

Preface

Copyright © 2003–7 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The \LaTeX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

Background

Since the 1930s, the subject of formal language theory, also known as automata theory, has been developed by computer scientists, linguists and mathematicians.

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 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).

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 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.

Background (Cont.)

Because of its many applications to computer science, e.g., to compiler construction, most computer science programs offer both undergraduate and graduate courses in this subject.

Background (Cont.)

Because of its many applications to computer science, e.g., to compiler construction, most computer science programs offer both undergraduate and graduate courses in this subject.

Many of the results of formal language theory are proved constructively, using algorithms that are useful in practice. In typical courses on formal language theory, students apply these algorithms to toy examples by hand, and learn how they are used in applications. But they are not able to experiment with them on a larger scale.

Background (Cont.)

Because of its many applications to computer science, e.g., to compiler construction, most computer science programs offer both undergraduate and graduate courses in this subject.

Many of the results of formal language theory are proved constructively, using algorithms that are useful in practice. In typical courses on formal language theory, students apply these algorithms to toy examples by hand, and learn how they are used in applications. But they are not able to experiment with them on a larger scale.

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.

Background (Cont.)

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize an automaton that accepts some language, L .

Background (Cont.)

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize an automaton that accepts some language, L .

With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (perhaps) prove that it is correct.

Background (Cont.)

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize an automaton that accepts some language, L .

With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (perhaps) prove that it is correct.

But, given the right computer tools, another approach would be possible.

Background (Cont.)

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize an automaton that accepts some language, L .

With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (perhaps) prove that it is 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 (union, intersection, difference, concatenation, closure).

Background (Cont.)

Consider, e.g., a typical exercise of a formal language theory class in which students are asked to synthesize an automaton that accepts some language, L .

With the paper-and-pencil approach, the student is obliged to build the machine by hand, and then (perhaps) prove that it is 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 (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 .

Background (Cont.)

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

Since 2001, I have been designing and developing a computer toolset, called Forlan, for experimenting with formal languages.

Integrating Experimentation and Proof

Since 2001, I have been designing and developing a computer toolset, called Forlan, for experimenting with formal languages.

Forlan is implemented in the functional programming language Standard ML, a language whose notation and concepts are similar to those of mathematics.

Integrating Experimentation and Proof

Since 2001, I have been designing and developing a computer toolset, called Forlan, for experimenting with formal languages.

Forlan is implemented in the functional programming language Standard ML, a language whose notation and concepts are similar to those of mathematics.

Forlan is used interactively. In fact, a Forlan session is simply a Standard ML session in which the Forlan modules are pre-loaded. Users are able to extend Forlan by defining SML functions.

Integrating Experimentation and Proof (Cont.)

In Forlan, the usual objects of formal language theory—automata, regular expressions, grammars, labeled paths, parse trees, etc.—are defined as abstract types, and have concrete syntax. Leonard Lee and Jessica Sherrill designed and implemented JFA, a graphical editor for finite automata, and JTR a graphical editor for regular expression and parse trees, respectively; both are written in Java.

Integrating Experimentation and Proof (Cont.)

In Forlan, the usual objects of formal language theory—automata, regular expressions, grammars, labeled paths, parse trees, etc.—are defined as abstract types, and have concrete syntax. Leonard Lee and Jessica Sherrill designed and implemented JFA, a graphical editor for finite automata, and JTR a graphical editor for regular expression and parse trees, respectively; both are written in Java.

The standard algorithms of formal language theory are implemented in Forlan, including conversions between different kinds of automata and grammars, the usual operations on automata and grammars, equivalence testing and minimization of deterministic finite automata, etc.

Integrating Experimentation and Proof (Cont.)

The introductory textbook on formal language theory *Formal Language Theory: Integrating Experimentation and Proof* and these associated lecture slides are based on Forlan.

Integrating Experimentation and Proof (Cont.)

The introductory textbook on formal language theory *Formal Language Theory: Integrating Experimentation and Proof* and these associated lecture slides are based on Forlan.

The textbook goes into more detail than these lecture slides.

Integrating Experimentation and Proof (Cont.)

The introductory textbook on formal language theory *Formal Language Theory: Integrating Experimentation and Proof* and these associated lecture slides are based on Forlan.

The textbook goes into more detail than these lecture slides.

I am attempting to keep the conceptual and notational distance between the textbook and toolset as small as possible.

Integrating Experimentation and Proof (Cont.)

The introductory textbook on formal language theory *Formal Language Theory: Integrating Experimentation and Proof* and these associated lecture slides are based on Forlan.

The textbook goes into more detail than these lecture slides.

I am attempting to keep the conceptual and notational distance between the textbook and toolset as small as possible.

The book treats each concept or algorithm both theoretically, especially using proof, and through experimentation, using Forlan.

Integrating Experimentation and Proof (Cont.)

The introductory textbook on formal language theory *Formal Language Theory: Integrating Experimentation and Proof* and these associated lecture slides are based on Forlan.

The textbook goes into more detail than these lecture slides.

I am attempting to keep the conceptual and notational distance between the textbook and toolset as small as possible.

The book treats each concept or algorithm both theoretically, especially using proof, and through experimentation, using Forlan.

Readers of the book are assumed to have a significant amount of 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.

Integrating Experimentation and Proof (Cont.)

The book assumes no previous knowledge of Standard ML.

Integrating Experimentation and Proof (Cont.)

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 one of the following books:

- L. C. Paulson, *ML for the Working Programmer*, second edition, Cambridge University Press, 1996.
- J. D. Ullman, *Elements of ML Programming: ML97 Edition*, Prentice Hall, 1998.

Outline of the Book

The book consists of five chapters. Chapter 1, *Mathematical Background*, consists of the material on set theory, induction principles for the natural numbers, and trees and inductive definitions that is required in the remaining chapters.

Outline of the Book

The book consists of five chapters. Chapter 1, *Mathematical Background*, consists of the material on set theory, induction principles for the natural numbers, and trees and inductive definitions that is required in the remaining chapters.

In Chapter 2, *Formal Languages*, we say what symbols, strings, alphabets and (formal) languages are, introduce and show how to use several string induction principles, and give an introduction to the Forlan toolset. The remaining three chapters introduce and study more restricted sets of languages.

Outline of the Book

The book consists of five chapters. Chapter 1, *Mathematical Background*, consists of the material on set theory, induction principles for the natural numbers, and trees and inductive definitions that is required in the remaining chapters.

In Chapter 2, *Formal Languages*, we say what symbols, strings, alphabets and (formal) languages are, introduce and show how to use several string induction principles, and give an introduction to the Forlan toolset. The remaining three chapters introduce and study more restricted sets of languages.

In Chapter 3, *Regular Languages*, we study regular expressions and languages, four 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 and lexical analysis.

Outline of the Book (Cont.)

In Chapter 4, *Context-free Languages*, we study context-free grammars and languages, algorithms for processing grammars and for converting regular expressions and finite automata to grammars, and properties of context-free languages.

Outline of the Book (Cont.)

In Chapter 4, *Context-free Languages*, we study context-free grammars and languages, algorithms for processing grammars and for converting regular expressions and finite automata to grammars, and properties of context-free languages.

It turns out that the set of all context-free languages is a proper superset of the set of all regular languages.

Outline of the Book (Cont.)

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study a universal programming language based on Lisp, which we use to define the recursive and recursively enumerable languages.

Outline of the Book (Cont.)

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study a universal programming language based on Lisp, which we use 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.

Outline of the Book (Cont.)

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study a universal programming language based on Lisp, which we use 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.

It turns out that the context-free languages are a proper subset of the recursive languages, that the recursive languages are a proper subset of the recursively enumerable languages, and that there are languages that are not recursively enumerable.

Outline of the Book (Cont.)

Finally, in Chapter 5, *Recursive and Recursively Enumerable Languages*, we study a universal programming language based on Lisp, which we use 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.

It turns out that the context-free languages are a proper subset of the recursive languages, that the recursive languages are a proper subset of the recursively enumerable languages, and that there are languages that are not recursively enumerable.

Furthermore, there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Further Reading and Related Work

This book covers the core material that is typically presented in an undergraduate course on formal language theory. On the other hand, a typical textbook on formal language theory covers much more of the subject than we do. Readers who are interested in learning more about the subject, or who would like to be exposed to alternative presentations of the material in the book, should consult one of the many fine books on formal language theory, such as:

Further Reading and Related Work

This book covers the core material that is typically presented in an undergraduate course on formal language theory. On the other hand, a typical textbook on formal language theory covers much more of the subject than we do. Readers who are interested in learning more about the subject, or who would like to be exposed to alternative presentations of the material in the book, should consult one of the many fine books on formal language theory, such as:

- J. E. Hopcroft and R. Motwani and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, second edition, Addison-Wesley, 2001.
- H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, second edition, Prentice Hall, 1998.
- J. C. Martin, *Introduction to Languages and the Theory of Computation*, second edition, McGraw Hill, 1991.

Further Reading and Related Work (Cont.)

The existing formal language toolsets fit into two categories.

Further Reading and Related Work (Cont.)

The existing formal language toolsets fit into two categories.

In the first category are tools like JFLAP, Pâté the Java Computability Toolkit, and Turing's World that are graphically oriented and help students work out relatively small examples.

Further Reading and Related Work (Cont.)

The existing formal language toolsets fit into two categories.

In the first category are tools like JFLAP, Pâté the Java Computability Toolkit, and Turing's World that are graphically oriented and help students work out relatively small examples.

The second category consists of toolsets that, like Forlan, are embedded in programming languages, and so that support sophisticated experimentation with formal languages. Toolsets in this category include Automata, Grail+, HaLeX, Leiß's Automata Library and Vaucanson.

Further Reading and Related Work (Cont.)

The existing formal language toolsets fit into two categories.

In the first category are tools like JFLAP, Pâté the Java Computability Toolkit, and Turing's World that are graphically oriented and help students work out relatively small examples.

The second category consists of toolsets that, like Forlan, are embedded in programming languages, and so that support sophisticated experimentation with formal languages. Toolsets in this category include Automata, Grail+, HaLeX, Leiß's Automata Library and Vaucanson.

I am not aware of any other textbook/toolset packages whose toolsets are members of this second category.

Acknowledgments

Leonard Lee and Jessica Sherrill designed and implemented JFA, the Java program for editing finite automata, and JTR, the Java program for editing regular expression and parse trees, respectively. It is a pleasure to acknowledge helpful conversations or e-mail exchanges relating to this textbook/toolset project with Brian Howard, Rodney Howell, John Hughes, Nathan James, Patrik Jansson, Jace Kohlmeier, Dexter Kozen, Aarne Ranta, Ryan Stejskal and Colin Stirling. Some of this work was done while I was on sabbatical at the Department of Computing Science of Chalmers University of Technology.