A Development Environment for Formal Languages

Thomas E. O'Neil
Computer Science Department
University of North Dakota
oneil@cs.und.edu

Abstract

Formal languages and Automata provide models for study of the fundamental principles and properties of computers and computer programs. The models, by definition, are universal and abstract, and their relation to practical computing is not immediately obvious. The simplicity and relevance of these models become clearer when they are implemented as computational objects in some programming language. This paper describes a collection of Java objects that implement formal languages, automata, and grammars in a hands-on development environment. The central classes, called FormalLanguage and LanguageLab, provide a user interface for creation, viewing, and manipulation of languages, grammars, and automata.

A FormalLanguage object is a possibly infinite set of strings over some finite alphabet of symbols. The Alphabet class is an ordered set of characters. An instance of a FormalLanguage can have a Generator, which has methods that produce an ordered list of the strings in the language, and an Acceptor, which has a method that takes strings as input and determines whether they belong to the language. The Grammar class is an extension of the Generator class, and the Automaton class is an extension of the Acceptor class. Subclasses of Grammar and Automaton can be defined to implement regular and context-free grammars, finite automata, push-down automata, and Turing machines. The LanguageLab contains facilities for creating, testing, and modifying grammars and automata.

The LanguageLab has methods for converting generators to acceptors and vice-versa. The challenge of implementing these methods and of creating generators and acceptors for various languages quickly raises the major issues in computability and decidability. This collection of Java objects provides a framework and a robust set of tools for a hands-on exploration of the theory of computing.

Introduction

One of the primary goals of any undergraduate computer science program is to make students aware that there are severe limitations on computational systems. The world of computing would reap huge benefits if general-purpose programs could be written to analyze other programs and determine such properties such as program equivalence, correctness, and reliability. In courses on the theory of computation, we try to demonstrate that such programs do not exist. This endeavor requires a simple, unified model for computational problems and the programs that are written to solve them. There are several models that can be employed, but taking the approach of formal languages, we represent all programs and computational problems as possibly infinite sets of strings using some fixed alphabet of symbols. This model, by definition, is universal and abstract, and its relation to practical computing may not be immediately obvious. The simplicity and relevance of the model, however, becomes clearer when it is implemented in a development environment that allows creation and testing of specific languages using grammars, automata, and other formal systems that specify sets of strings. This paper describes such an environment, implemented in Java (JDK 1.2) for a Linux operating environment.

The central classes in the formal language development environment are called FormalLanguage and LanguageLab. A FormalLanguage object is a possibly infinite set of strings over some finite alphabet of symbols, implemented by the Alphabet class. Every FormalLanguage object must have an Alphabet object and a StringSpecifier object that defines what strings are in the language. StringSpecifiers can have fixed or variable alphabets. StringSpecifier has subclasses called Generator and Acceptor. Generators produce sequences of strings and acceptors test individual strings for acceptance or rejection. A FormalLanguage can be defined to have just a Generator, just an Acceptor, or both a Generator and an Acceptor.

The LanguageLab class provides the user interface for creating, testing, loading, and storing FormalLanguages, Acceptors, and Generators. The testing interface provides a mechanism for both acceptance and generation of strings, even when the language does not have both kinds of StringSpecifier. This is accomplished by using generators to implement rudimentary (and inefficient) acceptors, and vice versa.

The FormalLanguage class has a subclass FiniteLanguage for finite sets of strings. The Generator object for a finite language can produce a complete list of strings, and the list can be searched to implement a simple Acceptor. With infinite languages, however, no complete list exists, and Generators and Acceptors must be implemented as grammars, automata, or other algorithms that recognize or express properties and repeatable patterns within strings. The LanguageLab has editing frames for various common generation and acceptance systems, including regular and context-free grammars, finite automata, pushdown automata, and Turing machines.

The FormalLanguage, Alphabet, and StringSpecifier Classes

A formal language is a possibly infinite set of strings over some finite alphabet of symbols. The FormalLanguage class encapsulates this definition. Each FormalLanguage object has an Alphabet, a name, a description, a Generator, and an Acceptor. Strings in the language are retrieved or tested by running the Generator or the Acceptor. The Generator and Acceptor are both extensions of the abstract class StringSpecifier, and at least one of them must be non-null for the FormalLanguage object to be fully defined. Some FormalLanguages have both an Acceptor and a Generator. When this is the case, there is no guarantee that the Acceptor and Generator actually specify the same set of strings. Beyond verifying that the alphabets are the same, independently designed Generators and Acceptors are not checked for consistency. If a user defines both a Generator and an Acceptor for a language, it is the user's responsibility to assure their consistency. This is obviously a flaw in our automated system, but it is unavoidable. We have already encountered one of those problems for which no general solution exists.

The Alphabet class encapsulates a finite set of symbols. It is an extension of the Java ArrayList class. It has an *add*(char *c*) method for adding characters so that alphabets can be built incrementally, and it has a Boolean method *contains*(char *c*) that tests whether the alphabet contains a character *c*. We normally view alphabets as ordered sets, so that the order of the alphabet can be used to determine order among strings over the alphabet. To implement an ordering of the symbols, the Alphabet class has four methods that return characters: *getFirstSymbol*(), *getLastSymbol*(), *getSuccessor*(char *c*), and *getPredecessor*(char *c*). Finally, the class has a *getSize*() method that returns the number of symbols in the alphabet, and a *getAlphaString*() method that returns the string formed by concatenating all the alphabet characters in order.

The abstract StringSpecifier class has an Alphabet and a type as member components, and it has methods for setting and getting these components. The type is simply a string that indicates how the StringSpecifier is implemented. The class also has an abstract Boolean method that tells whether or not the Alphabet is fixed. Some StringSpecifiers may depend only on alphabet-independent properties of strings, such as string length. Such StringSpecifier objects can allow their alphabets to change.

The StringSpecifier class is extended as either a Generator or an Acceptor. Both of these extending classes are abstract. The Generator class has two abstract methods that return strings: *getFirst*(), which returns the first string in the language, and *getNext*(), which returns the next string in the language after the previously generated string. Thus a Generator incrementally produces a list of strings. The class provides no specification of the order of the list. The only essential requirement in designing a language is that the Generator produces a string if and only if it is in the language.

The Acceptor class has a single abstract Boolean method *accepts*(String *s*) and a concrete Boolean method *alphaTest*(String *s*). The *alphaTest*() method is intended to be used in the implementation of the *accepts*() method to screen input strings for symbols that are not in the alphabet. The *accepts*() method takes any string as a parameter and returns

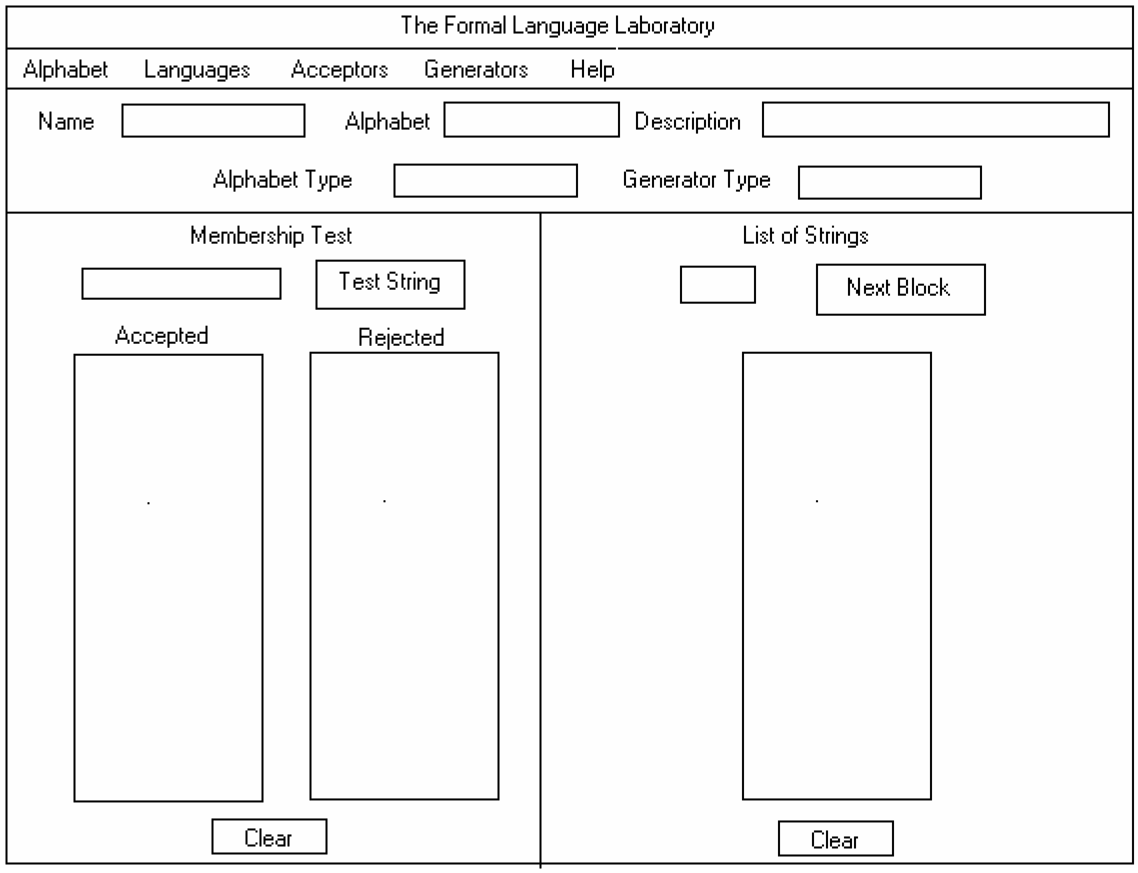

Figure 1. The LanguageLab interface.

true if the string is in the language. We would hope that the method would also return false if the string is not in the language, but this is not required. The minimum requirement is that the accepts(*s*) method returns true if and only if string *s* is in the language.

The LanguageLab Class

The LanguageLab class implements an interface for experimenting with formal languages (see Figure 1). Information about the language currently being viewed or developed is displayed in the top panel of the LanguageLab frame. This information includes the Alphabet, the name and description of the language, and the names and types of the Generator and Acceptor for the language. The remainder of the frame is divided into two panels, one dedicated to running the Acceptor and keeping a history of the results, and the other dedicated to running the Generator and keeping a list of the results. The Acceptor testing panel has a text box for entering a string to be tested, and a button to activate the Acceptor. Depending on the results, the string is placed in a list box of accepted strings or a list box of rejected strings. The panel also has a button to clear the contents of the list boxes. The Generator test panel has a text box that allows the user to specify how many strings to generate. After entering the number of strings desired, the user presses a button that invokes the Generator repeatedly and places each string generated in a list box. This panel also has a button to clear the contents of the list box.

As described above, a FormalLanguage object must have an Acceptor or a Generator, but it does not need to have both. If either of these StringSpecifier objects is missing, the LanguageLab will attempt to build the other to provide a robust testing environment.

To perform an acceptance test using the Generator, the LanguageLab iteratively generates strings, starting with a call to *getFirst*(). Each string is compared with the test string, and if it matches, the test string is accepted. Of course, we might have a problem if the test string is not in the language – nothing will match and the LanguageLab will have to decide when to stop calling the Generator to get the next string in the language. If the Generator is guaranteed to produce strings in non-decreasing order, the testing can stop when the first string longer than the test string is produced by the Generator. However, not all Generators have this property, so our automated system has another significant flaw reflecting another major limitation of formal systems.

To build a Generator from an Acceptor, the LanguageLab needs the help of a general-purpose, predefined FormalLanguage that contains all the strings over a specified alphabet. This language, called AlphaStar, is packaged with the FormalLanguage class. It has a predefined Generator called AllGenerator (packaged with the Generator class) and a predefined Acceptor called AllAcceptor (packaged with the Acceptor class). Both are implemented directly in Java code. The *accepts*(String *s*) method of AllAcceptor simply calls *alphaCheck*(*s*) and returns the results. The *getFirst*() method of AllGenerator returns the empty string, and the *getNext*() method uses the alphabet to build the next string in lexicographic order. To build a Generator from some Acceptor, LanguageLab creates the AlphaStar language using the alphabet of the Acceptor. It then repeatedly gets strings from AlphaStar's Generator and uses the Acceptor to filter them. This will result in a Generator that produces strings in lexicographic order. A problem will arise only if the Acceptor is not total – that is, if the Acceptor does not halt on all inputs. Again, we come face-to-face with a fundamental limitation of formal systems.

The LanguageLab has a menu bar that can be used to invoke pop-up frames for creating and editing Alphabets, Acceptors, and Generators. It also allows FormalLanguages, Acceptors, and Generators to be loaded from files (or classes) and stored as files. File formats for FormalLanguages and most types of Acceptors and Generators are defined and implemented in the input/output methods.

Extensions of the Acceptor Class

The user of LanguageLab can create a language by defining an extension to the Acceptor class. There are two options for defining an Acceptor. The first option is to write Java code for a class that extends Acceptor and implements its abstract method. Such Acceptors should return the string "Custom" as the Acceptor type. The AlphaStar language mentioned in the previous section is an example of a language with a custom

Acceptor. Custom Acceptors are defined outside of the LanguageLab. The user writes the code with a text editor, compiles it, and places the ".class" file in some directory. When running LanguageLab, the user can select the "Load Class" option from the Acceptor menu to browse for the class to be loaded. The selected class is then instantiated by LanguageLab as the Acceptor for a new FormalLanguage object.

The second option for creating an Acceptor is to build an Automaton. The Automaton class is an abstract extension of Acceptor that provides a parent class for state transition machines such as finite automata, pushdown automata, and Turing machines. It has fields and methods held in common by all its subclasses. Every Automaton has a set of states and a set of state transition rules that are used to process an input word. One of the states is designated to be the start state. It has concrete methods for getting and setting the state set, for getting and setting the transition rules, for setting the start state, and for getting the current state. It also has abstract methods *changeState*(), *run*(), *getCurrentSymbol*(), and *initialize*(String *s*). An Automaton computation is started by calling the *initialize*() method with an input word. The computation can proceed one step at a time with calls to *changeState*(), or it run from current state to completion with a call to *run*().

There are several classes representing the components of Automaton. The class State defines an object that has a name and can be designated to be final or non-final. The StateSet class is an extension of the Java ArrayList class that is used to store states. It has all the methods of the ArrayList class, plus methods that allow states to be accessed by name. An abstract class called Tuple and a class called Rule are used to represent state transition rules. Every Tuple has a State and possibly some other symbols as components. Every Rule is a pair of Tuples: an input Tuple and an output Tuple. The input Tuple describes the current state of an Automaton, and the output Tuple specifies how to move to the next state and possibly update other machine components. The class OrderedRuleList represents a set of Rules. It has methods for adding and deleting Rules and a method for retrieving the output Tuple for any given input Tuple. The set of transition rules for every Automaton is implemented as an OrderedRuleList.

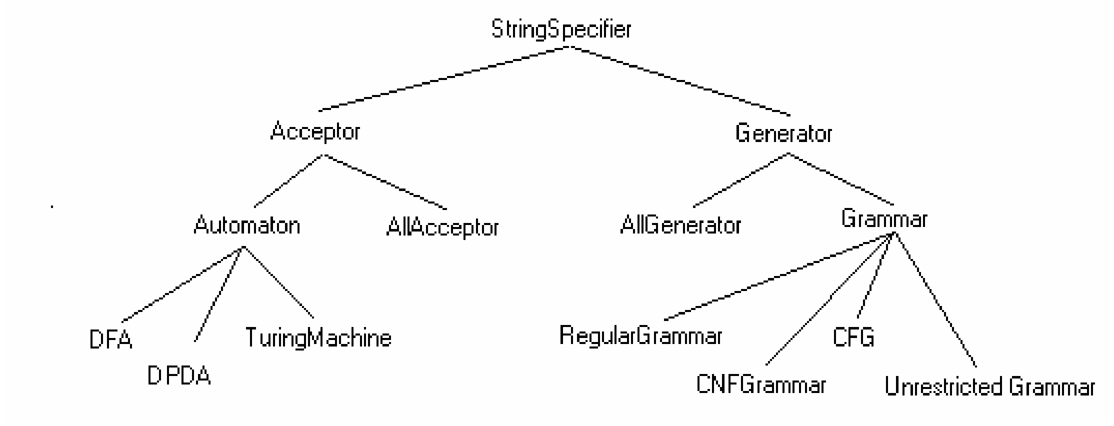

Figure 2. The StringSpecifier class hierarchy.

The abstract class Automaton is extended by concrete classes DFA (for deterministic finite automata), DPDA (for deterministic pushdown automata), and TuringMachine (see Figure 2). DFAs have Rules in which the input Tuple is a State/symbol pair, and the output Tuple is a State singleton. DFAs have no internal storage medium. For DPDA Rule sets, the input Tuples are triples with a State and two symbols, and the output Tuples are State/string pairs. DPDAs have a stack for an internal storage medium. For TuringMachine Rules, both the input Tuple and the output Tuple are State/symbol pairs. TuringMachines have a tape for an internal storage medium.

The DFA class is packaged with a DFAFrame class that allows editing of deterministic finite automata (see Figure 3). The editing frame displays the Alphabet, the State set, and the list of Rules. The Alphabet and State set are updated as new Rules are added. States can be selected from the list to be designated as final states or as the start state. There is a text box for an input word and a control button that allows step-by-step monitoring of a computation. When the user is ready for more comprehensive testing, he/she presses an "Apply" button to create a new FormalLanguage with the new DFA as its acceptor. Control is then returned to the LanguageLab frame, where further acceptance and

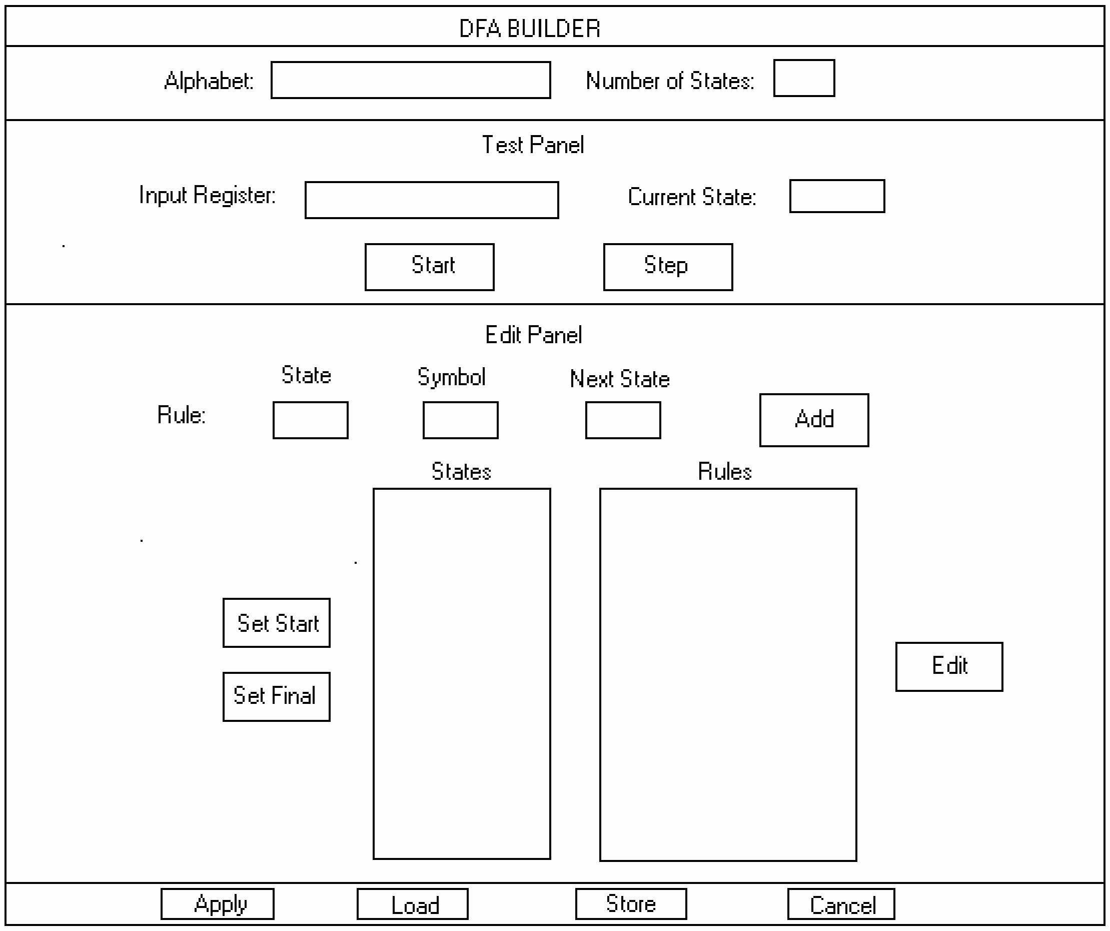

Figure 3. The DFA editing frame.

generation tests can be conducted. The editing frame also allows DFAs to be loaded from and stored to files. PDAs and TuringMachines have editing frames with similar capabilities.

Extensions of the Generator Class

LanguageLab users can also create new languages by defining new Generators. As with Acceptors, custom Generators can be written directly in Java code and loaded into the LanguageLab as Java classes. Alternatively, users can take advantage of the abstract Grammar class, which extends the Generator class. The Grammar class is a parent to concrete subclasses for the common types of grammars such as regular grammars (RegularGrammar), context-free grammars (CFG), Chomsky normal form grammars (CNFGrammar), and unrestricted grammars (UnrestrictedGrammar). Every Grammar has an Alphabet, a set of NonTerminals, a set of rewriting Rules, and a start symbol. Supporting classes are defined to implement these components. The different kinds of Grammars are more uniform than different kinds of Automata. All grammars use the same mechanism for generating strings. The only variations are in the form of the rules.

The editing frame for Grammars shows the Alphabet, the NonTerminals, and the list of Rules. These components are updated automatically as new rules are added. The frame provides a mechanism for deriving strings by applying one rule at a time. As with Automata, more comprehensive tests can be conducted in the LanguageLab frame after the Grammar has been designed and applied. The editing frame has buttons for loading and storing Grammars in the file system.

The OrderedGenerator and TotalAcceptor Interfaces

As discussed in a previous section, the *getNext*() method of Generator objects will produce an ordered list of strings when called iteratively, but there are no specifications for what the order will be. We strongly prefer a Generator that can produce strings in order of non-decreasing length. This greatly simplifies the problem of determining whether a string is in the language. Our collection of StringSpecifier classes contains a Java interface called OrderedGenerator, which Generators can implement to designate that they produce strings in non-decreasing order. OrderedGenerators have the methods of the Generator class plus three additional methods: *getFirst*(int *length*), *getLast*(int *length*), and *getStringsOfLength*(int *length*). The *getFirst*() and *getLast*() methods return the first and last strings, respectively, with the specified length. The *getStringsOfLength*() method returns a Java LinkedList containing all the strings in the language of the specified length. Whenever a user can program these methods in a custom Generator, it should be defined to implement the OrderedGenerator interface. Among the predefined Grammar subclasses, the RegularGrammar and CNFGrammar classes are designated to implement the OrderedGenerator interface.

The Acceptor class has a related interface called TotalAcceptor. As discussed earlier, there are no guarantees that the *accepts*(*s*) method will halt for strings that are not in the Acceptor's language. With TotalAcceptors, however, we do have that guarantee. The interface has the same methods as the Acceptor class with one addition: *rejects*(String *s*). An Acceptor may be designated to implement the TotalAcceptor interface if it has a *rejects*(*s*) method that returns true if and only if string *s* is not in its language. Whenever a user creates a custom Acceptor whose *accepts*() method always halts, the user should define a rejects() method that returns the opposite of the *accepts*() method and designate the Acceptor to implement the TotalAcceptor interface. Of the predefined subclasses of Automaton, DFA and PDA are designated to implement TotalAcceptor.

Conclusion

The theory of computation is necessarily abstract material, and most students have difficulty mastering it in their first exposure to it. The LanguageLab is intended to place the abstractions in a concrete development environment, in hopes that hands-on experimentation will help students build a mental model of computation. The LanguageLab can provide a laboratory component for any course in formal languages and automata, using standard textbooks (e.g. Flanagan, 1999, and Hopcroft, Motwani, and Ullman, 2001). Students will achieve a more thorough grasp of grammars and automata by building and testing them in the laboratory. They can also be sent on a quest for OrderedGenerators and TotalAcceptors for various computational problems, such as determining whether two grammars or automata are equivalent with respect to the languages they define. This quest will bring them face to face with the fundamental limitations of computational systems. They will discover that a system powerful enough to encapsulate infinite sets of strings is loaded with problems for which there are no complete computational solutions.

References

Flanagan, D. (1999). *Java in a Nutshell: A Desktop Quick Reference*. Sebastopol, CA: O'Reilly and Associates.

Hopcroft, J., R. Motwani, and J. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Addison-Wesley.

Lewis, H, and C. Papadimitriou (1998). *Elements of the Theory of Computation*. Upper Saddle River, NJ: Prentice Hall.