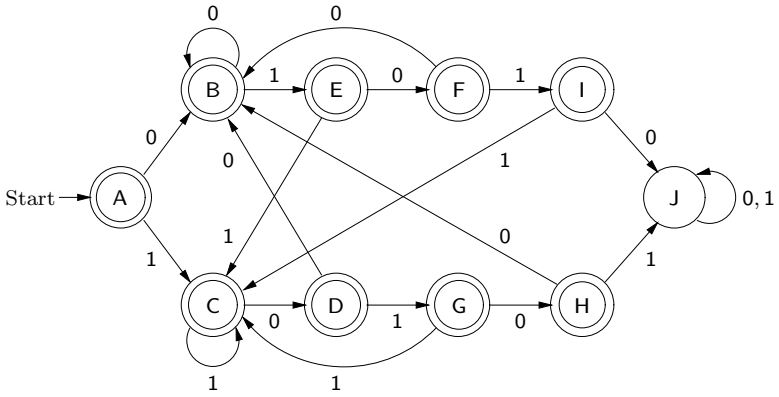
# Formal Language Theory

## Integrating Experimentation and Proof

**Alley Stoughton**

Draft of September 2018



# Contents

# List of Figures

# Preface

## Background

Since the 1930s, the subject of formal language theory, also known as automata theory, has been developed by computer scientists, linguists and mathematicians. Formal languages (or simply languages) are sets of strings over finite sets of symbols, called alphabets, and various ways of describing such languages have been developed and studied, including regular expressions (which "generate" languages), finite automata (which "accept" languages), grammars (which "generate" languages) and Turing machines (which "accept" languages). For example, the set of identifiers of a given programming language is a formal language—one that can be described by a regular expression or a finite automaton. And, the set of all strings of tokens that are generated by a programming language's grammar is another example of a formal language.

Because of its applications to computer science, most computer science programs offer both undergraduate and graduate courses in this subject. Perhaps the best known applications are to compiler construction. For example, regular expressions and finite automata are used when specifying and implementing lexical analyzers, and grammars are used to specify and implement parsers. Finite automata are used when designing hardware and network protocols. And Turing machines—or other machines/programs of equivalent power—are used to formalize the notion of algorithm, which in turn makes possible the study of what is, and is not, computable.

Formal language theory is largely concerned with algorithms, both ones that are explicitly presented, and ones implicit in theorems that are proved constructively. In typical courses on formal language theory, students apply these algorithms to toy examples by hand, and learn how they are used in applications. Although much can be achieved by a paper-and-pencil approach to the subject, students would obtain a deeper understanding of the subject if they could experiment with the algorithms of formal language theory using computer tools.

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize a deterministic finite automaton that accepts some language, $L$. With the paper-and-pencil approach, the student is obliged

to build the machine by hand, and then (hopefully) prove it correct. But, given the right computer tools, another approach would be possible. First, the student could try to express $L$ in terms of simpler languages, making use of various language operations (e.g., union, intersection, difference, concatenation, closure). The student could then synthesize automata accepting the simpler languages, enter these machines into the system, and then combine these machines using operations corresponding to the language operations used to express $L$. Finally, the resulting machine could be minimized. With some such exercises, a student could solve the exercise in both ways, and could compare the results. Other exercises of this type could only be solved with machine support.

## Integrating Experimentation and Proof

To support experimentation with formal languages, I designed and implemented a computer toolset called Forlan [Sto05, Sto08]. Forlan is implemented in the functional programming language Standard ML (SML) [MTHM97, Pau96], a language whose notation and concepts are similar to those of mathematics. Forlan is a library on top of the Standard ML of New Jersey (SML/NJ) implementation of SML [AM91]. It's used interactively, and users are able to extend Forlan by defining SML functions.

In Forlan, the usual objects of formal language theory—finite automata, regular expressions, grammars, labeled paths, parse trees, etc.—are defined as abstract types, and have concrete syntax. Instead of Turing machines, Forlan implements a simple functional programming language of equivalent power, but which has the advantage of being much easier to program in than Turing machines. Programs are also abstract types, and have concrete syntax. Although mainly *not* graphical in nature, Forlan includes the Java program JForlan, a graphical editor for finite automata and regular expression, parse and program trees. It can be invoked directly, or via Forlan.

Numerous algorithms of formal language theory are implemented in Forlan, including conversions between regular expressions and different kinds of automata, the usual operations (e.g., union) on regular expressions, automata and grammars, equivalence testing and minimization of deterministic finite automata, a general parser for grammars, etc. Forlan provides support for regular expression simplification, although the algorithms used are works in progress. It also implements the functional programming language used as a substitute for Turing machines.

This undergraduate-level textbook and Forlan were designed and developed together. I have attempted to keep the conceptual and notational distance between the textbook and toolset as small as possible. The book treats most concepts and algorithms both theoretically, especially using proof, and through experimentation, using Forlan.

In contrast to some books on formal language theory, the book emphasizes

the concrete over the abstract, providing numerous, fully worked-out examples of how regular expressions, finite automata, grammars and programs can be designed and proved correct. In my view, students are most able to learn how to write proofs—and to see the benefit of doing so—if their proofs are about things that they have designed.

I have also attempted to simplify the foundations of the subject, using alternative definitions when needed. E.g., finite automata are defined in such a way that they form a set (as opposed to a proper class), and so that more restricted forms of automata (e.g., deterministic finite automata (DFAs)) are proper subsets of that set. And finite automata are given semantics using labeled paths, from which the traditional $\delta$ notation is derived, in the case of DFAs. Furthermore, the book treats the set theoretic foundations of the subject more rigorously than is typical.

Readers of this book are assumed to have significant experience reading and writing informal mathematical proofs, of the kind one finds in mathematics books. This experience could have been gained, e.g., in courses on discrete mathematics, logic or set theory. The book assumes no previous knowledge of Standard ML. In order to understand and extend the implementation of Forlan, though, one must have a good working knowledge of Standard ML, as could be obtained by working through [Pau96] or [Ull98].

Drafts of this book were successfully used at Kansas State University in a semester long, undergraduate course on formal language theory.

## Outline of the Book

The book consists of five chapters. Chapter **??**, *Mathematical Background*, consists of the material on set theory, induction and recursion, and trees and inductive definitions that is required in the remaining chapters.

In Chapter **??**, *Formal Languages*, we say what symbols, strings, alphabets and (formal) languages are, show how to use various induction principles to prove language equalities, and give an introduction to the Forlan toolset. The remaining three chapters introduce and study more restricted sets of languages.

In Chapter **??**, *Regular Languages*, we study regular expressions and languages, five kinds of finite automata, algorithms for processing and converting between regular expressions and finite automata, properties of regular languages, and applications of regular expressions and finite automata to searching in text files, lexical analysis, and the design of finite state systems.

In Chapter **??**, *Context-free Languages*, we study context-free grammars and languages, algorithms for processing grammars and for converting regular expressions and finite automata to grammars, top-down (recursive descent) parsing, and properties of context-free languages. We prove that the set of context-free languages is a proper superset of the set of regular languages.

Finally, in Chapter **??**, *Recursive and Recursively Enumerable Languages*, we study the functional programming language that we use instead of Turing machines to define the recursive and recursively enumerable languages. We study algorithms for processing programs and for converting grammars to programs, and properties of recursive and recursively enumerable languages. We prove that the set of context-free languages is a proper subset of the set of recursive languages, that the set of recursive languages is a proper subset of the set of recursively enumerable languages, and that there are languages that are not recursively enumerable. Furthermore, we show that there are problems, like the halting problem (the problem of determining whether a program halts when run on a given input), that can't be solved by programs.

## Further Reading and Related Work

This book covers most of the material that is typically presented in an undergraduate course on formal language theory. On the other hand, typical textbooks on formal language theory cover much more of the subject than we do. Readers who are interested in learning more, or who would like to be exposed to alternative presentations of some of the material in this book, should consult one of the many fine books on formal language theory, such as [HMU01, Koz97, LP98, Mar91, Lin01].

Neil Jones [Jon97] pioneered the use of a programming language with structured data as an alternative to Turing machines for studying the limits of what is computable. In Chapter **??**, we have followed Jones's approach in some ways. On the other hand, our programming language is functional, not imperative (assignment-oriented), and it has explicit support for the symbols and strings of formal language theory.

The existing formal languages toolsets fit into two categories. In the first category are tools, like JFLAP [BLP$^+$97, HR00, Rod06], Pâté [BLP$^+$97, HR00], the Java Computability Toolkit [RHND99], and Turing's World [BE93], that are graphically oriented and help students work out relatively small examples. The books [Rod06] (on JFLAP) and [Lin01] (an introduction to formal language theory) are intended to be used in conjunction with each other. The second category consists of toolsets that, like Forlan, are embedded in programming languages, and so support sophisticated experimentation with formal languages. Toolsets in this category include Automata [Sut92], Grail+ [RW94, RWYC17], HaLeX [Sar02], Leiß's Automata Library [Lei10] and Vaucanson [LRGS04]. I am not aware of other textbook/toolset packages whose toolsets are members of this second category.

## Notes, Exercises and Website

In the "notes" subsections that conclude most sections of the book, I have restricted myself to describing how the book's approach differs from standard practice. Readers interested in the history of the subject can consult [HMU01, Koz97, LP98].

The book contains numerous fully worked-out examples, many of which consist of designing and proving the correctness of regular expressions, finite automata, grammars and programs. Similar exercises, as well as other kinds of exercises, are scattered throughout the book.

The Forlan website

`http://alleystoughton.us/forlan`

contains:

- instructions for downloading and installing the Forlan toolset, and JForlan;
- the Forlan manual;
- instructions for reporting errors or making suggestions; and
- the Forlan distribution, including the source for Forlan and JForlan, as well as the LaTeX source for this book.

## Acknowledgments

Leonard Lee and Jessica Sherrill designed and implemented graphical editors for Forlan finite automata (JFA), and regular expression and parse trees (JTR), respectively. Their work was unified and enhanced (of particular note was the addition of support for program trees) by Srinivasa Aditya Uppu, resulting in an initial version of JForlan. Subsequently, Kenton Born carried out a major redevelopment of JForlan, resulting in JForlan Version 1.0. A further revision, by the author, led to JForlan Version 2.0.

It is a pleasure to acknowledge helpful discussions relating to the Forlan project with Eli Fox-Epstein, Brian Howard, Rod Howell, John Hughes, Nathan James, Patrik Jansson, Jace Kohlmeier, Dexter Kozen, Matthew Miller, Aarne Ranta, Ryan Stejskal, Colin Stirling, Lucio Torrico and Lyn Turbak. Much of this work was done while I was employed by Kansas State University, and some of the work was done while I was on sabbatical at the Department of Computing Science of Chalmers University of Technology.

# Bibliography


[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1991.

[BE93] J. Barwise and J. Etchemendy. *Turing's World 3.0 for Mac: An Introduction to Computability Theory*. Cambridge University Press, 1993.

[BLP+97] A. O. Bilska, K. H. Leider, M. Procopiuc, O. Procopiuc, S. H. Rodger, J. R. Salemme, and E. Tsang. A collection of tools for making automata theory and formal languages come alive. In *Twenty-eighth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 15–19. ACM Press, 1997.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, second edition, 2001.

[HR00] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, pages 6–10. ACM Press, 2000.

[Jon97] N. J. Jones. *Computability and Complexity: From a Programming Perspective*. The MIT Press, 1997.

[Koz97] D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1997.

[Lei10] H. Leiß. The Automata Library. `http://www.cis.uni-muenchen.de/~leiss/sml-automata.html`, 2010.

[Lin01] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, 2001.

[LP98] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, second edition, 1998.

[LRGS04] S. Lombardy, Y. Régis-Gianas, and J. Sakarovitch. Introducing vaucanson. *Theoretical Computer Science*, 328:77–96, 2004.

[Mar91] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw Hill, second edition, 1991.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised 1997*. The MIT Press, 1997.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[RHND99] M. B. Robinson, J. A. Hamshar, J. E. Novillo, and A. T. Duchowski. A Java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In *Thirtieth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 105–109. ACM Press, 1999.

[Rod06] S. H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, 2006.

[RW94] D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17:341–350, 1994.

[RWYC17] D. Raymond, D. Wood, S. Yu, and C. Câmpeanu. Grail+: A symbolic computation environment for finite-state machines, regular expressions, and finite languages. `http://www.csit.upei.ca/~ccampeanu/Grail/`, 2017.

[Sar02] J. Saraiva. HaLeX: A Haskell library to model, manipulate and animate regular languages. In *ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)*, Pittsburgh, October 2002.

[Sto05] A. Stoughton. Experimenting with formal languages. In *Thirty-sixth ACM SIGCSE Technical Symposium on Computer Science Education*, page 566. ACM Press, 2005.

[Sto08] A. Stoughton. Experimenting with formal languages using Forlan. In *FDPE '08: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 41–50, New York, NY, USA, 2008. ACM.

[Sut92] K. Sutner. Implementing finite state machines. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 15, pages 347–363. American Mathematical Society, 1992.

[Ull98] J. D. Ullman. *Elements of ML Programming: ML97 Edition*. Prentice Hall, 1998.

# Index