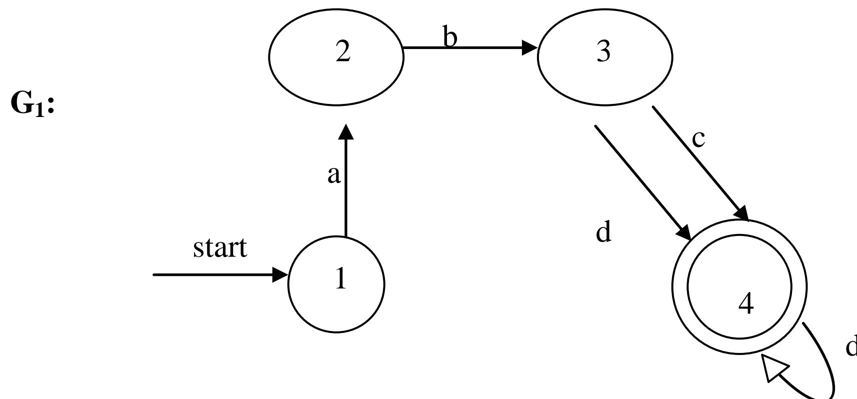
The regular expression accepted by G is obtained by joining the expressions for each G_t with "/".

4.4.2 Application

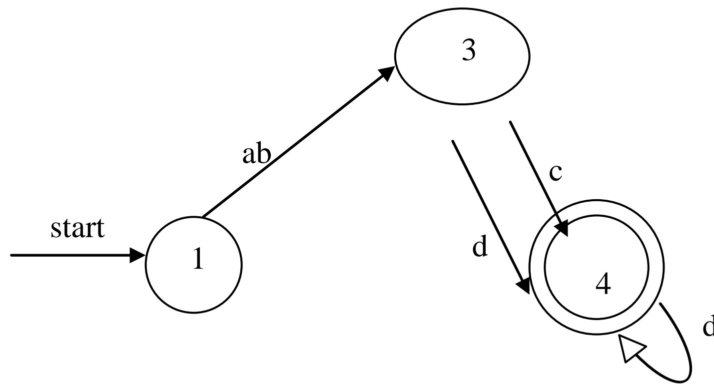
Once again the best way to show the application of the above algorithm is to use an example and show how it will work for all other cases. Consider the deterministic finite acceptor obtained in the previous section. The state diagram, G, is shown in Figure 17 (pg. 66). Since there are two accepting state sets, two expression graphs are constructed, each with a single accepting node.

Note that the states are numbered 1, 2, 3, 4, and 5. This is essential in the construction of the expression graph.

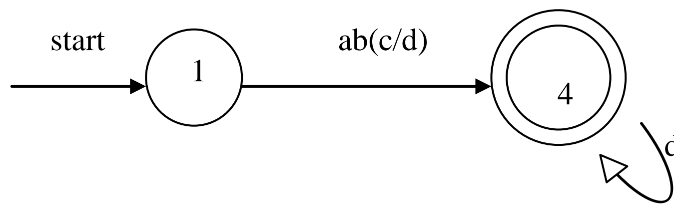
G_1:



Reducing G_1 consists of deleting node 2.

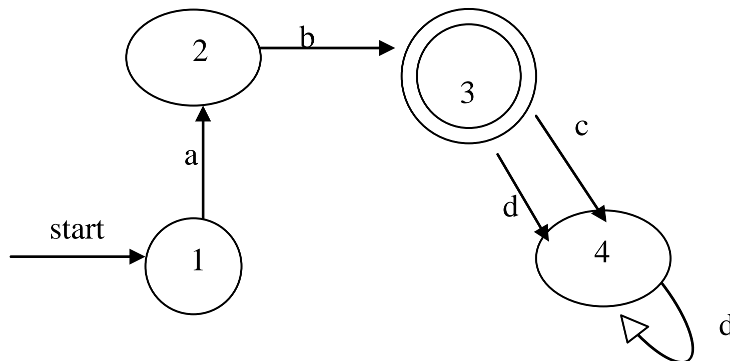


Further reduction consists of removing node 3.

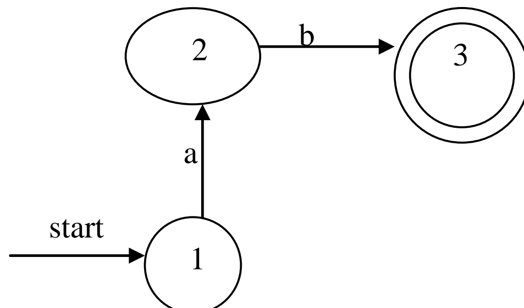


The expression excepted by G_1 is a.b(c/d)d*.

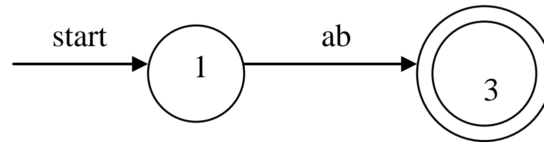
G_2:



Reducing G_2 consists of removing node 4.



Further reduction consists of removing node 2.



The expression accepted by G_2 is a.b.

The expression accepted by G, built from the expressions accepted by G_1 and G_2, is a.b/a.b(c/d)d*.

4.5 Translating from Regular Expression into a Regular Grammar

The previous sections presented methods for translating between regular expressions and finite automaton. The third important way of describing a language is by specifying the grammar rules by which strings in the language can be constructed. We will now show that the language defined by a regular expression can also be defined by a regular grammar. To do this, we will translate the language defined by a regular expression into an equivalent language defined by a regular grammar. Algorithms for translating a regular expression into a regular grammar are presented by Cohen in [12] and by Carrol and Long in [51].

Given a regular expression, we construct a regular grammar considering the three types of operations that we have for a regular expression. The algorithm parses through the entire regular expression, and for each of the three types of operations, introduces new equivalent production rules.

4.5.1 Base Algorithm

Algorithm 6: Constructing a regular grammar from a regular expression [51]

Input: regular expression $\mathcal{R}$.

Output: formal grammar $\mathcal{G}$.

Method:

1. Introduce a nonterminal S as a start nonterminal and a pseudoproduction $S \longrightarrow R$.

2. For a pseudoproduction $C \longrightarrow R$:

   If $R = a \bullet b$ introduce a nonterminal B and productions

   $C \longrightarrow aB$, $B \longrightarrow b$ instead of $C \longrightarrow R$;

   If $R = a/b$ introduce two new productions

   $C \longrightarrow a$ and $C \longrightarrow b$ instead of $C \longrightarrow R$;

   If $R = a^*$ introduce a new nonterminal B and productions

   $C \longrightarrow a \bullet B$, $B \longrightarrow a \bullet B$, $C \rightarrow \Lambda$, $C \longrightarrow a$, $B \longrightarrow a$ instead of $C \longrightarrow R$, and for all productions with C on the right hand side, i.e. $X \longrightarrow a \bullet C$, introduce a new production $X \longrightarrow A$.

3. Repeat 2 until no pseudoproductions are left.

4. Eliminate all unary and empty productions except the start empty production if it exists or appears.

5. The resulting grammar will be a right linear grammar.

4.5.2 Application

An example of how this algorithm can be used will now be given. Consider a language defined by the regular expression a.b.c.d*/a.b.d*. Using Algorithm 6 above, this expression

will now be translated into a regular grammar. The first step is to introduce a nonterminal S as a start nonterminal and a pseudo production S⟶R.

S⟶$a.b.c.d^{*}/a.b.d^{*}$

The next step is to identify which operation is being performed, and apply the appropriate rule on that operation. The best way to do this is to first identify all concatenation operations and to convert these into the appropriate form. Hence we first take the operation S⟶ $a.b.c.d^{*}/a.b.d^{*}$, and form a new productions B⟶$b.c.d^{*}$ and C⟶$b.d^{*}$, and replacing $(b.c)$ in the first operation with **B**. This gives us:

S⟶ a.B/a.C

B⟶$b.c.d^{*}$

c⟶$b.d^{*}$

We can also apply the same rule on the concatenation operation B⟶$b.c.d^{*}$ resulting in the productions:

S⟶ a.B/a.C

B⟶bD , C⟶bE, D⟶c.E, E⟶ d^{*}

We can now apply the closure rule on the operation E⟶ d^{*}, where we will introduce new productions: E⟶dF , E⟶d, F⟶dF , F⟶d, E⟶λ, F⟶ λ.

Hence,

S⟶ a.B/a.C

B⟶bD , C⟶bE, D⟶c.E

E⟶dF , E⟶d, F⟶dF , F⟶d, E⟶λ, F⟶ λ

All that remains is to now split the choice operations into two productions to form:

S→ a.B, S→a.C

B→bD , C→bE, D→c.E

E→dF , E→d, F→dF , F→d, E→λ, F→ λ

The resulting language defined using a regular grammar is equivalent to the language defined by a regular expression. All the properties of a regular grammar are met.

4.6 Translating from Regular Grammar into a Regular Expression

The previous section showed that the language defined by a regular expression can also be defined as a regular grammar. For the language to be regular, the opposite must also hold, i.e. that the language defined by a regular grammar can also be defined by a regular expression. A method for translating a regular grammar into a regular expression is now presented. Algorithms for translating a regular grammar into a regular expression are presented by Cohen in [12] and by Carrol and Long in [51].

4.6.1 Base Algorithm

The method for translating a regular grammar into a regular expression follows. The algorithm takes the regular grammar and tries to combine the rules into one rule, containing no nonterminals. Since nonterminals cannot be replaced directly by a terminal, the algorithm makes a few passes through the set of rules, making sure that when a nonterminal is eliminated, the language defined by the resulting rules is not affected in any way.

Algorithm 7: Constructing a regular expression from a regular grammar. [51]

Input: formal grammar G .

Output: regular expression $\mathcal{R}$.

Method:

1. For each nonterminal A combine all pseudoproductions of the form $A \longrightarrow R_1$, $A \longrightarrow R_2$ ... $A \longrightarrow R_k$ in one $A \longrightarrow R_1 / R_2 / \ldots / R_k$
2. Eliminate consecutively those nonterminals the right sides of which do not contain nonterminals by substituting their occurrences with the right side of the associated pseudoproduction.
3. For each nonterminal A regroup the right side of $A \longrightarrow R_1 / R_2 / \ldots / R_k$ in such a way that all components containing rightmost occurrence of A are situated leftmost. Transform

$$A \rightarrow Q_1A / Q_2A / \ldots / Q_l\ A / \mathcal{R}_1 / \mathcal{R}_2 / \ldots / \mathcal{R}_m$$

   into

$$A \rightarrow (Q_1 / Q_2 / \ldots / Q_l\ )^*\ (R_1 / R_2 / \ldots / R_m)$$

4. Eliminate consecutively the occurrences of nonterminals such that the right sides of the pseudoproductions associated with them do not contain nonterminals by substituting their occurrences with the right side of the associated pseudoproduction.
5. If the right side $\mathcal{R}$ of the pseudoproduction for the start nonterminal, S, does not contain nonterminals exit with result $\mathcal{R}$. Otherwise eliminate one of the nonterminals different from S by substituting its occurrences with the right side of the associated pseudoproduction and repeat the steps from 2.

4.6.2 Application

An example is now shown of how a regular grammar can be translated into a regular expression. We will consider the regular grammar that resulted from the translation in the previous section. The rules are as follows:

S→ a.B, S→a.C

B→bD , C→bE, D→c.E

E→dF , E→d, E→ λ

F→dF , F→d, F→ λ

The aim of the algorithm is to combine the productions into one production, consisting of only terminal symbols. We first combine pseudo productions to form:

S→ a.B, a.C

B→bD , C→bE, D→c.E

E→dF , d, λ

F→dF , d, λ

Now the product F→dF indicates that a closure operation occurs here. We therefore replace the production with the production F→d^*. This give us:

S→ a.B, a.C

B→bD , C→bE, D→c.E

E→dF , d, λ

F→d^* , d, λ

Now, where possible, we replace the nonterminal symbols with terminal symbols. Following the procedure above, we first replace all occurrences of the nonterminal **F**, since F produces only terminals. This gives us:

S$\longrightarrow$ a.B, a.C

B$\longrightarrow$$bD , C\longrightarrow$$bE, D\longrightarrow$$c$.E

E$\longrightarrow$$dd*$, dd , d, $d*$, λ

We notice that **E** now produces only terminals, so we now replace all occurrences of **E**. Hence the following productions are formed.

S$\longrightarrow$ a.B, a.C

B$\longrightarrow$$b$D

C$\longrightarrow$$b.d.d*$, $b.d.d$, $b.d$, $b.d*$, b

D$\longrightarrow$$c.$ $d.d*$, $c.d.d$, $c.d$, $c.d*$, c

This set can now be repeated with C and D, and then B. Hence we combine all these to form:

S$\longrightarrow$ $a.b.c.$ $d.d*$, $a.b.c.d.d$, $a.b.c.d$, $a.b.c.d*$, $a.b$, $a.b.c$

, $a.b.d.d*$, $a.b.d.d$, $a.b.d$, $a.b.d*$, $a.b$

We now combine the productions to give us:

S$\longrightarrow$ $a.b.c.$ $d.d*$/ $a.b.c.d.d$ /$a.b.c.d$/ $a.b.c.d*$/ $a.b$/ $a.b.c$ /

$a.b.d.d*$/ $a.b.d.d$ / $a.b.d$/ $a.b.d*$/ $a.b$

Hence the regular expression equivalent to the language defined by the regular grammar will be:

$$a.b.c.\ d.d^*/\ a.b.c.d.d\ /a.b.c.d/\ \ a.b.c.d^*/\ \ a.b/\ a.b.c\ /$$

$$a.b.d.d^*/\ a.b.d.d\ /\ a.b.d/\ a.b.d^*/\ \ a.b$$

Although this expression looks different to the expression we started out with in the previous section, the two expressions define the same language.

4.6.3 Possible Problem Areas

The algorithm presented above assumes that the regular grammar does not contain any redundant symbols. The regular grammar could have redundant symbols if there exist symbols that do not appear in any derivable string or though it appears in some derivable string it does not help in the derivation of any terminal string. If the regular grammar does contain symbols of this sort, translating the regular expression could lead to an expression that does not define an equivalent language to the language defined by the regular grammar. To eliminate this problem, it would be best to eliminate such redundancies by eliminating unreachable and nonterminating symbols from the rules in the regular grammar first, and then translating the regular grammar into its equivalent regular expression. Procedures for eliminating redundancies have been given in Section 2.2.3 above. This is however outside the scope of this thesis, and hence not implemented.

4.7 Translating from Regular Grammar into a Nondeterministic Finite Acceptor

The previous sections demonstrated that a language that is defined by a regular expression can also be defined by a finite automaton and by a regular grammar. What remains is to show that the language defined by a regular grammar can also be defined by a finite automaton, and vice verse. A method for translating a regular grammar into a nondeterministic finite acceptor is now given. Algorithms for translating a regular grammar into a nondeterministic finite acceptor are presented by Cohen in [12], Carrol and Long in [51], Davis and Weyuker in [23], Hopcroft and Ullman in [19], McNaughton in [24], Gries in [17] and Glenn Brookshear in [22].

The algorithm for constructing a nondeterministic finite acceptor from a regular grammar takes the regular grammar, and for each rule constructs a transition from one state to another, where the nonterminals are used to label the states, and the terminals are the inputs for the production. Since the regular grammar is not necessarily in its simplified form, it is assumed that the resulting machine is a nondeterministic finite acceptor, and hence using the methods presented in previous sections, the nondeterministic finite acceptor can be translated into a deterministic finite acceptor and then minimized.

4.7.1 Base Algorithm

Algorithm 8: Constructing a nondeterministic finite acceptor from a regular grammar. [24]

Input: formal grammar G

Output: nondeterministic finite acceptor $\mathcal{F}$

Method:

- Draw a state for every nonterminal. Make the state S_0 the start state. Then draw one extra state that is an accept state and label it S_1.
- For any production of the form X →wY where wY is a semiword, draw a transition from state X to state Y and label it with w. Note that X and Y may be the same state, which means you should draw a self-loop on that state labeled with w.
- For any production of the form X →w where w is either a word or λ, draw a transition from state X to the accept state S_1 and label it with w.
- The resulting machine is in the form of a transition graph.

4.7.2 Application

An example of how a regular grammar can be translated into a nondeterministic finite acceptor is now given. We will consider the regular grammar that resulted from the translation in the previous section. The grammar rules are as follows:

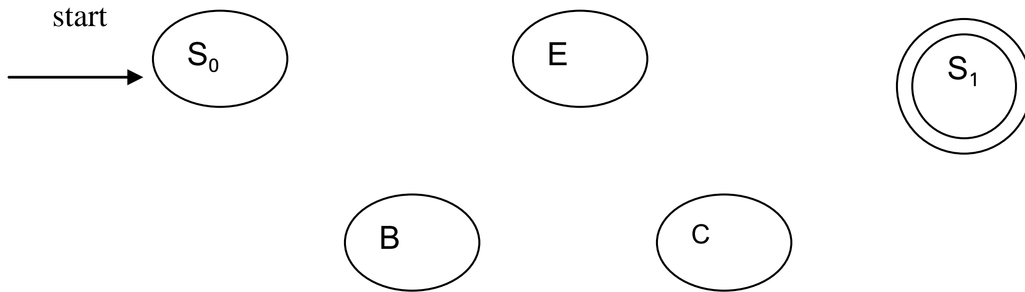
$S_0 \longrightarrow a\mathrm{E}$, $S_0 \longrightarrow b\mathrm{E}$, $S_0 \longrightarrow a\mathrm{B}$

$\mathrm{E} \longrightarrow a\mathrm{E}$, $\mathrm{E} \longrightarrow b\mathrm{E}$, $\mathrm{E} \longrightarrow a\mathrm{B}$

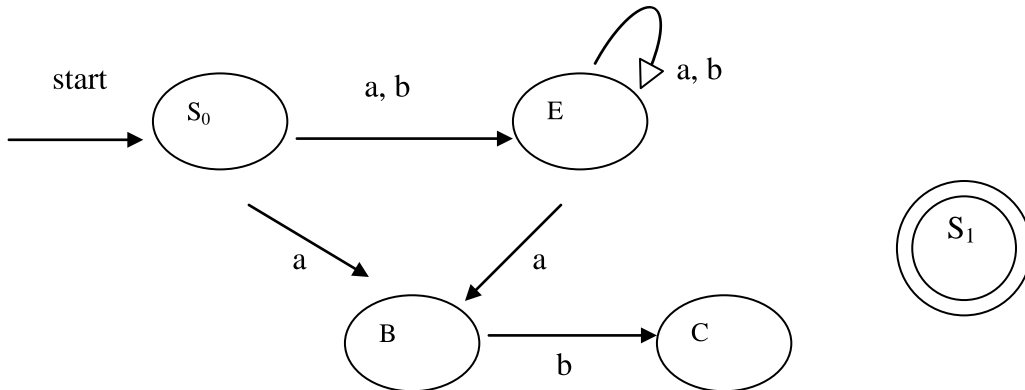
$\mathrm{B} \longrightarrow b\mathrm{C}$

$\mathrm{C} \longrightarrow b$

The first step is to draw a state for each nonterminal, making $\mathbf{S_0}$ the start state, and to draw an additional state, S_1, that is an end state. This gives us:



Now, for each production of the form X →wY, we draw a transition from state X to state Y and label w. This gives us:



For any production of the form X →w, we draw a transition from state X to the end state S_1 and label it with w. This will result in the following finite acceptor:

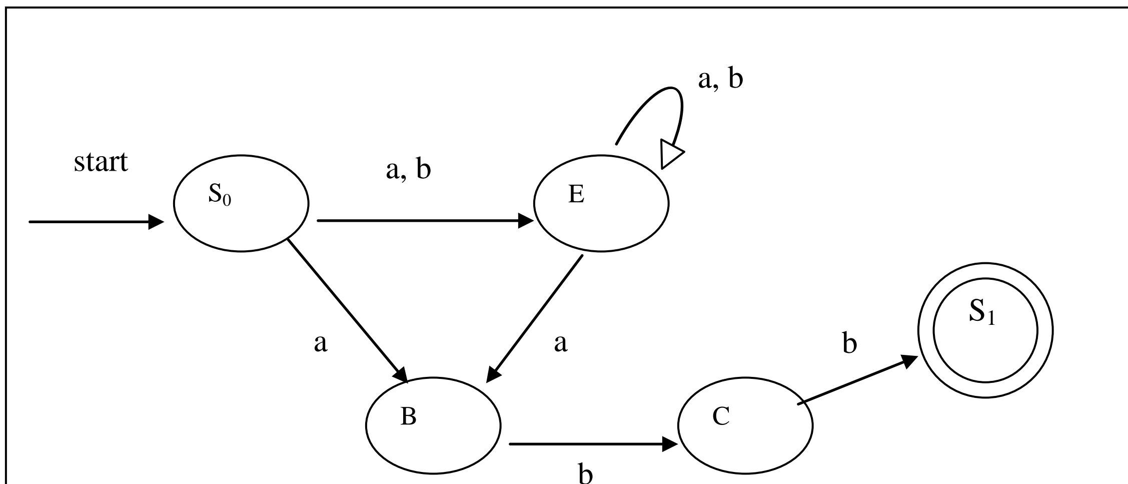


Figure 18: State diagram for the DFA representing the language (a.b)*.a.b.b

Since states $\mathbf{S_0}$ and **E** both have more than one production for the input a, the finite acceptor is a nondeterministic finite acceptor. Using steps in previous sections, this can easily be translated into a minimized deterministic finite acceptor. The language defined by the finite acceptor is equivalent to the language defined by the regular grammar shown above.

4.8 Translating from Deterministic Finite Acceptor into a Regular Grammar

The final step is to show that a language defined using a deterministic finite acceptor is equivalent to one defined using a regular grammar. A method for translating a deterministic finite acceptor into a regular grammar is now given. Algorithms for translating a nondeterministic finite acceptor into a regular grammar are presented by Cohen in [12], Carrol and Long in [51], Davis and Weyuker in [23], Hopcroft and Ullman in [19], McNaughton in [24], Gries in [17] and Glenn Brookshear in [22].

4.8.1 Base Algorithm

Let each state in the deterministic finite acceptor correspond to a non-terminal in the grammar. Each symbol used on the transitions corresponds to a terminal symbol.

Algorithm 9: Constructing a regular grammar from a deterministic finite acceptor [24]

Input: deterministic finite acceptor $\mathcal{D}$.

Output: formal grammar $\mathcal{G}$.

Method:

Let S_i denote the non-terminal corresponding to state i of the deterministic finite acceptor

1. The start symbol of the grammar is S_0, the non-terminal corresponding to the start state of the deterministic finite acceptor
2. For each transition from state i to state j on some symbol 'a', create a production rule of the form: $S_i \rightarrow aS_j$
3. For each state i of the deterministic finite acceptor which is a final state, create a production rule of the form: $S_i \rightarrow \lambda$

Note that λ-transitions between states would generate production rules of the form $S_i \rightarrow S_j$, which means that "wherever we see S_i we can replace it with S_j" – another indication that λ-transitions allow for the transition from one state to another without an element of the input string being read.

4.8.2 Application

An example of how a language defined by a deterministic finite acceptor can be translated into an equivalent language defined by a regular grammar is now given. Consider the deterministic finite acceptor, with {0} as the start state, shown in the figure below.

State	Input Symbol			
	a	b	c	d
{0}	{1}	-	-	-
{1}	-	{2}	-	-
{2}	-	-	{3}	{3}
{3}	-	-	-	{3}

Figure 19: State diagram for the DFA representing the language a.b.(c/d*)

To construct a regular grammar, for each transition from state i to state j on some symbol ‘a’, we need to create a production rule of the form: $S_i \rightarrow aS_j$

Hence we obtain the following set of rules.

$S_0 \rightarrow aS_1$

$S_1 \rightarrow bS_2$

$S_2 \rightarrow cS_3$

$S_2 \rightarrow dS_3$

$S_3 \rightarrow dS_3$

$S_3 \rightarrow \lambda$

4.9 Conclusions

The aim of this chapter was to present different techniques for translating from one model representation of a language to another model representation, defining either the same language, or an equivalent language. We have shown that it is possible to translate from any one of the three models into either of the other two models, and we are confident that the language defined by the resulting model is in fact equivalent to the initial language definition. We have also shown that, in the case of the finite acceptor, it is possible to translate from a nondeterministic finite acceptor to a deterministic finite acceptor, which aids the process of translating from a finite acceptor to any of the other two models. A λ-free algorithm for translating a regular expression into a nondeterministic finite acceptor was also presented.

Chapter 5

Implementation

In this chapter of the dissertation, we will consider the design and implementation of an application that deals with the three different representations of regular languages, namely, regular expressions, regular grammars and finite acceptors. Some of the algorithms that were presented in chapter 4 will be implemented in this software package. The package is made up two parts, one for converting a regular expression to a nondeterministic finite acceptor, named “Re2Nfa.exe”, and the other to convert an NFA to a DFA, a DFA to a grammar, and a grammar to an NFA, named “Fa2Rg.exe”. The package allows the user to define a language using a regular expression, a regular grammar or a finite acceptor. The main purpose of the first part is to implement Algorithm 2 (pg. 47) in Chapter 4. The second part is an implementation of some of the other algorithms used to translate from one representation to another. This is similar to some of the implementations that have been given in Chapter 3. The design of the package is in two parts, the first being the interface for entering the three different representations of the language, and the second being the implementation of the algorithms to translate from one representation to another.

5.1 Programming Languages

The program is designed using Borland Delphi. There are several reasons for this choice. First of all Delphi is a modern rapid application development tool and is very comfortable to work with. Delphi is based on the Object Pascal programming language, which although not as powerful as C++, is much more readable, easy to use, and protects by design against

programmer mistakes. According to the experts Teixeira & Pacheco [52], Calvert [53], Cantu [54] for most programming tasks Delphi needs smaller source code and produces smaller and faster applications. Additionally, according to Teixeira & Pacheco [52], the Pascal compiler upon which Delphi is based is known as the fastest compiler for the Windows operating system. All these features make Delphi appropriate for the tasks of this project.

5.2 Model Designs

The entire program consists of three forms, one for each representation method available. Users will select their desired representation and enter the description of the language they are using. A brief explanation for each of the forms is now given.

5.2.1 The Regular Expression Editor

It is desired that the regular expression accepts a string of any length. This is because a regular expression, although commonly represented as a short string of characters, can be of any length in an unsimplified form. The system must also be able to recognise the three basic operations on a regular expression, i.e. concatenation, ".", choice among alternatives, "/", and repetition or closure, "*". This was achieved using the TRichEdit component in Delphi. The user can input a regular expression with all its components, using symbols from the keyboard. There is, however, a restriction in that the length of each input symbol can only be a single character. This is done for programming purposes only, and can be increased to a string of any length if desired. A sample for the regular expression editor is shown below.

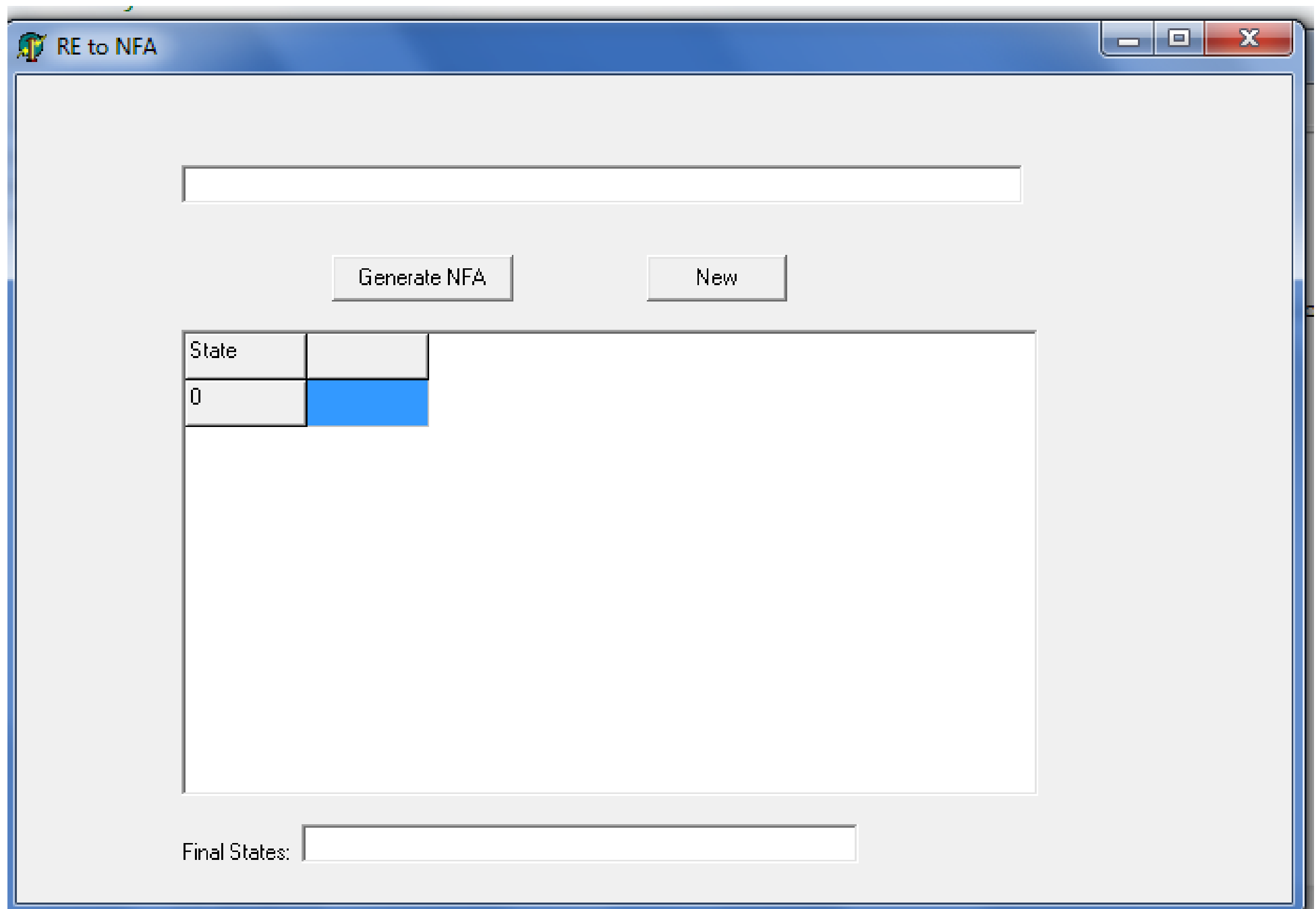


Figure 20: Design of the regular expression editor

Once the user has entered their regular expression, they will have the option to translate it into a nondeterministic finite acceptor by clicking on the appropriate button on the screen.

5.2.2 The Finite Acceptor Editor

The finite acceptor will be represented using transition tables. This is one method for representing a finite state acceptor. The user will be able to enter the states for the acceptor, the input symbols that the language will use, as well as the transition functions from one state

to another for a given input. The finite acceptor editor is therefore implemented using a TStringGrid. A sample for the finite acceptor editor is shown below.

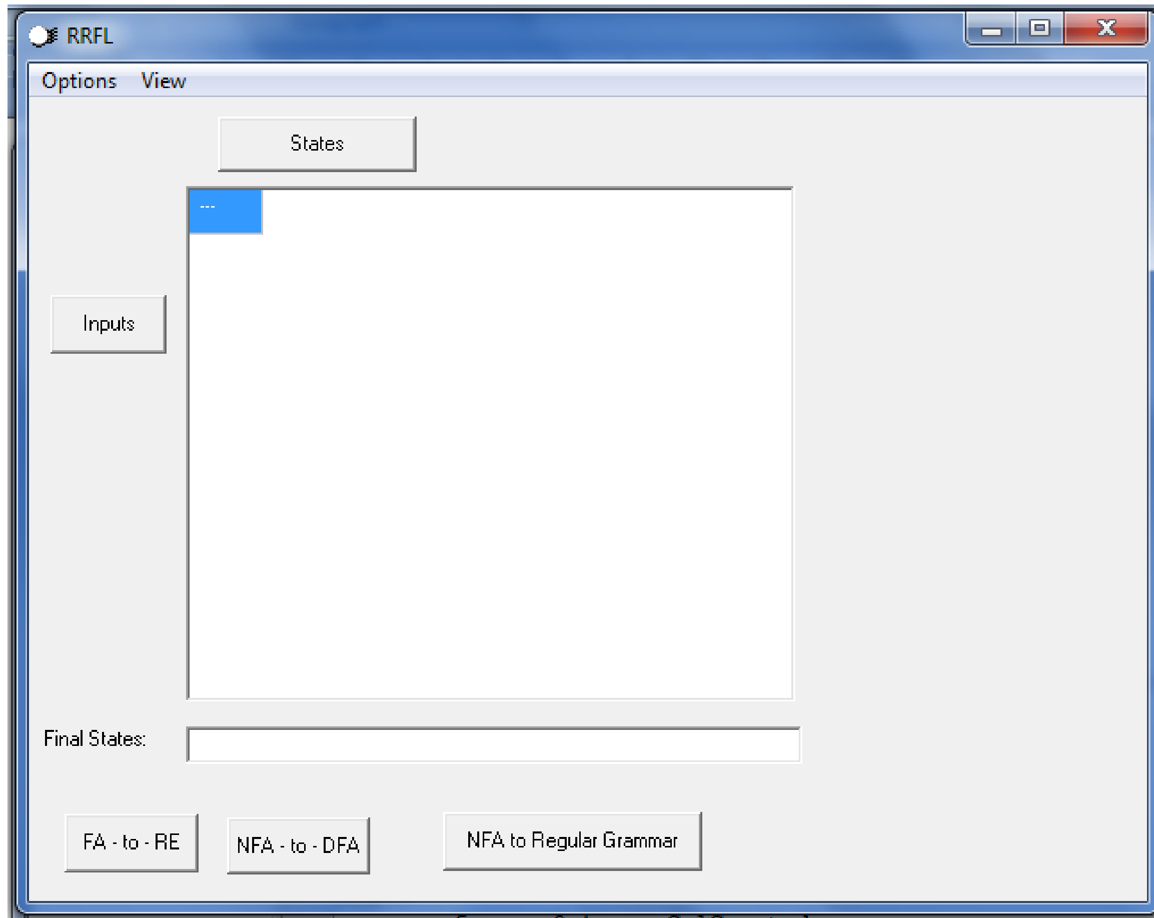


Figure 21: Design of the finite acceptor editor

To insert an input symbol or a state, the user clicks on the appropriate button, and a new state/ input symbol cell is presented. The user will then insert the name for that particular cell. It should be noted that the user will not be able to insert a new state or input symbol if the previous one has not been named as yet. The system will also check that no duplicate state names or input symbols are created. Should the user wish to use the λ symbol, the user must insert a "_". To insert a transition from one state to another, the user will select the intersecting cell of the state onto which the transition function will be applied, and the input symbol that will be used for that transition and will insert the state to which the transition will lead into

that cell. If there is more than one transition from a given state on a given input, the user can double click on the intersecting cell, and a memo box will appear. The user will then type the different states that the different transitions lead to, one state per line, and click on the "OK" button to return to the editor. The "||" symbol separates the different states that can be reached from a particular state. Although the user can choose whether to input a nondeterministic finite acceptor or a deterministic finite acceptor, it is assumed that a nondeterministic finite acceptor had been entered. Final states of the nondeterministic finite acceptor should be entered in the space provided, each state separated by a ",".Once the user has entered the definition of the language represented, the user will have the option of either translating the representation into a regular grammar, or to translate the nondeterministic finite acceptor into a deterministic finite acceptor, by clicking on one of the buttons at the bottom of the screen, or by using the options in the menu bar at the top of the screen. Since the algorithms presented in Chapter 4 for translating a finite acceptor into one of the other representations was for a deterministic finite acceptor, if the user chooses to translate the nondeterministic finite acceptor into one of the other representations, the system will first translate the nondeterministic finite acceptor into a deterministic finite acceptor and then into the other representation of the language. This process is unseen to the user.

5.2.3 The Regular Grammar Editor

The regular grammar will be represented as a grid, where the first cell of each row represents the nonterminal symbol, and each cell of a row represents a production rule from that nonterminal. The regular grammar editor is therefore implemented using a TStringGrid. A sample of the regular grammar editor is shown below.

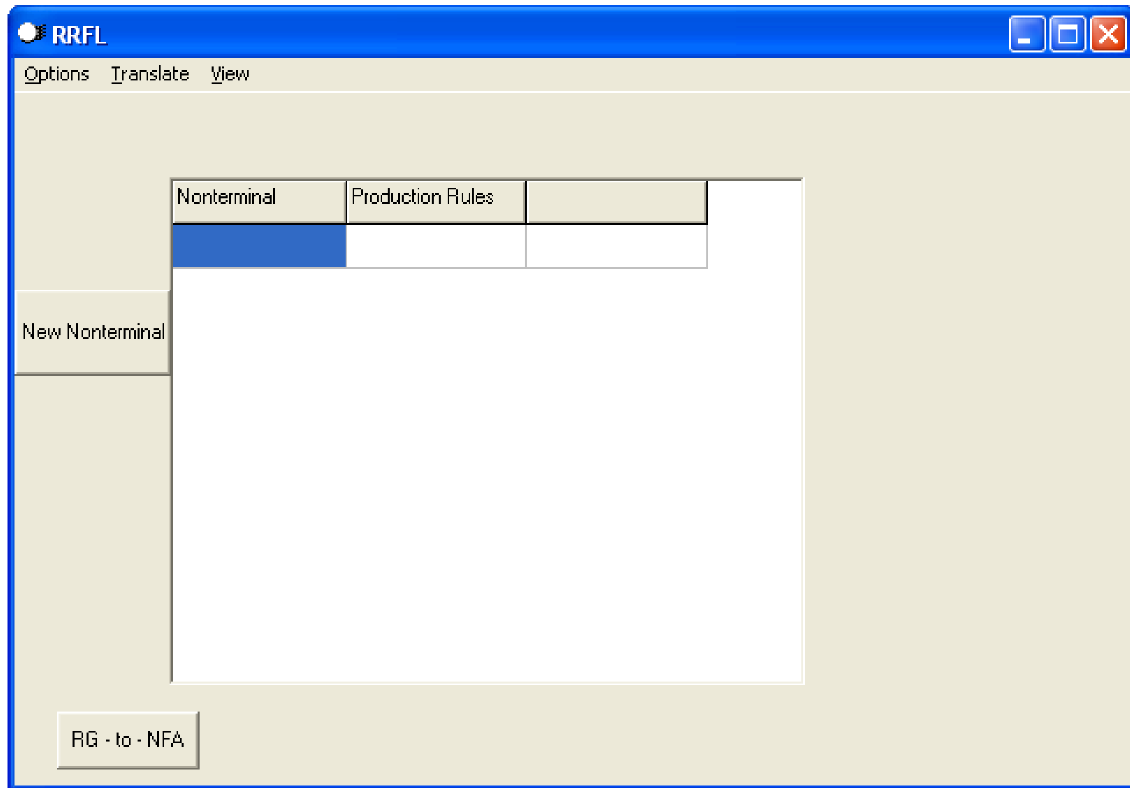


Figure 22: Design of the regular grammar editor

For continuity and ease of use, the nonterminals will be numbered, where state 0 represents S_0, the start symbol. The set of terminals may be any string of characters, excluding numeric characters. The user will be able to insert the production rules for the nonterminals, and each time a new nonterminal is entered in a production rule, a new row is created in the grid for that nonterminal. The user will start with two blank columns in which the user will be able to insert production rules for a given nonterminal. Should the user require more columns, once the user has filled the columns that are blank, the user will be able to insert a new column by pressing the right arrow key on the keyboard. A new column will automatically be inserted. It must be noted that all the cells in a row need to be filled in before a new column can be inserted. As the user enters the production rule, the set of terminals and nonterminals, which is

located at the top of the screen, will automatically be updated. Once the user has entered the regular grammar, the user will have the option of translating the representation into either a regular expression or a nondeterministic finite acceptor , either by clicking on one of the buttons at the bottom of the screen, or by using the options in the menu bar at the top of the screen.

5.3 Translation techniques

5.3.1 Translating from Regular Expression to Nondeterministic Finite Acceptor

Once the user has entered the regular expression to be tested and pressed the "Finite Acceptor" button, the program translates the regular expression into a nondeterministic finite acceptor, using Algorithm 2 (pg 42). There are five different cases that the program needs to check for and apply the appropriate rule for that case. The five cases are:

- a symbol,
- the concatenation operation,
- the choice operation,
- the closure operation,
- a bracket.

When a symbol is encountered, the program first reads in the entire string, if it is more than one character in length. The program then checks whether the symbol has appeared before. If the symbol has not appeared before, then the program will create a new state in the

transition table for the nondeterministic finite acceptor. A new input is also created that's labeled with the symbol name.

When a concatenation operation is encountered, the program creates a new input labeled with the next symbol's name, if the input did not appear before, and a new state. It then inserts a transition from the previous state to the new state on input "symbol".

When a union operation is encountered, the program reads the two nondeterministic finite acceptors on either side of the union operator, using the above two procedures. The nondeterministic finite acceptor is constructed by creating a transition from the current state to the second state of each of the two nondeterministic finite acceptors. If the nondeterministic finite acceptor only has one state, the transition is created from the current state to the first state of the nondeterministic finite acceptor. The input symbol that is inserted for this transition is the same as the input symbol for the transition from the first state of the nondeterministic finite acceptor to the second sate of the nondeterministic finite acceptor. The set of accepting states of the two nondeterministic finite acceptors become the set of accepting states for the combined nondeterministic finite acceptor.

When the closure operation is encountered, the program using the procedures above reads in the nondeterministic finite acceptor for the operation that the closure is to be performed on. Once this has been done, a transition is created from the last state in the nondeterministic finite acceptor to the second state in the nondeterministic finite acceptor. The input symbol for this transition is the same as the input symbol for the transition from the first state in the nondeterministic finite acceptor to the second state in the nondeterministic finite

acceptor. The first state of the nondeterministic finite acceptor also becomes the accepting state of the nondeterministic finite acceptor.

When a bracket operation is encountered, the nondeterministic finite acceptor for everything within the brackets is created using the operations above. In this way the entire regular expression is translated into its corresponding nondeterministic finite acceptor.

5.3.2 Translating from Nondeterministic Finite Acceptor to Deterministic Finite Acceptor

The program takes the nondeterministic finite acceptor, with the start state, the accepting states and the transitions for each of the states defined. To construct the deterministic finite acceptor, the program takes the start state for the nondeterministic finite acceptor and creates a new state with the same label in another grid. For each input symbol defined in the nondeterministic finite acceptor, the program defines new states in the new grid with the same input names. From the start state with each input, the program computes all the possible transitions on that input. The combination of all the possible transitions on a particular input becomes a single new state set. For all the new state sets that have been defined, all the possible transitions on all the inputs are computed and the new state sets are defined as before. The transitions for these state sets are then computed as before and new state sets are defined. Note that if there is a repetition of a state set, a new state set does not have to be created. The one that has already been created is used. This process is repeated until no new states are defined, and all transitions for all the states that are defined have been computed. Once this is complete, the deterministic finite acceptor is fully defined.

5.3.3 Minimizing the Deterministic Finite Acceptor

It is desirable that the deterministic finite acceptor have the least number of states possible but still define the given language. To achieve this, the program will minimize the deterministic finite acceptor obtained above. The program, using a memo box, partitions the deterministic finite acceptor into two subgroups. One group contains the set of accepting states, and the other group consists of all the nonaccepting states. The program takes the subgroup containing the set of accepting states, and on each input checks if the state makes a transition either to itself, or to a state that is in that subgroup. If a particular state in the subgroup makes a transition to a state that is not in that subgroup, then that state is removed from that subgroup and placed in a new subgroup. The subgroup containing the nonaccepting states is then examined and if necessary, new subgroups are created. This process is repeated until no new subgroups are created. Once this has been completed, a single state from each subgroup, usually the first state in the subgroup, is employed as the representative state of that group. All transition to states that are not representative states are changed such that the transition is made to the state that is the representative of that subgroup. These states are then removed from the deterministic finite acceptor since they are not accessible from the start state. The resultant deterministic finite acceptor will be the minimal deterministic finite acceptor.

5.3.4 Translating from Regular Grammar to Deterministic Finite Acceptor

The program will first creates n states for each of the n nonterminals in the regular grammar. The program also creates a new state, labelled n+1, to represent the final state. Then for each production of the form $i \rightarrow$ xj, where i and j represent nonterminals, and x represents a

terminal, the program creates a transition from state *i* to state *j* on input x. For each production of the form *i*→ x, where *i* represents a nonterminal, and x represents the terminal, the program creates a transition from state *i* to the final state *n*+1 on input x.

5.3.5 Translating from Deterministic Finite Acceptor to Regular Grammar

First, the program creates a start nonterminal {0} of the grammar, which corresponds to the start state of the deterministic finite acceptor, state {0}. Then for each transition from one state, say *i*, to another state, say *j*, in the deterministic finite acceptor on some input "a", the program first checks to see if the nonterminal {*i*} corresponding to the state *i* exists, and if it does not, it creates a new nonterminal {*i*}. The program then creates a new production "a*j*" on the nonterminal {*i*}. Hence the transition from say state 2 to state 3 on input x will result in the creating of the production 2→ x3. In this way the deterministic finite acceptor will be translated into a regular grammar.

5.3.6 Translating from Regular Expression to a Regular Grammar

Once the user has inputted the regular expression to be tested and presses the "Regular Grammar" button, the program translates the regular expression into a regular grammar, using Algorithm 6 above. There are four different cases that the program needs to check for and applies the appropriate rule for that case. The five cases are:

- the concatenation operation,
- the choice operation,
- the closure operation,
- a bracket.

When the initial symbol is read, the program created a new rule in the regular grammar, from a nonterminal labeled A, with the pseudoproduction A produces the symbol that we encountered.

When a concatenation operation is read, the program adds a new nonterminal, labeled as the next character in the alphabet, to the end of the current pseudoproduction, and creates a new pseudoproduction, which has the left side labeled with the new nonterminal that was just added, and which produces the symbol on the right side of the concatenation operation.

When a choice operation is read, the program reads the symbol on the right of the expression, and adds a new pseudoproduction with the current nonterminal on the left side of the production, and the symbol that is on the right of the expression on the right of the pseudoproduction.

When the closure operation is read, if the last rule created by the steps above is $C \rightarrow a$, the program will add a new nonterminal, labeled as the next character in the alphabet, and adds the following rules: $C \longrightarrow a \bullet D$, $D \longrightarrow a \bullet D$, $C \rightarrow \Lambda$, $C \longrightarrow a$, $D \longrightarrow a$.

When a bracket operation is read, the regular grammar for everything within the brackets is created using the operations above. The start point of the bracket is note, so any operation that follows the bracket will be added after this point in the pseudoproductions. In this way the entire regular expression is translated into its corresponding regular grammar.

5.3.7 Translating from Regular Grammar to a Regular Expression

Once the user has input the regular grammar to be converted and presses the "RG to RE" button, the program translates the regular grammar into a regular expression, using Algorithm

7 above. The program starts by finding all production with the same nonterminal on the left side of the production and on the right of the production on the right side, e.g. $F \longrightarrow dF$. The nonterminal on the extreme right is replace with a * to indicate Kleene closure. Now, where only terminals occur on the right side of a production, all instances of the nonterminal that were on the left side of that production will be replaced on the right side where those productions occur. This is done repeatedly until no nonternimals occur on the right side of and production rules. The production Swill now produce the rules that make up the regular expression, and these are joined together using union operations.

5.3.8 Translating from Deterministic Finite Acceptor to Regular Expression

From the deterministic finite acceptor the output includes a memo box that contains the set of accepting states. The algorithm for translating a deterministic finite acceptor into a regular expression requires that m, the number of accepting states, copies of the deterministic finite acceptor be made. In terms of computational efficiency it seems that this will waste memory space. Instead of making multiple copies of the deterministic finite acceptor, the deterministic finite acceptor was considered with one state from the set of accepting states as the only accepting state. The deterministic finite acceptor with this state as accepting state was translated, and the process was repeated with another accepting state from the set of accepting states being used as the accepting state. This process was repeated for all the accepting states in the set of accepting states. Note that each state was visited only once. The process for translating the deterministic finite acceptor with only one accepting state into a regular expression was computed as follows. A pointer is placed at the first state, all the transitions from that state were checked. If there was a transition from that state on a certain input to the

next state, the input symbol is added to the regular expression. The pointer is then moved to the next state. If there is more than one transition from the state pointed at, then if the other transition is to itself on a certain input, say a, then the input symbol with repetition, i.e. a*, is added to the regular expression. If the transition is to another state other than itself, then an open bracket is added to the regular expression followed by the input symbol for the transition from that state to one of the possible states, and a second pointer is placed at that state. The first pointer is moved to the first state of the state chosen from the possible states. Using the first pointer, the rest of the deterministic finite acceptor is traversed as mentioned before until a transition is made to the accepting state. A choice operator, i.e. a"/" is then added to the regular expression and the first pointer in moved back to where the second pointer is pointing to. The input symbol for the transition to another state of the possible states is then added to the regular expression and the path from that state to the accepting state is added to the regular expression. This process is repeated until all the possible transitions have been considered. Once this has been completed, the regular expression corresponding to the deterministic finite acceptor with the given accepting state is computed. A choice operator, i.e. a "/", is then added to the regular expression and the process is repeated for another accepting state. This is repeated until all the accepting states are considered. The result is the regular expression that corresponds to the given deterministic finite acceptor, i.e. the desired output of the entire computational process.

5.4 Test case examples

An example on how the software works will now be presented. For further instruction on how to use the software, refer to the user manual found in Appendix A of this dissertation.

Consider the expression (a.b.c.d*/a.b.d*). First type this expression into the regular expression editor, using the program RE2NFA.exe.

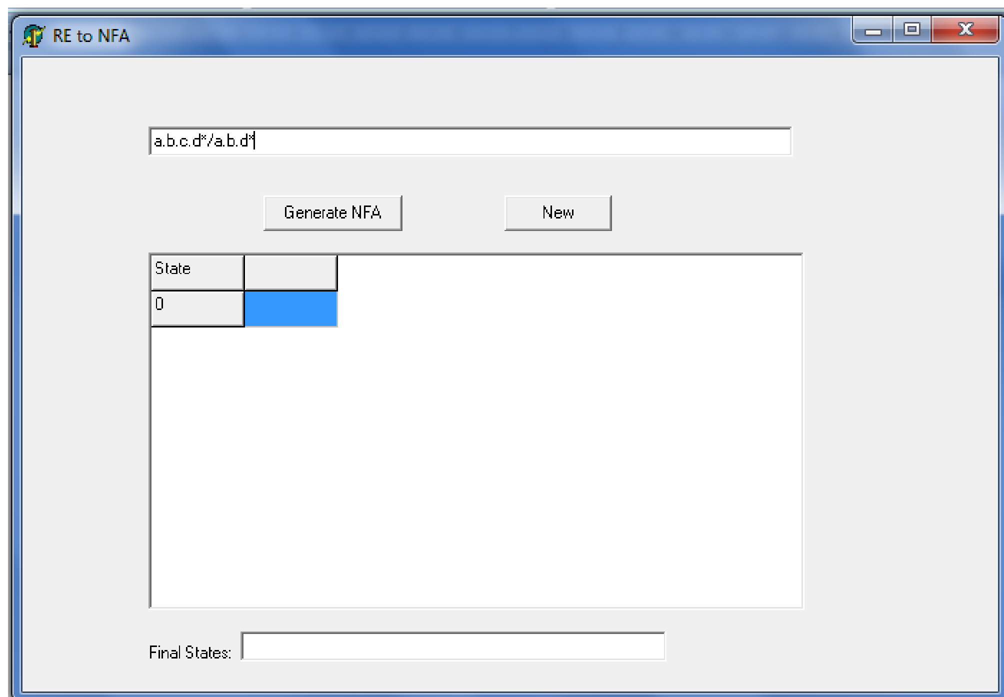


Figure 23: Regular expression editor for (a.b.c.d*/a.b.d*)

The user now has the option to translate the expression into a NFA. If the user selects to translate the expression into a NFA, the following will result:

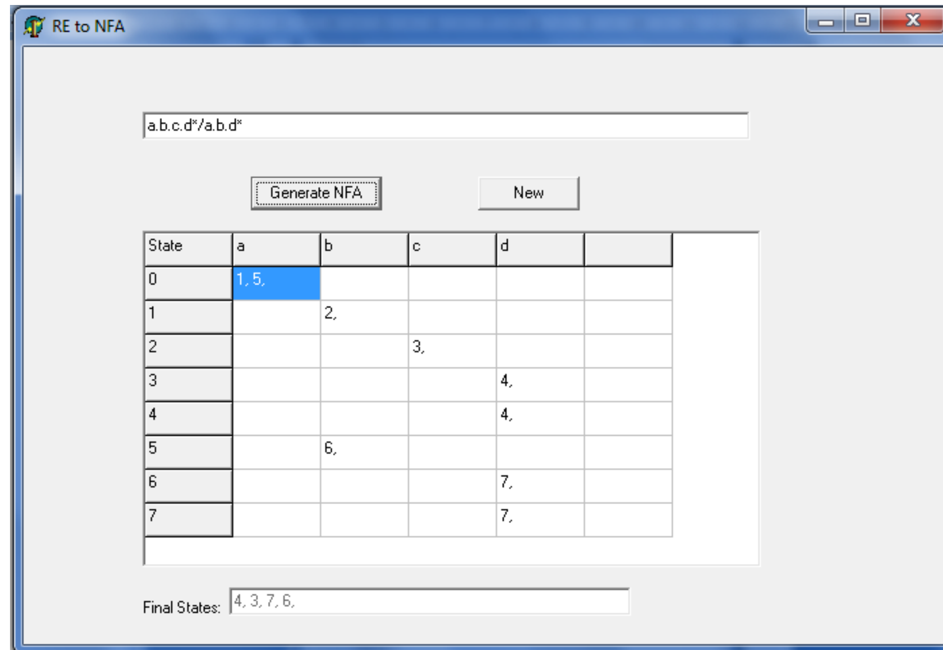


Figure 24: Finite acceptor editor for (a.b.c.d*/a.b.d*)

This NFA can now be entered into the second package, using the program Fa2Re.exe. The following will result:

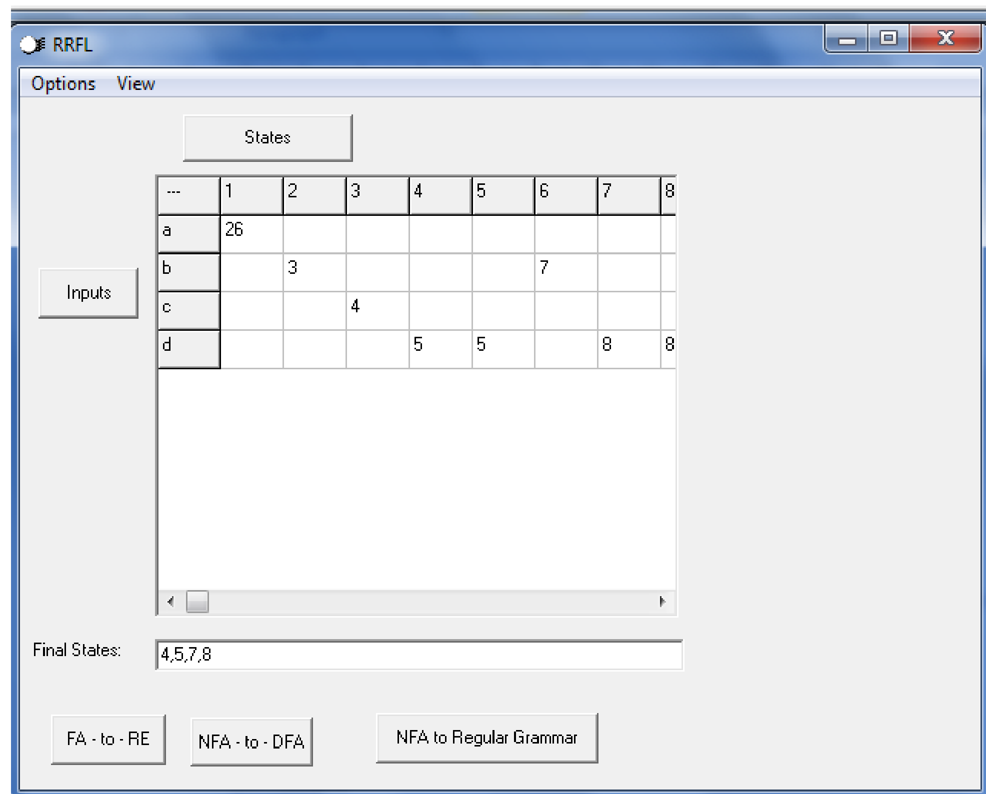


Figure 25: NFA editor for (a.b.c.d*/a.b.d*)

It should be noted that the format of the NFA in the first package and that of the second package is slightly different. This does not affect the NFA in any way. Whereas in the first package the rows contain the states and the columns contain the labels, in the second package this is vice-verse. Also, the states in the second package are numbered from 1, whereas in the first the states are numbered from 0. This was purely a design choice, and could be standardized as a future improvement. The user could now select to convert the NFA into either a DFA or a regular grammar. If the user now selects to convert this NFA into a regular grammar, the following will result:

RRFL

Options View

New Nonterminal

1	a2		
2	b3		
3	c4	d5	_
4	d6		_
5	d5		_
6	d6		_

RG - to - NFA

Figure 26: Regular Grammar for (a.b.c.d*/a.b.d*) translated from a NFA

It should be noted that the NFA was first translated into a DFA and then into the regular grammar. Also, the "_" indicates a final state from the DFA. This grammar can now be translated into a NFA, which gives us the following:

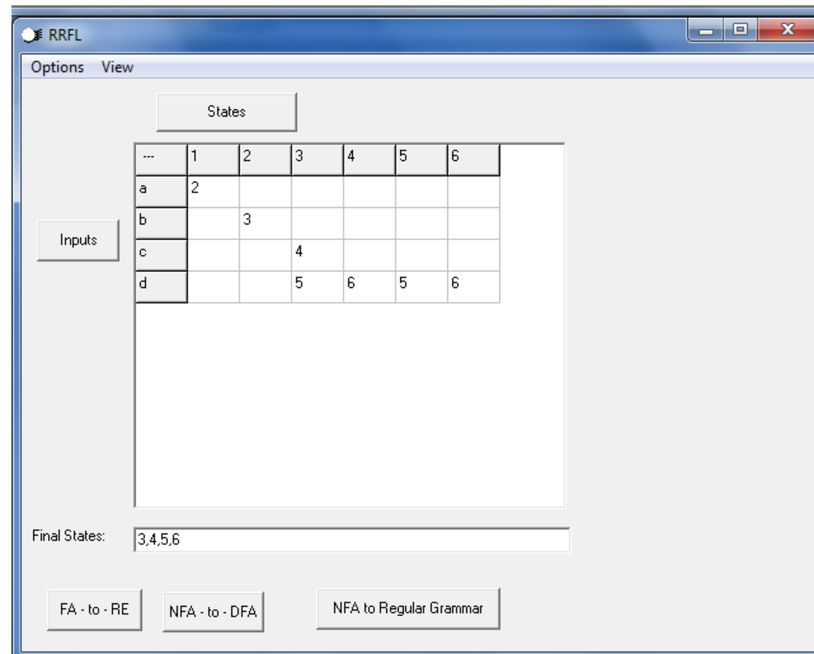


Figure 27: NFA for (a.b.c.d*/a.b.d*) translated from a regular grammar

If the user had selected to convert the initial NFA, see Figure 25, into a DFA, which gives us the following:

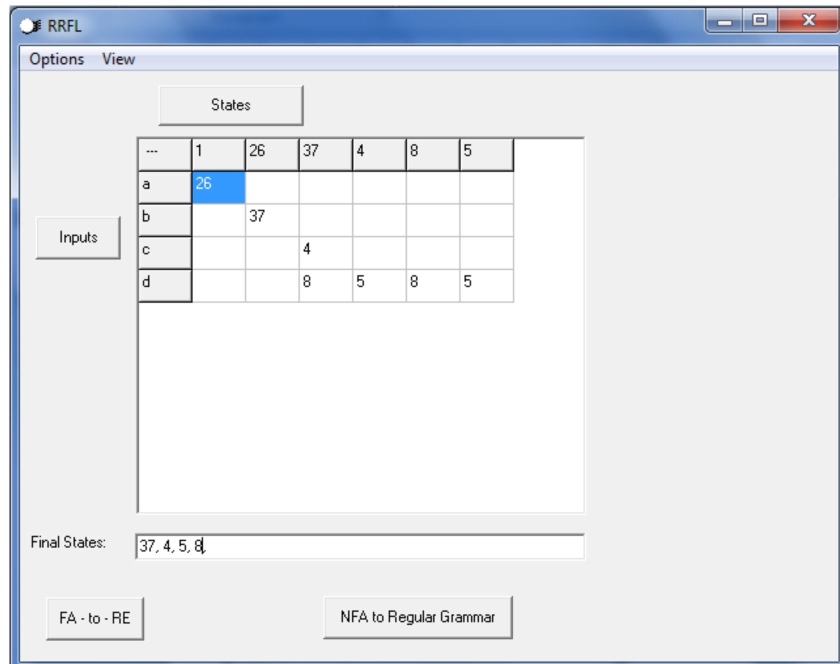


Figure 28: DFA for (a.b.c.d*/a.b.d*)

Notice the similarity between the NFA in Figure 27 and the DFA in Figure 28. This indicated that the NFA obtained from translating the regular grammar into an NFA and the DFA representing the same language are equivalent.

The DFA in Figure 28 could be converted into a regular grammar. If the DFA is converted into a regular grammar, the following will result:

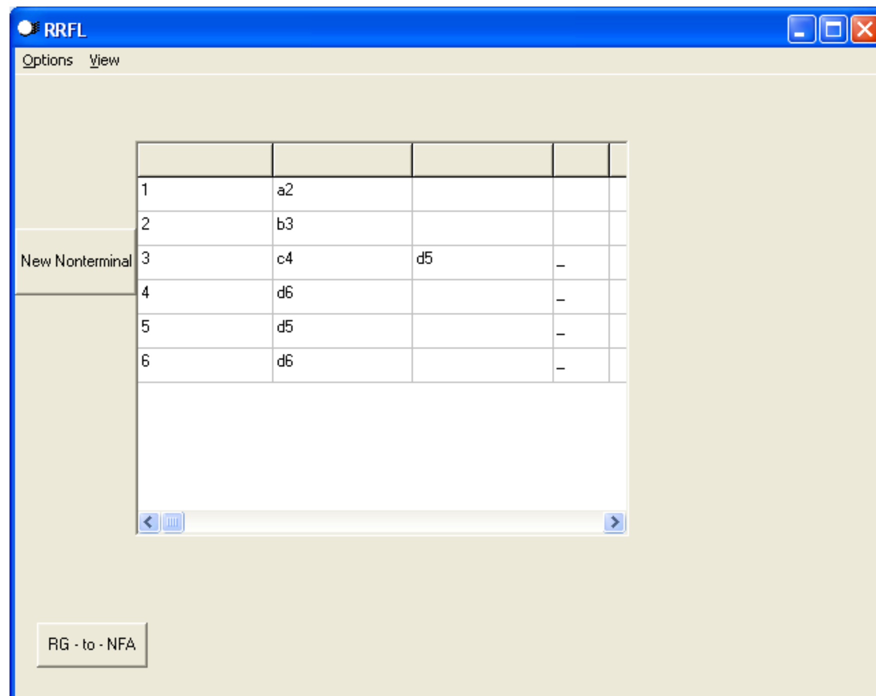


Figure 29: Regular grammar translation of DFA for (a.b.c.d*/a.b.d*)

If this grammar is converted back into a NFA, the resultant NFA will be the NFA in Figure 27.

5.5 Content of accompanied CD

At the back of this dissertation one will find a CD attached. The CD contains the following:

- A folder with the executable for the software.
- A folder with the source code for the software.

- A folder with the user documentation for the software.
- A folder with a .pdf copy of this dissertation.

Chapter 6

Conclusion

6.1 Summary

- Three approaches to represent a language were presented. The first approach uses a notational device for defining the symbols and the operations, which can be done using a *regular expression*. The second approach uses a mechanism for recognizing strings in a language. The device modeled in this dissertation was the *finite automaton*. The third approach is to specify the production rules for a "formal" language. Here the *regular grammar* is the most appropriate device and was implemented.

- Techniques for translation from any one of the three main models to any other have been presented, as well as technique to translate from one representation of the finite state acceptor to another. In a few cases, references to other methods for translation have been given.

- In the case of translating a regular expression into a nondeterministic finite acceptor, an algorithm has been proposed that constructs a λ-free nondeterministic finite acceptor.

- A single application was constructed to allow for the practical investigation of the three representations of a regular language, as well as for translating from one model to another. This tool should be a useful addition to the theoretical Computer Science lecturer's arsenal.

6.2 Future Improvements

It has been noticed that when a language defined using one model is translated to another model definition of the language, and the resultant model translated back into the original model representation, the initial definition and the resultant definition, although equivalent, seem to be different. One solution to this problem is to provide a means to “simplify” the definition of the language for any given model. This is most apparent in the case of the regular expression and the regular grammar. Methods for removing redundancy, such as removing unreachable and nonterminal symbols from the regular grammar, and simplifying the regular expression, could be incorporated into the software. Another possible improvement could be to incorporate a graphical representation of finite acceptors.

References

[1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* (Addison-Wesley, Reading, MA, 1974).

[2] Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. (Addison-Wesley, Reading, MA, 1988).

[3] Aho, A.V. and J.D. Ullman. *The Theory of parsing, Translating, and Compiling*, Vol. 1, (Prentice-Hall, Englewood Cliffs, 1972).

[4] Aho, A.V. and J.D. Ullman. *Foundations of Computer Science*. (Computer Science Press, New York, 1992).

[5] Bilska, A. A collection of tools for making automata theory and formal languages come alive. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) 29* (1997), 15–19.

[6] Berry, G. and Sethi, R., "From Regular Expressions to Deterministic Automata", *Theoretical Computer Science*, 48 (1986) 117 – 126.

[7] Brauer W., "On Minimizing Finite Automata", *EATCS Bulletin*, 35 (1988) 113–116.

[8] Bruggemann-Klein, A. “Regular expressions into finite automata”, *Theoretical Computer Science* 120 (1993) 197- 213.

[9] Brzozowski, J.A., “Derivatives of regular expressions”, J. *ACM* 11(4) (1964) 481-494.

[10] Champarnaud, J.M., “*From a regular expression to an automaton*”, Technical Report, IBP, LITP, Universite’ Paris 7, Paris, France, Working document 23 September 1993.

[11] Chang, C.-H., “From regular expressions to DFAs using compressed NFAs”, *Proceedings of the Third Symposium on Combinatorial Pattern Matching*, (1992) 90-110.

[12] Cohen D.I.A., *“Introduction to Computer Theory”*, (Wiley, New York, 1986).

[13] Chang, C.-H. and R. Paige., “*From regular expressions to DFAs using compressed NFAs*”, Technical Report, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, NY, 1992.

[14] DeRemer, F.L. Lexical analysis, in: F.L. Bauer and J. Eickel, eds., *Compiler Construction: an Advanced Course*, Lecture Notes in Computer Science 21 (Springer Verlag, Berlin, 1974) 109- 120.

[15] Glushkov, V.M., “The abstract theory of automata”, *Russian Mathematical Surveys* 16 (1961) 1-53.

[16] Goldberg, R.R., “Finite state automata from regular expression trees”, *The Computer Journal* 36(7) (1993) 623-630.

[17] Gries, D., *Compiler Construction for Digital Computers*, (John Wiley $ Sons, New York, 1971).

[18] Hopcroft, J.E., “An n log n algorithm for minimizing the states in a finite automaton”, in: Z. Kohavi, ed., *The Theory of Machines and Computations*. (Academic Press, New York, 1971) 189-196.

[19] Hopcroft, J.E. and J.D. Ullman., *Introduction to Automata, Theory, Languages, and Computation*. (Addison-Wesley, Reading, MA, 1979).

[20] Huffman, D.A., “The synthesis of sequential switching circuits”, *J. Franklin Institute* 257(3) (1954) 161-191 and 257(4) (1954) 275- 303.

[21] Kelley D., *“Automata and Formal Languages, An Introduction”*, (Prentice_Hall, Englewood Cliffs, 1995).

[22] Glenn Brookshear J., *Theory of Computation, Formal Languages, Automata, and Complexity,* (Benjamin/Cummings Publishing Company, California, 1989).

[23] Martin D. Davis and Elaine J. Weyuker, *Computability, Complexity, and Languages, Fundamentals of TheoreticalComputer Science*, (Academic Press, New York, 1983).

[24] Robert McNaughton, *Elementary Computability, Formal Languages, and Automata*, (Prentice-Hall, Englewood Cliffs, 1982).

[25] McNaughton, R. and H. Yamada., "Regular expressions and state graphs for automata", *IEEE Trans. on Electronic Computers* 9(1) (1960), 39- 47.

[26] Myhill, J., "Finite automata and the representation of events", *WADD TR-57-624* (1957), 112-137.

[27] Nerode, "A., Linear automaton transformations", *Proc. AMS* 9 (1958), 541- 544.

[28] Parker, S.P., ed., *Dictionary of scientific and technical terms*. (McGraw-Hill, New York, 4th edition, 1989).

[29] Procopiuc, M., Procopiuc, O., and Rodger, S. H. Visualization and interaction in the computer science formal languages course with jflap. *ASEE/IEEE Frontiers in Education (FIE) Conference* (1996), 121–125.

[30] Rabin, M.O. and D. Scott., “Finite automata and their decision problems”, *IBM J. Research* 3(2) (1959) 115- 125.

[31] Raymond, D.R. and D. Wood., “*The Grail papers: Version 2.0*”, Department of Computer Science, University of Waterloo, Canada, January 1994. Available by ftp from CS-archive.uwaterloo.ca.

[32] Robinson, M., Hamshar, J., Novillo, J., and Duchowski, A. A java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In *30th Annual ACM SIGCSE Symposium (Special Interest Group on Computer Science Education)* (New Orleans, Louisiana, March 1999).

[33] Rodger, S. H. Integrating hands-on work into the formal languages course via tools and programming. *Workshop on Implementing Automata, Lecture Notes In Computer Science 1260* (1996), 132–148.

[34] Rodger, S. H., and Gramond, E. Using JFLAP to interact with theorems in automata theory. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) 31* (1999), 336–340.

[35] Rodger, S. H., and Hung, T. Increasing visualization and interaction in the automata theory course. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) 32* (2000), 6–10.

[36] Deus ex machine, http://www.ics.uci.edu/ savoiu/dem/. Web site, March 2013.

[37] Thompson, K., "Regular expression search algorithms", *Comm. ACM* 11(6) (1968) 419- 422.

[38] van de Snepscheut, J.L.A., "*Trace theory and VLSI design*", Ph.D dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1985. Also available as Lecture Notes in Computer Science 200 (Springer-Verlag, Berlin, 1985).

[39] Luiz Filipe M. Vieira, Marcos Augusto M. Vieira, Newton J. Vieira, "Language Emulator, a helpful toolkit in the learning process of Computer Theory", *ACM SIGCSE Bulletin* 36(1) (2004), 135-139

[40] Watson, B.W., "*A taxonomy of finite automata minimization algorithms*", Computing Science Report 93/44, Eindhoven University of Technology, The Netherlands, 1993. Available for ftp from ftp.win.tue.nl in directory /pub/techreports/pi/automata/.

[41] Watson, B.W., "*The performance of single-keyword and multiple-keyword pattern matching algorithms*", *Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands*, 1994. Available for ftp from ftp.win.tue.nl in directory /pub/techreports/pi/pattm/.

[42] Watson, B.W., “An introduction to the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions”, *Computing Science Report* 94/21, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from ftp.win.tue.nl in directory /pub/techreports/pi/automata/.

[43] Watson, B.W., “The design and implementation of the FIRE Engine: A C++ toolkit for Finite automata and Regular Expressions”, *Computing Science Report* 94/22, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from ftp.win.tue.nl in directory /pub/techreports/pi/automata/.

[44] Watson, B.W., *“Taxonomies and Toolkits of Regular Language Algorithms”,* PhD Dissertation, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1995.

[45] Wood, D., *Theory of Computation*. (Harper & Row, New York, 1987).

[46] Yu., S., “Chapter 2, Regular languages”, *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa eds., Springer 1997.

[47] Gertjan van Noord, *FSA Reference Manual*, http://odur.let.rug.nl/~vannoord/Fsa/Manual/node1.html. Web page, March 2012.

[48] João Saraiva, *HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages*, http://www.di.uminho.pt/~jas/Research/HaLeX/HaLeX.html. Web site, March 2012

[49] Burak Emir, *AMoRE*, http://www.informatik.uni-kiel.de/inf/Thomas/amore.html. Web Site, March 2008.

[50] A3Software, http://aliagas.gr/Ale3hs/sc/lang.asp?language=perl. Web Site, June 2013

[51] Carrol, J., Long, D., *Theory of Finite Automata with an introduction to Formal Languages*, (Perntice Hall, Englewood Cliff, 1989).

[52] Teixeira, S., Pacheco, X., *Delphi 5 Developer's Guide*, (Sams Publishing, 1999).

[53] Calvert, C., *Delphi 2 Unleashed*, (Sams Publishing, 1996)

[54] Cantu, M., *Mastering Delphi 5*, (Sybex Inc , 1999)

[55] Brown, CW., Hardisty, EA., *RegeXeX: An Interactive System Providing Regular Expression Exercises*, ACM SIGCSE Bulletin inroads 39(1) (2007), 445-449

[56] Kopeinig, S., Thiemann, R., *Finite Automata Tool*,

http://cl-informatik.uibk.ac.at/software/fat/, Web Site, July 2013.

[57] Bovet, J., *Visual Automata Simulator*, http://www.cs.usfca.edu/~jbovet/vas.html, Web Site, July 2013.

[58] White, TM., Way, TP., *jFAST*:A Java Finite Automata Simulator, *ACM SIGCSE Bulletin inroads* 37 (2006), 58-62..

[59] Shawabkeh, M., *Automata Editor*, http://max99x.com/school/automata-editor, Web Site, July 2013.

Appendix

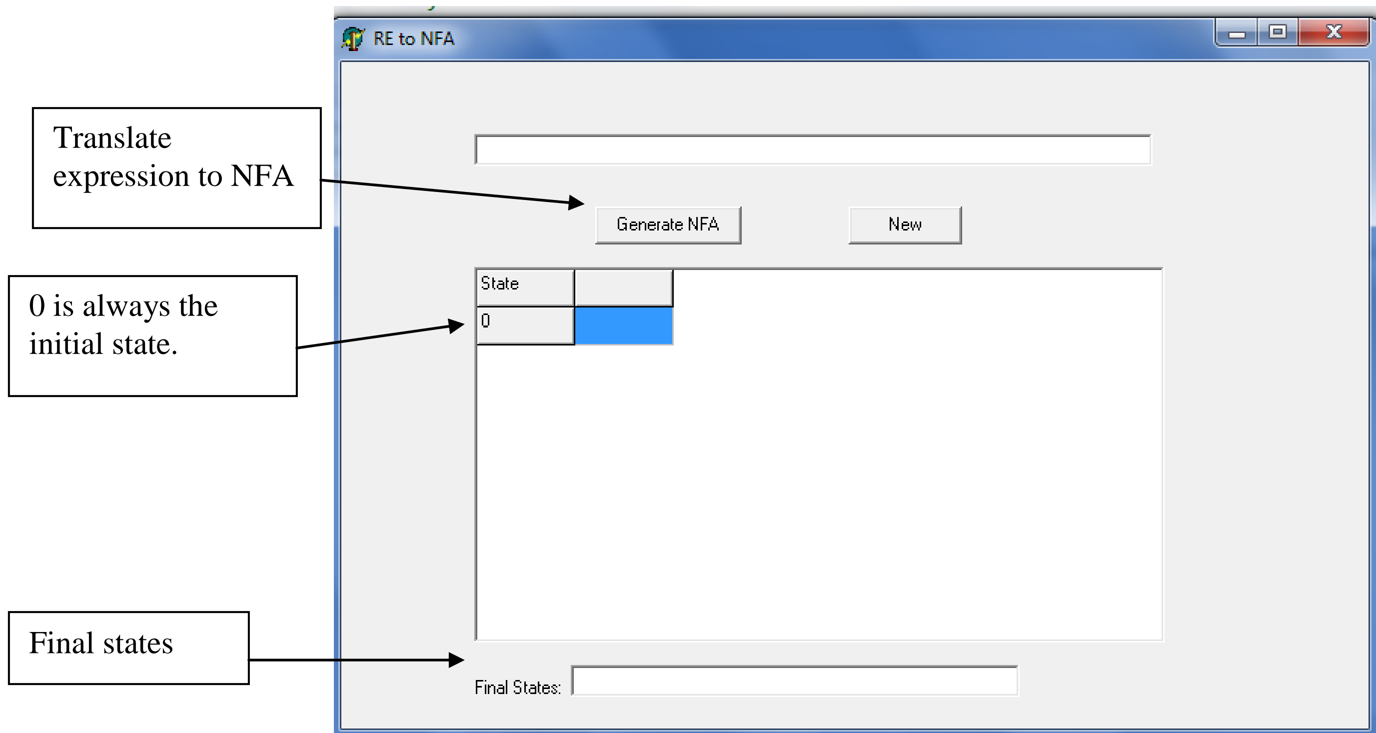
Appendix A: User Documentation

A1: Running the software.

In order to run the software, double click on the file RRFL.exe. This will start the program and will display the regular expression editor.

A2: Regular Expression Editor.

In the regular expression editor, the user has options to type in an expression and convert the expression to an NFA.



If the user wishes to enter an expression, the user will just type in the expression, using the "." key for concatenation, the "/" key for choice, and the "*" key for closure. Should the user need brackets, the "(" and ")" brackets should be used.

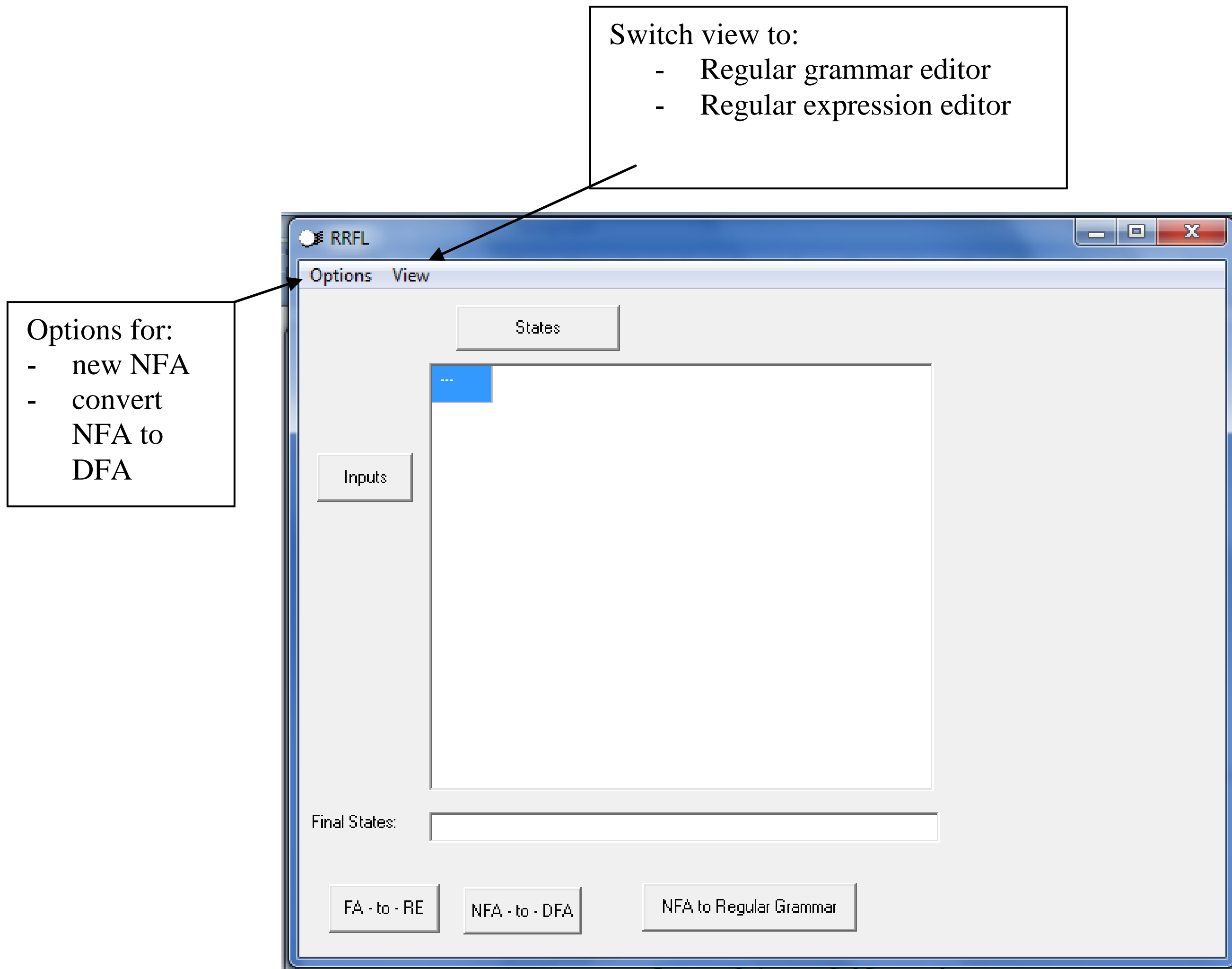
Once a regular expression has been entered, the user can now translate the expression to an NFA. This is done by clicking on the appropriate button on the screen.

A3: Finite Acceptor Editor.

In this editor, the user has options to type in an NFA, convert an NFA to a DFA, convert a DFA to a regular grammar, or to change the view to one of the other editors.

If the user chooses to enter a new NFA, the user must specify the transitions using a transition table. The user will click on the "States" button to add new states, which will be numbered automatically. To add a new input symbol, the user will first click on the "Inputs" button, which will add a new cell for the input symbol, and the user will then select this cell and type in the symbol for that input. Note that no new input symbols can be added if the previous input symbol is left blank. To add transitions from one state to another on a particular input, the user will select the cell at the intersection of the row and column combination of the state from where the transition will occur and the input symbol that the transition will be on, and the user will type in the state to which the transition will occur in this cell. Should the user wish to input more than one transition for a particular cell, the user will double click on that particular cell. A memo box will appear, where the user should type in the state for each transition, each on a separate line. The user will then click the "OK" button to return to the NFA grid. In this way the user will enter all transitions. Note that if the user chooses to enter a

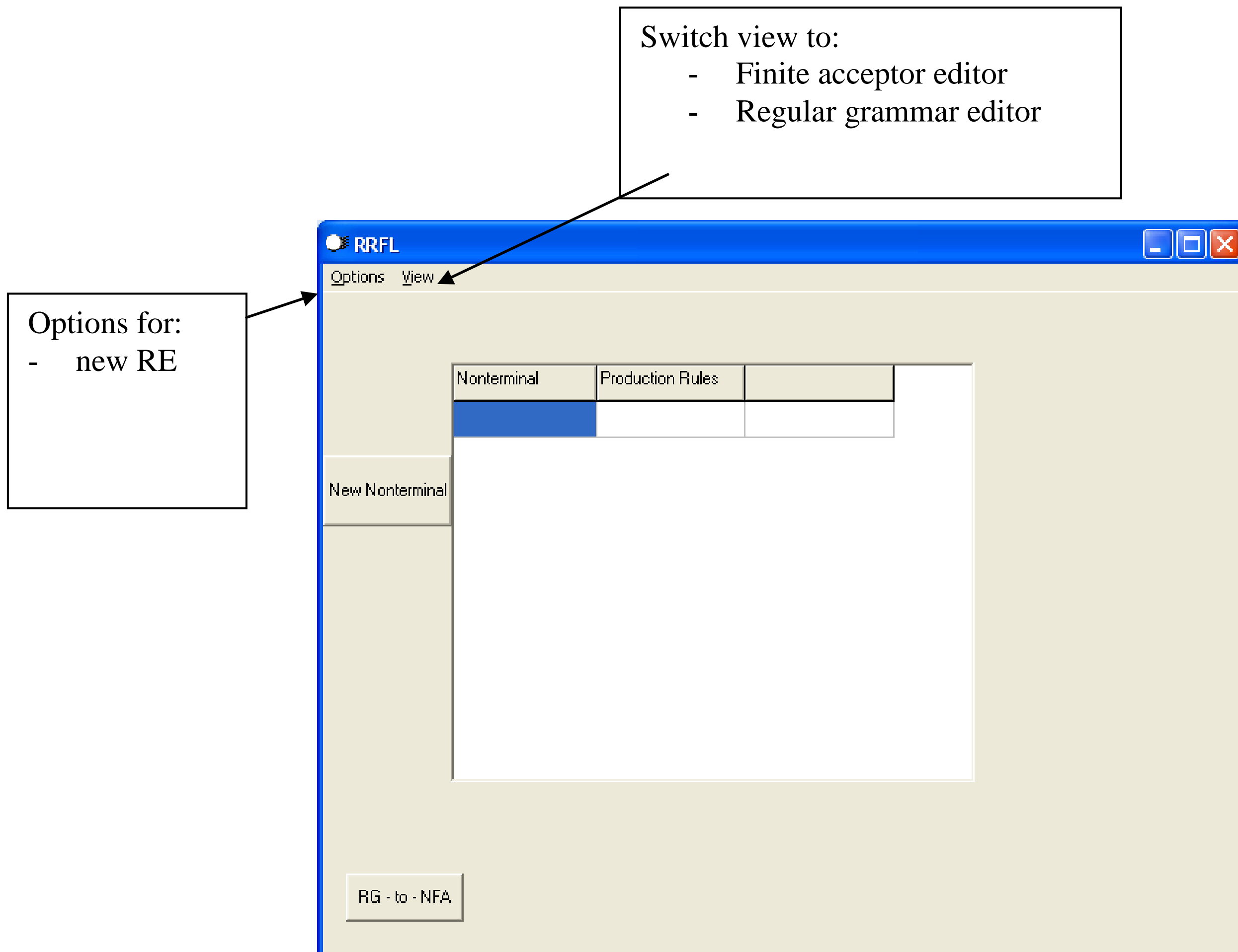
DFA, the user should enter this as an NFA, and the NFA to DFA conversion will result in the DFA being reproduced once the user clicks the appropriate button.



Once the NFA has been entered, the user can now choose to either convert the NFA to a DFA by either clicking on the appropriate option in the "Options" menu, or clicking on the appropriate button at the bottom of the screen, or to translate the NFA into a DFA and then to a regular grammar. This is done by clicking on the appropriate button at the bottom of the screen.

If the user wishes to switch to another editor view, the user will select the appropriate view from the "View" option in the menu bar.

A4: Regular Grammar Editor.



In this editor, the user has options to type in a regular grammar, translate the grammar to a NFA, or to change the view to one of the other editors.

If the user chooses to enter a new grammar, the user will specify the productions using a grid. The left most cell of each row is the nonterminal from which a production occurs, and

the cells to the right of this cell are the production rules on that nonterminal. The user can select the leftmost cell of each row and input the numeric label for the nonterminal. To add more nonterminals, the user will click on the "New Nonterminal" button, which will add a new row, and the user can label the leftmost cell of that row with the numeric label for that nonterminal. To add a production on a nonterminal, the user will select the first empty cell to the right of the nonterminal label, and enter the production rule. It should be noted that only valid rules will be allowed to be entered. Should the rightmost cell of a particular row be filled, and the user wishes to add more production rules for that particular nonterminal, the user will simply press on the right arrow button on the keyboard, and a new column will be added. This can only be done if the rightmost cell had been filled.

Once the grammar has been entered, the user can now choose to translate the grammar to an NFA. This is done by clicking on the appropriate button at the bottom of the screen.

If the user wishes to switch to another editor view, the user will select the appropriate view from the "View" option in the menu bar.