View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Rowan University

Rowan University

Rowan Digital Works

Theses and Dissertations

7-22-2002

A technique for converting NFA's and DFA's to regular expressions in Lex

Gregory Safko
Rowan University

Follow this and additional works at: https://rdw.rowan.edu/etd

Part of the Higher Education Commons

Let us know how access to this document benefits you - share your thoughts on our feedback form.

Recommended Citation

Safko, Gregory, "A technique for converting NFA's and DFA's to regular expressions in Lex" (2002). *Theses and Dissertations*. 1506.
https://rdw.rowan.edu/etd/1506

This Thesis is brought to you for free and open access by Rowan Digital Works. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Rowan Digital Works. For more information, please contact LibraryTheses@rowan.edu.

A TECHNIQUE FOR CONVERTING NFA'S AND DFA'S TO REGULAR EXPRESSIONS IN LEX

by
Gregory Safko

A Thesis

Submitted in partial fulfillment of the requirements of the
Master of Arts Degree
of
The Graduate School
at
Rowan University
July 19, 2002

Approved by ______________________
Professor

Date Approved 7/22/2002

ABSTRACT

Gregory Safko, A Technique For Converting NFA's And DFA's To Regular Expressions In Lex, 2002, Seth Bergmann, Masters in Higher Education in Computer Science.

In the design and creation of compilers, a compiler-writer is called upon to make assumptions and declarations about how their language's syntax performs and behaves. Lex is a software tool used to create grammar rules. A flat file is read as input to lex, and the corresponding C code is output. However, in order to properly format the flat file, the compiler-writer creates the lex expression either on a "hunch", or based on some given set of criteria. Among these criteria are: case sensitivity, the introduction of special characters and symbols, the handling of a space character, and the handling of escape sequence characters, namely the tab, new-line, and carriage return symbols. Lex can create the code to perform such verification, but it is up to the compiler-writer to provide lex with the rules. Sometimes the rules can be seen more easily diagrammatically through an NFA or a DFA. Once the FA is created, it is just a matter of extracting the rules from the diagram. The following technique can be employed to create lex rules from the Finite Automata.

MINI-ABSTRACT

Gregory Safko, A Technique For Converting NFA's And DFA's To Regular Expressions In Lex, 2002, Seth Bergmann, Masters in Higher Education in Computer Science.

The technique described in this thesis converts a Finite Automata (as either a DFA or an NFA), to a regular expression, then writes it to a lex-formatted flat file, to be used as a lex input file.

ACKNOWLEDGEMENTS

The author would like to acknowledge the selfless contributions of his graduate advisor, Seth Bergmann. His legacy of study with his advisors and professors from the University of Pennsylvania provided an invaluable source of first hand knowledge in the field of Formal Language Theory and Compiler Design.

TABLE OF CONTENTS

Chapter 1: Background Research 20

Chapter 2: DFA and NFA Basics 20

Chapter 3: DFA/NFA Conversion Techniques....................20

Chapter 4: Lex rules....................20

Chapter 5: Properties of Regular Expressions....................20

Chapter 6: The Technique for the Conversion from a DFA to lex input....................20

Chapter 7: Software to generate lex patterns....................20

Chapter 8: Future refinements 20

Appendix A: Java source code 20

LIST OF TABLES AND CHARTS

figure 2.1 9

figure 2.2 10

figure 2.3 10

figure 3.1 11

figure 3.2 12

figure 3.3 13

figure 4.1 17

figure 4.2 18

figure 4.3 19

figure 4.4 19

figure 5.1 24

figure 5.2 24

figure 5.3 25

figure 5.4 27

figure 6.1 29

figure 6.2 30

figure 6.3 30

figure 6.4 31

figure 7.1 32

figure 7.2 32

figure 7.3 32

figure 7.4 33

figure 8.1 52

Chapter 1: Background Research

Lex is a programming tool that assists a programmer in writing programs and compilers from structured input rules. The rules follow a specific pattern, or grammar, and this grammar can be thought of as being similar to linguistic grammar, found among the spoken languages on the planet. Research in the 1950's by Noam Chomsky (with respect to spoken grammars) and in the 1960's by John Backus and Peter Naur (with respect to computer grammars) have laid the foundation with which computer programming languages are structured. With similarities to natural languages, computer-programming languages are built to accept and follow a specific set of rules and patterns. Unlike spoken languages, computer languages are quite specific; there is no direct corollary to a spoken language's "dialect", "colloquialism", or "intent". Thus, a computer programming language is quite specific in it's own set of rules, grammars, lexicon, and construction. But, like a spoken (or written) language, we expect to derive a specific sense or connotation from the sentence we read or hear. We can tell almost instantly if the sentence makes sense, or if something is "missing" (wrong verb, incorrect tense, incorrect agreement in numbers, etc.). The same is true for programming languages. If a statement ("sentence" in linguistics) does not have the correct structure ("nouns, verbs, adjectives, etc." in linguistics), then the compiler terminates execution, and the code generation is stopped. The program will not compile successfully until the offending line (or lines) of code is corrected.

For computer languages to function intuitively, and to behave as expected certain rules of format and construction must be followed. Each rule is unique to each language. However once a base set of rules are established, the language generation and usage is pretty straight-forward.

The research and works of Noam Chomsky and Backus and Naur provide insight into the construction and format of languages, as well as to their syntactical generation and use. Where the Chomsky rules apply to human linguistics, the research of Backus and Naur has direct implications to formal programming languages. It is these rules that lex tools use as a model. The Backus-Naur Form (or BNF) is not only important in describing syntax rules in books, but it is also very commonly used (with variations) by syntactic tools (such as lex and yacc).

The same vocabulary used to describe human linguistics is also used by programming languages. Linguistic vocabulary words such as "syntax", "sentence", "grammar", and "word" have similar connotations in programming languages, with similar behavior and construction. Lex is used to create the "words" of a language. "Words" in this paper will be understood to mean keywords and variable names, operation symbols, and other character-based symbols associated with programming languages. However, the list of these appropriate words in a programming language is vastly larger than those found in spoken languages. With Lex as the authoring tool of these languages, a writer of compilers could use it to create the appropriate syntax for these for the programming language. The use of an NFA (Non-deterministic Finite Automata) or a DFA (Deterministic Finite Automata) could assist in creating scenarios of appropriate and inappropriate words for this language. As background, any NFA or a

DFA is a graphical or tabular representation of how characters can be accepted by this "machine" (automaton). If the programmer could construct the finite automata (FA), then this diagram could be converted, via algorithms described in this paper, to acceptable lex statements. Although this does not create the entire programming language, this does, in fact, create the acceptable grammars of the variables and strings used in the language.

As a side note, a DFA is more straightforward an automaton than an NFA. A DFA processes input and seeks a single path; NFA's (by their name: non-deterministic) create machines in which several paths could be taken based on input. If an incorrect path is found, backtracking occurs, and an alternate path (if available) is explored.

The sources consulted were books by the following authors: Aho, Bergmann, Brown, Estier, Hein, Hopcroft, Hutton, Levine, Mason, Parsons, Sethi, and Ullman. All of these authors (with the exception of Estier) discuss FA's in their books. The goal of this paper will be (assuming no NFA exists) to construct the NFA, convert it to a DFA, convert the DFA to tabular (matrix) form, and process this table using the algorithm presented to a lex rule. A program will also be used to create a flat data file that lex can recognize as an input file.

Chapter 2: DFA and NFA Basics

The author assumes that the reader is moderately familiar with the concept of NFA's and DFA's. For a thorough discussion on FA's, consult the books by Hein or Hopcroft and Ullman. All FA's have the following in common (in one form or another): a starting state, an input vocabulary, transition states, transition edges, and acceptance states.

There are many techniques that can be used to convert NFAs to DFAs. If it cannot be converted by inspection alone, we can use any number of techniques available to us. Hopcroft and Ullman provide an example of how we can convert the weaker-cased NFA to the stronger-cased DFA.

Here are a few brief notes about DFA's and NFA's. First, all DFA's are NFA's (but not all NFA's are DFA's). When an NFA is said to be "weaker" than a DFA, it means that a DFA gives you no choice about a transition from one state to another; an NFA could make choices, and if that choice is not correct, it could backtrack to find a different path to determine if that might be a correct choice. And finally, in a DFA all possibilities of input characters from a given alphabet have a definite path to follow; in an NFA if an input character from the alphabet does not have a corresponding path, we assume that it goes to a "dead state" (i.e. a state from which we have no hope of continuing to success).

In conclusion to this chapter, the following examples are used to demonstrate DFA's and NFA's pictorially. First, we present a diagram of a simple DFA that is used to

verify if we have a valid identifier (using the conventional understanding that identifiers must begin with a letter, and can be followed by any number of letters and digits). The abbreviation "a" for alpha characters (a through z and A through Z) and "d" for digit (0 through 9) shall be used in this diagram.

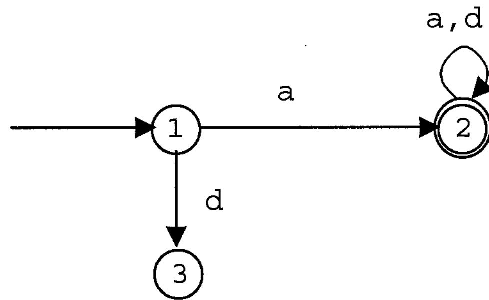

figure 2.1

The above diagram, as an NFA is as follows (note the absence of the "dead state"):

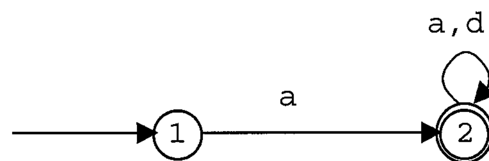

figure 2.2

The following is an NFA that can make decisions between two states based on the same input character (in this case, reading input "a" while in state 3 leads to two choices of subsequent states):

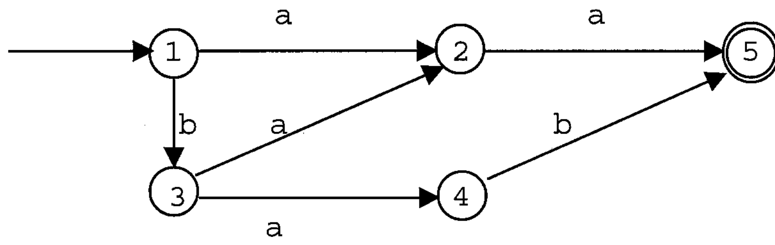

figure 2.3

Chapter 3: DFA/NFA Conversion Techniques

Pictorially, a DFA (Deterministic Finite Automata) can be described as a finite directed graph over a finite alphabet A, with a series of connected nodes (vertices) and directed edges. There is exactly one labeled edge from each node for each distinct element in A. The nodes are called states, and every DFA has an initial (or start) state, and at least one final (or accepting) state (Heim). Given alphabet {a,b} the following DFA (figure 3.1) accepts a string made up of zero or more a's, followed by one or more b's.

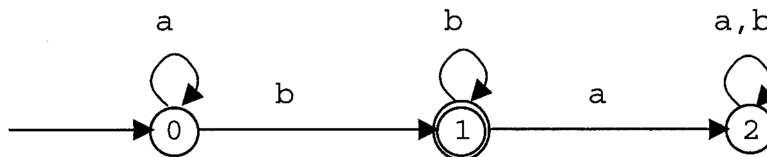

figure 3.1

The node labeled by the number 0 designates the initial state (in this DFA), and the node labeled by the number 1 and the double concentric circles designate the final state. Node 2 exists since, by the definition of a DFA, all nodes must have exactly one labeled edge for each distinct member of the alphabet. The strings {ab, aaabbb, abbb, bbb} are accepted by this DFA. The strings {aaa, abab, abbabb} are not accepted by this DFA. The NFA representation of this diagram is shown (figure 3.2) as follows:

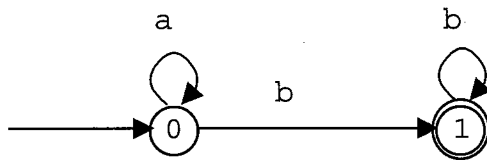

figure 3.2

Note the absence of node 2, and the edges that point to it. This falls under the definition of an NFA, since state 1 is missing an edge representing the alphabet element “a”.

For now we will proceed with the understanding that the reader is familiar with NFA/DFA Theory. The Hopcroft & Ullman book and the Heim book provide and describe the concept in further detail. The only issue of concern is regarding the behavior of an NFA. Since, by its nature, it is non-deterministic, it strays from the definition of a DFA in the following manner: there can be more than one (or no) labeled edge from each node for each distinct element or series of elements in A. This conversion technique will also work for NFAs. It should be noted that we will call the NFA in figure 3.2 a quasi-DFA (or q-DFA), since it violates the definition of a DFA given that it does not have an edge labeled “a” coming from node 1, and going to node 2 (which, of course means that edges labeled “a” and “b” must come from node 2 and go back to node 2, as in figure 3.1).

NFAs and DFAs (and q-DFAs) can also be represented in tabular (matrix) form. The matrix will contain enough columns to represent the number of elements in A. The matrix will also contain enough rows to represent the number of states. Using the DFA in figure

3.1, we have 2 columns, and three rows. An asterisk will prefix an accepting state. Figure 3.1 in tabular form looks as follows:

	a	b
0	0	1
*1	2	1
2	2	2

figure 3.3

Conversely, given a table, an FA can easily be created. The tabular form of the DFA/qDFA will be used for the lex conversion technique.

Chapter 4: Lex rules

Lex is a software tool used to generate code to test the acceptance or rejection of a string pattern. Once an acceptance rule of a pattern is found, it can be used in a compiler-generator (such as yacc) to test for, among other things, the validity of variable names, the declaration of keywords, and the proper use of conglomerate operands (i.e., operands that, when concatenated, take on a different computational purpose. Some conglomerate operands, among many, in C++ are the !=, <=, ->, and the ++.)

The following is a list of patterns that, when interpreted by lex, take on a specific meaning.

Pattern	Meaning
c	The char "a"
"c"	The char "c" even if it is a special char in this table
\c	Same as "c", used to quote a single char. Especially useful when the special char is a lex directive
[cd]	The char "c" or the char "d"
(cd)	The combined pair "cd"
[a-z]	Any single char in the range a through z (this also includes any other ASCII range, but is more appropriately useful as a range or numbers or case sensitive letters)
[^c]	Any char but c

.	Any char but newline
^x	The pattern x if it occurs at the beginning of a line.
x$	The pattern x at the end of a line.
x?	An optional x
x*	Zero or more occurrences of the pattern x
x+	One or more occurrences of the pattern x
xy	The pattern of x concatenated with the pattern y
x|y	An x or a y
x/y	An x only if followed by a y
<S>x	The pattern x when lex is in start condition S
{name}	The value of a macro definition from definitions section
x{m}	m occurrences of the pattern x
x{m,n}	m through n occurrences of the pattern x (takes precedence over the concatenation pattern)

Lex also allows the pattern to perform any side-effect operations (such as reporting if a string is invalid, or if it's a keyword)

Just as in a programming language, the input file must be written in a particular format, or lex will not recognize it. The body of a lex input file has three sections, and each section is delineated by the symbol %%; we will only concern ourselves with the second section. The reader is referred to Levine (Chapter 1) or Bergmann (Chapter 2, Section 2.4) for further discussions on the first and third section. The skeleton body of a lex file is shown in figure 4.1:

```
/* Section 1 - the directives and definitions
An optional section
*/
%%
/* Section 2 - the rules
A required section
*/
%%
/* Section 3 - user defined code
An optional section
*/
```

figure 4.1

If we desired to write an input file that would be used to test for valid variable names (using the C/C++ rules for variable names), it would look as follows:

```
%%
[a-zA-Z][a-zA-Z0-9]* printf("valid variable name\n");
[0-9].               printf("invalid variable name\n");
%%
```

figure 4.2

Notice that this would make sure that the first character is alphabetic (either upper or lower case), and any subsequent character is alphabetic or numeric. (For sake of simplicity, we will ignore allowing special characters in the variable name).

When this code is run, the user can type a string and press <Enter>. The code will report (via the following examples)

Input	Output
ab1	valid variable name
123	invalid variable name
A1	valid variable name
1A	invalid variable name

The q-DFA corresponding to this is as follows:

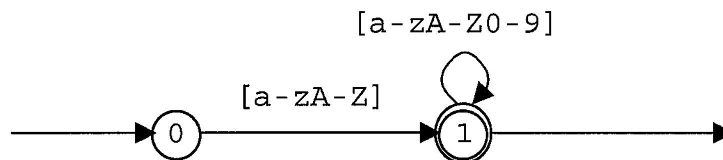

figure 4.3

Note that if, upon initial input, the string contains a digit, the FA will not accept the string. This unspecified transition state is sometimes referred to as a "dead state" (a state from which there is no success at accepting the string). This is noted as {deadstate} in the tabular form:

	a-z	A-Z	0-9
0	1	1	{deadstate}
*1	1	1	1

figure 4.4

The table in figure 4.4 is a trivial example. Some tables can be quite complex in their analysis of the input. The next chapter will describe some properties of regular expressions, as well as presenting an established algorithm for converting automata to regular expressions.

Chapter 5: Properties of Regular Expressions.

In order to simplify a regular expression, we need to look at the properties of `regular` expressions. Although they appear algebraic in form, these rules require a bit of intuition if we wish to correctly use them to simplify (if possible) a larger regular expression.

The first series of rules applies to regular expressions in the form of strings, using R, S, and T as strings, and declaring ε (epsilon) as a null string, and ∅ (null) as the empty set. As a note, an epsilon string has a length of zero, whereas an empty set is a set of no elements. With respect to concatenation, the epsilon can be considered the identity element of regular expressions, and the empty set can be considered the zero element.

Union (+) Properties

$R + T = T + R$ (property 5.1)

$R + \varnothing = \varnothing + R = R$ (property 5.2)

$R + R = R$ (property 5.3)

$(R + S) + T = R + (S + T)$ (property 5.4)

$R^* + \varepsilon = R^*$ (property 5.5)

Concatenation (.) Properties

$R\varnothing = \varnothing R = \varnothing$ (property 5.6)

$R\varepsilon = \varepsilon R = R$ (property 5.7)

$(RS)T = R(ST)$ (property 5.8)

(Note, in almost all cases $RS \neq SR$)

$R(S + T) = RS + RT$ (property 5.9)

$(R + S)T = RT + ST$ (property 5.10)

Closure (*) Properties

(Recall that R* is zero or more occurrences of R, i.e. $R^* = \{\varepsilon, R, RR, RRR \ldots\}$)

$\varnothing^* = \varepsilon^* = \varepsilon$ (property 5.11)

$R^* = R^*R^*$ (property 5.12)

$R^* = (R^*)R^*$ (property 5.13)

$(RS)^*R = R(SR^*)$ (property 5.14)

$(R + S)^* = (R^* + S^*)^*$ (property 5.15)

$= R^*(R + S)^*$ (property 5.16)

$= (R + SR^*)^*$ (property 5.17)

$= (R^*S^*)^*$ (property 5.18)

$= R^*(SR^*)^*$ (property 5.19)

$= (R^*S)^*R^*$ (property 5.20)

Some texts also define the unary + operator (not to be confused with the + operator in union properties) as:

$R+ = R \cdot R^*$ (property 5.21)

Distributive Properties

$R(S + T) = RS + RT$ (property 5.22)

$(S + T)R = SR + ST$ (property 5.23)

Miscellaneous Rules and Properties

Some of these rules are derived via inspection, rather than as a series of concrete rules. Terms in the strings can either be dropped (due to redundancy), or incorporated (due to consumption) into larger, general-purpose strings. (These will not be listed with property numbers since they can be derived from the previous properties)

$(RT^*S + RS) = (RT^*S)$ (The string RS can be derived from RT*S, making the union redundant)

$(RR)^*$ + (any even numbers of the string R) = $(RR)^*$

$R(RR)^*$ + (any odd numbers of the string R) = $R(RR)^*$

$(R + S + T)^*$(any expression) = Any string that ends with (any expression)

The following theorem (from Hopcroft and Ullman, p.33) states that:

If a language L is accepted by a DFA, then L is denoted

by a regular expression.

Without going into the formal proof of the theorem, we need to see that there exists a series of strings within the finite automaton that can take it from state q_i to q_j without going through a state higher than k (*i*,*j*, and *k* are arbitrary here, but need to be declared initially in the automaton). The numbers represented by *i* or *j* may be greater than *k*, so long as the transition does not pass through *k*. (Passing through a state means both entering and leaving that state.)

We can represent this set of strings as R_{ij}^{k} .

We can define R_{ij}^{k} recursively:

$$R_{ij}^{k} = R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^{*} R_{kj}^{k-1} \cup R_{ij}^{k-1}$$

$$R_{ij}^{0} = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j, \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{if } i = j, \end{cases}$$

And the language L is represented by

$$\bigcup R^{n}_{1j}$$

which is the union of all strings that take you from start state 1 to accepting state j without going through any state higher than n.

We can apply this algorithm to Figure 5.1 (Taken from Hopcroft and Ullman, page 35), which demonstrates a simple DFA.

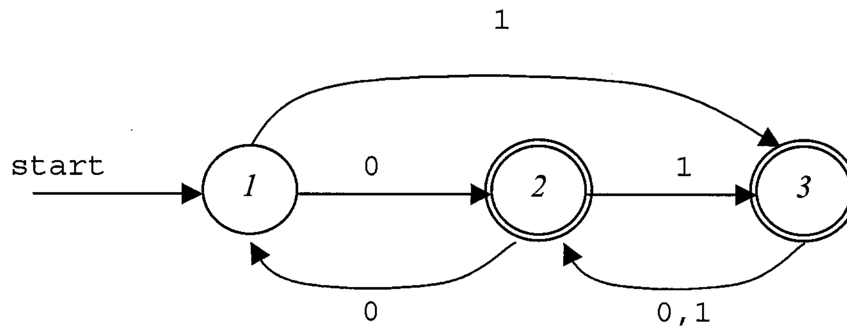

figure 5.1

In tabular form, the DFA of figure 5.1 is as follows:

	0	1
1	2	3
*2	1	3
*3	2	2

figure 5.2

Using the theorem and the algorithm (from Hopcroft and Ullman), we can create a table with the following parameters:

The number of rows = 9 (the number of states squared, which represents the number of possible transitions from a state to another state, including itself). The row states start at r_{11} and go to r_{33}. The state labels are filled in the chart by starting at r_{xy}, and moving x-wise y-wise fashion. Therefore, r_{11} is placed first, then r_{12}, r_{13}, r_{21}, r_{22}, r_{23}, r_{31}, r_{32}, and finally r_{33}.

The number of columns = 4 (the number of states plus the 0 state, which is the base case)

The table is filled in row-wise column-wise fashion. Therefore, r^0_{11} is filled first, then r^0_{12} next, until we reach r^3_{33}.

	k = 0	k = 1	k = 2
r^k_{11}	ε	ε	$(00)^*$
r^k_{12}	0	0	$0(00)^*$
r^k_{13}	1	1	0^*1
r^k_{21}	0	0	$0(00)^*$
r^k_{22}	ε	$\varepsilon + 00$	$(00)^*$
r^k_{23}	1	$01 + 1$	0^*1
r^k_{31}	$\varnothing$	$\varnothing$	$(0 + 1)(00)^*0$
r^k_{32}	$(0 + 1)$	$(0 + 1)$	$(0 + 1)(00)^*$
r^k_{33}	ε	ε	$(0 + 1)0^*1 + \varepsilon$

figure 5.3

When $k = 0$ (the base case), the table is easily filled in; you jut put in the symbol(s) which take you from *i* to *j*. The remaining entries are built from the recursive definition of r^k_{ij}.

As an example, the expression for r^1_{22} is:

$$r^1_{22} = r^0_{21}(r^0_{11})^* r^0_{12} + r^0_{22} = 0(\varepsilon)^* 0 + \varepsilon = \varepsilon + 00$$

The expression for r^2_{13} is:

$$\begin{aligned} r^2_{13} = r^1_{12}(r^1_{22})^* r^1_{23} + r^1_{13} &= 0(\varepsilon + 00)^*(1 + 01) + 1 \\ &= 0^+(1 + 01) + 1 \\ &= 0^+ 1 + 0^* 1 + 1 \\ &= 0^* 1 \end{aligned}$$

To complete the construction of the regular expression, we just observe that we need to make it from state 1 (the start state) to either state 2 or 3 (both accepting states). This is defined (from the algorithm) as $r^3_{12} \cup r^3_{13}$, (or $r^3_{12} + r^3_{13}$) which is:

$$\begin{aligned} r^3_{12} = r^2_{13}(r^2_{33})^* r^2_{32} + r^2_{12} &= 0^*1(\varepsilon + (0 + 1)0^*1)^*(0 + 1)(00)^* + 0(00)^* \\ &= 0^*1((0 + 1)0^*1)^*(0 + 1)(00)^* + 0(00)^* \end{aligned}$$

$$\begin{aligned} r^3_{13} = r^2_{13}(r^2_{33})^* r^2_{33} + r^2_{13} &= 0^*1(\varepsilon + (0 + 1)0^*1)^*(\varepsilon + (0 + 1)0^*1) + 0^*1 \\ &= 0^*1((0 + 1)0^*1)^* \end{aligned}$$

and

$$r^3_{12} + r^3_{13} \qquad = 0^*1((0 + 1)0^*1)^*(\varepsilon + (0 + 1)(00)^*) + 0(00)^*$$

figure 5.4

Described in words, this DFA accepts all strings that have an odd number of zeros (from the string 0(00)*) or any string that begins with any number of zeros (including none), followed by a 1 (from the string 0*1) concatenated with a zero or a 1 followed by any combination of 1's and zeros, including none, but at least one 1 when a zero is present (from the string ((0 + 1)(0*1)*)) concatenated with nothing or a 1 or a zero followed by an even number of zeros, including none (from the string (ε + (0 + 1)(00)*)). This is a very complex expression, and certainly difficult to create without the help of the algorithm. With such a DFA, a syntax checker (such as lex) could be used to create code to verify if a string conforms to the above expression. However the conversion (as has been seen) is not trivial.

The goal of the next chapter is to show how to convert the DFA in figure 5.1 or the table in figure 5.2 via a table such as figure 5.3 to lex input (based on an expression such as figure 5.4)

Chapter 6: The Technique for the Conversion from a DFA to lex input.

Consider the string from figure 5.4 with the following rules:

$$0^*1((0 + 1)0^*1)^*(\varepsilon + (0 + 1)(00)^*) + 0(00)^*$$

figure 6.1

The goal of this chapter is to convert this string to lex input syntax. The premise behind these expressions is that the reserved lex symbols (from chapter 3, namely +,*,[,], parends, /, etc.) are not used in the regular expression (although there may arise an occasions where they would be used). For now, use a unique substitution parameter, and then replace it later with "c" where c is the special reserved symbol.

Notice that the Kleene * operation is identical in both the regular expression, just as it is in lex input (meaning zero or more occurrences of the string or character). So, when we encounter *, we just leave them alone. If the algorithm behaves appropriately, then there will be no ambiguity between writing 00* when we really mean (00)*.

The union operation + could be represented in lex via one of two ways: the expression (0 + 1) could be written as either (0|1) or [01]. The expression (0 + 1 + 2) could be written

as (0|1|2) or [012]. We could have also said [0-2], and achieved the same results. Using the bracket (i.e. [02]) approach, figure 6.1 is now rewritten as:

$$0^*1([01]0^*1)^*(\varepsilon + [01](00)^*) + 0(00)^*$$

figure 6.2

We now need to address the epsilon (ε) in the expression. Recall that ε is equivalent to the identity element. Therefore it is a simple matter of applying a distributive property (valid under the rules of regular expressions) to figure 6.2 to obtain:

$$0^*1([01]0^*1)^* + 0^*1([01]0^*1)^*([01](00)^*) + 0(00)^*$$

or

$$0^*1([01]0^*1)^* + 0^*1([01]0^*1)^*[01](00)^* + 0(00)^*$$

figure 6.3

If there were a null (∅) in the expression, then this would behave as the zero element, and would affect the subexpressions accordingly.

Therefore, figure 6.3 would appear in lex as follows:

```
%%
0*1([01]0*1)* + 0*1([01]0*1)*[01](00)* + 0(00)*
                    printf("accepted by figure 5.1\n");
%%
```

figure 6.4

A computer program could then be used to convert a DFA (figure 5.1) to tabular form (figure 5.3), to create a regular expression (figure 5.4). That regular expression can then be expressed in lex syntax (figure 6.3), and then sent to an external file with appropriate formatting and code (figure 6.4). Chapter 7 will outline the steps (via a "user's manual" and pseudocode) that a software program would take to generate these lex patterns.

Chapter 7: Software to generate lex patterns

Consider the following DFA:

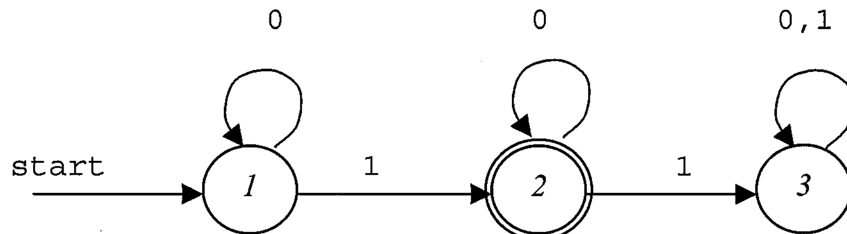

figure 7.1

The tabular representation of this DFA is:

	0	1
1	1	2
*2	2	3
3	3	3

figure 7.2

The r^i_{jk} tabular representation of this DFA is:

	k = 0	k = 1	k = 2	k = 3
r^k_{11}	$\varepsilon + 0$	0^*	0^*	0^*
r^k_{12}	1	0^*1	0^*10^*	0^*10^*
r^k_{13}	$\varnothing$	$\varnothing$	0^*10^*1	$0^*10^*1(\varepsilon + 0 + 1)* + 0^*10^*1$
r^k_{21}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
r^k_{22}	$\varepsilon + 0$	0^*	0^*	0^*
r^k_{23}	1	1	0^*1	0^*1
r^k_{31}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
r^k_{32}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
r^k_{33}	$(\varepsilon + 0 + 1)$	$(\varepsilon + 0 + 1)$	$(\varepsilon + 0 + 1)$	$(\varepsilon + 0 + 1)*$

figure 7.3

And finally as it's representation as a regular expression:

$$0^*10^*$$

figure 7.4

The regular expression can be thought of as: any string made up of zero or more 0's, a 1, and then zero or more 0's. The table in figure 7.3 can be used to create such an expression, since it is the union of the elements in the r^k_{12} row ($r^0_{12} \cup r^1_{12} \cup r^2_{12} \cup r^3_{12}$). This is equal to the set (1 + 0*1 + 0*10* + 0*10), which simplifies to 0*10*. It would take a bit of artificial intelligence on the part of a computer program to recognize this. For now, we can concern ourselves with reducing a regular expression with the following properties (based on properties in Chapter 5):

Reduction Rules

1. The union of a regular expression R and the null set is that regular expression R (from property 5.2).
2. The union of a regular expression R and that same regular expression is that regular expression R (from property 5.3).
3. The concatenation of a regular expression R and the null set is the null set (from property 5.7).
4. The concatenation of a regular expression R and epsilon is that regular expression R (from property 5.8).
5. The union of a Kleene star on a regular expression R* and an epsilon is that same Kleene star on a regular expression R* (from property 5.5).
6. The concatenation of a Kleene star with the same Kleene star is the same as the original Kleene star (from property 5.11).

Finally, a lemma that we can use to consolidate (consume) extraneous unions in a language:

> *Given language L with symbols $\Sigma^n = (\sigma_1, \sigma_2, \sigma_3, \ldots \sigma_n)$, then $L' = (\sigma_1 + \sigma_2 + \sigma_3 + \ldots + \sigma_n)^*$ defines an infinite language over L', and no additional regular expressions (over Σ^n) are required to describe L'.*

The proof of this lemma is by contradiction. Assume that an additional string s_1 is needed to further define L', given that L' = Σ^* over some alphabet Σ. Then L' = $\Sigma^* + s_1$ where s_1 is a string made up of the elements of Σ. Since s_1 is made up of any combination of elements in Σ, then $s_1 \in \Sigma^*$. So s_1 is not needed to further define L', which is a contradiction of our original assumption that we needed s_1 to define L'. Thus, if L' = Σ^* over some given alphabet Σ, than no additional strings are needed to further define L'

Therefore, if your alphabet is (0,1,2), and your language is defined by L = (0 + 1 + 2)*, then any additional regular expressions over (0,1,2) are not needed to describe the language. As an example, given alphabet (0,1), then the following regular expression:

$$(0 + 1)^* + 011 + (1 + 0)0$$

Can be simplified to:

$$(0 + 1)^*$$

Referring to figure 7.3, the steps needed to write computer code for this would be:

1. Request the number of states n from the user. This will create $n \times n$ numbers of rows in the r^i_{jk} table
2. Set up the default symbols for epsilon and null
3. Set up the k=0 column
4. Determine the accepting states m, and create the union of all r^i_{1m}
5. Process recursively for k = 1 to n, reducing along the way

Future modifications could be to ask for the symbol that make up language L, and sort them. This could assist in simplifying, since the program will know that (2 + 0 + 1) is same as (0 + 1 + 2)

The Java code listed in Appendix A runs against the DFA in figure 7.1 using the k=0 column of figure 7.3 as input. The classes Union and Concat create basic simplifications based on the above Reduction Rules. Although the output string can be further simplified, it is beyond the scope of the code in its present stage to do so. Future enhancements could include a GUI interface, and a user interface to request that a lex file be written (with or without appropriate action code).

Chapter 8: Future Refinements

The algorithm from Chapter 5, the code steps in Chapter 7, and the source code in Appendix A could be refined and simplified further. Future enhancements could include:

1. Sorting the alphas

   This is beneficial, so that two expressions such as (0 + 1 + 2) and (0 + 2 + 1) can be considered equivalent

2. Eliminating duplicity of expressions.

   For example, Concat(Concat(0, 1), 2) and

   Concat(0, Concat(1,2)) can be considered duplicate, although they yield the same result. However, the equals() methods would not consider them equivalent.

3. Creating the ability to accept an NFA in addition to a DFA

   This code was based on every alpha having a distinct transition from every state to another state (precisely the definition of a DFA). NFAs could have multiple transitions involving identical alphas, in addition to states not being reached by every distinct alpha.

Since the concatenation, union, and Kleene star operations are forms of expressions, the objects in the Java code were based on the following hierarchy:

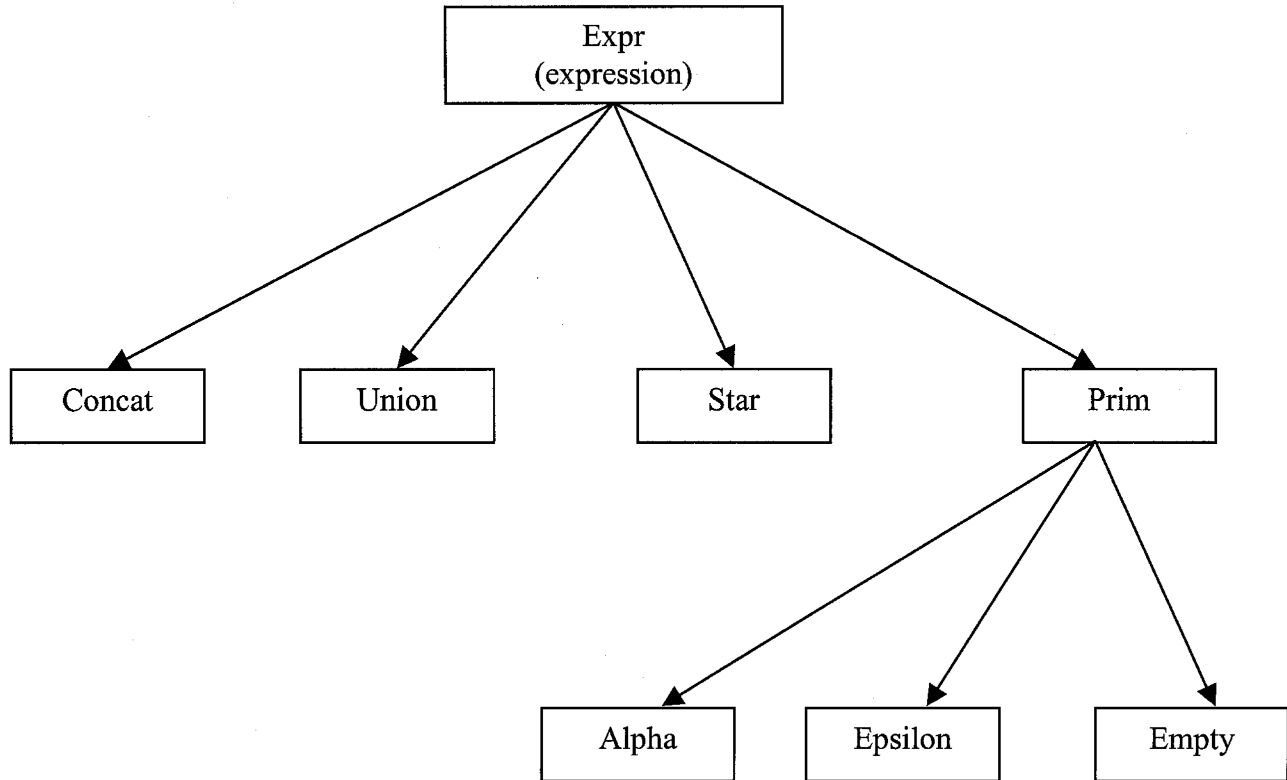

figure 8.1

The simplification (reduction) code (found in each class file in Appendix A) works for simple instance of reductions. For example, the code:

```
Alpha a = new Alpha(new Character('1'));
Alpha b = new Alpha(new Character('1'));
u = new Union(a + b);
u = u.reduce();
```

will reduce the variable u to be the Union of 1 + 1 to 1 (as per the rules of regular expressions)

However, a larger expression such as:

$$(1.(1.\varnothing))$$

where the period (.) is the concatenation operator yields

$$11.\varnothing$$

The code looks like:

```
Empty n = new Empty();

Alpha i = new Alpha(new Character('1'));

Expr x = new Concat(i, n);

System.out.println("x : Concat(i,n) : " + x);

Expr x1 = new Concat(i, x);

System.out.println("x1 : Concat(i,x) : " + x.reduce());
```

The output is:

```
x : Concat(i,n) : 1N

x1 : Concat(i,x) : 11N
```

Correct output is achieved only if reductions are performed at every step along the way. This may or may not be desirable, depending on if the user needs to see the output string as it is being developed.

Appendix A contains the source code used, and the URL where future enhancements can be made.

Appendix A: Java source code

The source code used to generate the strings of regular expressions is found on this page. As of the date of this document, the source code can also be found at www.rowan.edu/~safko/dfa/dfa.html. There is no error exception or handling written into the code; we assume for now that the end user can intuitively understand and use the code.

In keeping with the spirit of open source code, any and all refinements and enhancements are encouraged.

```
// Alpha.java

/** A regular expression for a singleton member of the alphabet in this Language.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Alpha extends Prim
{
	private Character alpha;
	/** Private variables for an expression.
	*  @param symbol The primitive symbol of the alphabet for this expression
	*/

	public Alpha (Character a)
	{
		super(a);
		alpha = a;
	}

	public Character getValue()
	{
		return alpha;
	}

	public String toString ()
	{
		return alpha.toString();
	}

	public boolean isEpsilon()
	{
		return false;
	}

	public boolean isNull()
	{
		return false;
	}

	public Expr reduce()
	{
		return this;
	}
}
```

```java
// Epsilon.java

/** A regular expression for epsilon.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Epsilon extends Prim
{
	private Character symbol = new Character('e');
	/** Private variables for an expression.
	*  @param symbol The primitive symbol of the alphabet for this expression
	*/

	public Epsilon ()
	{
		super(new Character('e'));
	}

	public Character getValue()
	{
		return symbol;
	}

	public String toString ()
	{
		return symbol.toString();
	}

	public boolean isEpsilon()
	{
		return true;
	}

	public boolean isNull()
	{
		return false;
	}

	public Expr reduce()
	{
		return this;
	}

}
```

```
// Empty.java

/** A regular expression for the empty set.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Empty extends Prim
{
        private Character symbol = new Character('N');
        /** Private variables for an expression.
        *  @param symbol The primitive symbol of the alphabet for this expression
        */

        public Empty ()
        {
                super(new Character('N'));
        }

        public Character getValue()
        {
                return symbol;
        }

        public String toString ()
        {
                return symbol.toString();
        }

        public boolean isEpsilon()
        {
                return false;
        }

        public boolean isNull()
        {
                return true;
        }

        public Expr reduce()
        {
                return this;
        }

}
```

```
// Prim.java

/** A primitive regular expression.
 * @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 * @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public abstract class Prim extends Expr
{
	private Character symbol;
	/** Private variables for an expression.
	* @param symbol The primitive symbol of the alphabet for this expression
	*/

	public Prim (Character a)
	{
		super(a);
	}

	public Character getValue()
	{
		return symbol;
	}

	public String toString ()
	{
		return symbol.toString();
	}


	public abstract boolean isEpsilon();

	public abstract boolean isNull();


}
```

```java
// Star.java

/** A regular expression representing the Kleene * of a
 *  regular expression.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Star extends Expr
{

  /** Constructor for a star.
   *  @param s The regular expression to be "starred"
   */
  public Star (Expr s)
  {
                super (s.reduce());
  }

 public boolean equals(Expr s)
 {
        if(s instanceof Star)
                if(this.getValue() == s.getValue())
                        return true;

        return false;
 }


 public Expr reduce()
 {
        if(this.getStar() instanceof Star)
                return this.getStar();

        return this; // in case the expression cannot be reduced further
 }

  /** Return the star in the form of a string which
   *  people can understand.
   */
  public String toString ()
  {
   // return the ToString, and also append the "*" symbol
                return  this.getStar().reduce().toString() + "*";
  }

}
```

```java
// Expr.java

/** A primitave regular expression. It could be as simple as a single
 *  letter, or an empty string, to as complex as an expression involving
 *  Unions, Concatenations, and Kleene Star
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */

public abstract class Expr
{

  /** Private variables for an expression.
   *  @param left The left operand of the concatenation.
   *  @param right The right operand of the concatenation.
   *  @param star The entire expression formatted for a * operation
   *  @param prim The character variable that holds primitive data
   *  @param epsChar The character variable that holds characters used to represent
   *  epsilon (the empty string)
   *  @param nullChar The character variable that holds characters used to represent the
   * empty set (the null set)
   */

  private Expr left, right, star;
  private Character prim;
  private Character epsChar = new Character('e');
  private Character nullChar = new Character('N');


  public Expr (Character p)
  {
              prim = p;
  }

/** Constructor for an Expr
   *  Return the expression in the form of a string which
   *  is ready for the star operation.
   */

  public Expr (Expr s)
  {
              star = s;
  }
```

```
// Expr.java - continued
/** Constructor for an Expr with left and right operands
 * Stores the expression in a left Expr and right Expr format
 */

public Expr (Expr l, Expr r)
{
        left = l;
        right = r;
        left = left.reduce();
        right = right.reduce();
}
    // Methods used to return the left, right, and star parameters
    public Expr getLeft()
    {
        return left;
    }

    public Expr getRight()
    {
        return right;
    }

    public Expr getStar()
    {
        return star;
    }

    public Character getValue()
    {
        return prim;
    }

 public boolean equals(Expr e)
 {
 if (this instanceof Union)
     if (e instanceof Union)
         return (this.getLeft().equals(e.getLeft())
                 && this.getRight().equals(e.getRight()))
                 ||
                 (this.getLeft().equals(e.getRight())
                 && this.getRight().equals(e.getLeft()));
     else
         return false;

 if (this instanceof Star)
```

```
		if (e instanceof Star)
			return (this.getStar().equals(e.getStar()));
		else
			return false;

	if (this instanceof Alpha)
		if (e instanceof Alpha)
			return (this.getValue().equals(e.getValue()));
		else
			return false;

		// If the code reaches here, assume for now that
		// a comparison cannot be made
		return false;
	}

		public boolean isEpsilon()
		{
		if (epsChar == prim)
			return true;
		else
			return false;
		}

		public boolean isNull()
		{
		if (nullChar == prim)
			return true;
		else
			return false;
		}

		public Expr reduce()
		{
			return this;
		}

		void setRight( Expr r)
		{
			right = r;
		}
}
```

```
// Concat.java
/** A regular expression consisting of the concatenation of two
 * other regular expressions.
 * @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 * @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a> */
public class Concat extends Expr
{
  /** Constructor for a concatenation.
   * @param l The left operand of the concatenation.
   * @param r The right operand of the concatenation.     */
  public Concat (Expr l, Expr r)
  {
              super (l.reduce(),r.reduce());
  }

 /** Return the concatenation reduce based on the rules of regular expressions   */
  public Expr reduce()
  {
        // if a star is concatenated with a similar star, return the star
        if(this.getLeft() instanceof Star)
              if (this.getRight() instanceof Star)
                    if (this.getRight().reduce().equals(this.getLeft().reduce()))
                          return this.getLeft().reduce();
        // If an expression is concatenated with a Null, return Null
              if(this.getLeft() instanceof Empty) return this.getLeft().reduce();
              if(this.getRight() instanceof Empty) return this.getRight().reduce();
        // If an expression is concatenated with an epsilon, return the expression
              if(this.getLeft() instanceof Epsilon) return this.getRight().reduce();
              if(this.getRight() instanceof Epsilon) return this.getLeft().reduce();

              if(this.getLeft() instanceof Star)
                    if(this.getRight() instanceof Star)
                          if(this.getLeft().equals(this.getRight()))
                                return this.getLeft().reduce();

        return this;
  }

  /** Return the concatenation in the form of a string which people can understand.  */
  public String toString ()
  // Insert the parentheses, and the concat operator "."
  {
        return "(" + this.getLeft().reduce().toString() + "." +
        this.getRight().reduce().toString() + ")";
  }
}
```

```
// Union.java
/** A regular expression consisting of the union of two
 *  other regular expressions.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>  */
public class Union extends Expr
{
  /** Constructor for a union.
   *  @param l The left operand of the union.
   *  @param r The right operand of the union.    */
   public Union (Expr l, Expr r)
   {
               super (l.reduce(),r.reduce());
   }
  /** Return the union reduce based on the rules of  regular expressions   */
        public Expr reduce()
        {
        if(this.getLeft() instanceof Empty) return this.getRight().reduce();
        if(this.getRight() instanceof Empty) return this.getLeft().reduce();
        if(this.getLeft().equals(this.getRight())) return this.getLeft().reduce();
        if(this.getLeft() instanceof Star)
                if(this.getRight() instanceof Star)
                        if(this.getLeft().getStar().equals(this.getLeft().getStar()))
                                return this.getLeft().getStar().reduce();

        // if an epsilon is reduced with a star, return the star
        if(this.getLeft() instanceof Epsilon)
                if(this.getRight() instanceof Star)
                        return this.getRight().reduce();
        if(this.getRight() instanceof Epsilon)
                if(this.getLeft() instanceof Star)
                        return this.getLeft().reduce();

        // if an alpha primitaive is unioned with an identical
        // alpha primitive, return the primitive
        if(this.getLeft() instanceof Alpha)
                if(this.getRight() instanceof Alpha)
                        if(this.getRight().getValue() == this.getLeft().getValue())
                                return this.getLeft().reduce();
        // otherwise, it couldn't be reduced
        return this;
        }
```

```
// Union.java - continued
  /** Return the union in the form of a string which people can understand.
  */
   public String toString ()
   {
  // Insert the parentheses, and the union operator "+" and return as a string.  Also convert
  //  the left and right operands to strings.  Every kind of Expr will have its own toString()
  // method.
        return "(" + this.getLeft().reduce().toString() + "+" +
        this.getRight().reduce().toString() + ")";
   }
}
```

```
// Figure7_1.java
// Based on the regular expression: 0*10*
//
import Expr;
import java.io.*;
import javax.swing.JOptionPane;

public class Figure7_1
{
public static void main(String [ ] args)
	{
	String reString = new String();
	String actionString = new String();
	String filename = new String();
	reString = r(1,2,0).reduce().toString() + " + " + r(1,2,1).reduce().toString() + " + "
+ r(1,2,2).reduce().toString();
	filename = JOptionPane.showInputDialog("Enter a file name (for lex input)");
	actionString = JOptionPane.showInputDialog("Enter an action for this NFA");
	System.out.println(reString);

	PrintWriter lexfile = null;
	try
	{
		lexfile = new PrintWriter(new FileOutputStream(filename));
	}
	catch(FileNotFoundException ex)
	{
		System.out.println("Error opening the output file");
		System.exit(0);
	}
	String eol = null;
	lexfile.println("%%");
	lexfile.println(reString + " " + actionString);
	lexfile.println("%%");
	lexfile.close();
	System.exit(0);
	}
```

```
// Figure7_1.java - continued
public  static Expr r(int i, int j, int k)
{
        if (k == 0)
                if(i == 1 && j == 1)
                        return new Star(new Alpha(new Character('0')));
        if (k == 0)
                if(i == 1 && j == 2)
                        return new Alpha(new Character('1'));
        if (k == 0)
                if(i == 1 && j == 3)
                 return new Empty();
        if (k == 0)
                if(i == 2 && j == 1)
                 return new Empty();

        if (k == 0)
                if(i == 2 && j == 2)
                        return new Star(new Alpha(new Character ('0')));

        if (k == 0)
                if(i == 2 && j == 3)
                 return new Empty();

        if (k == 0)
                if(i == 3 && j == 1)
                 return new Empty();

        if (k == 0)
                if(i == 3 && j == 2)
                 return new Empty();

        if (k == 0)
                if(i == 3 && j == 3)
                        return new Union(new Epsilon(), new Union(new Alpha(new
                        Character('0')), new Alpha(new Character('1'))));
        if (k != 0)
                {
                return new Union(new Concat((new Concat(r(i,k,k-1).reduce(),
                new Star(r(k,k,k-1).reduce())).reduce()),
                r(k,j,k-1).reduce()).reduce().reduce(), r(i,j,k-1)).reduce();
                }
        // should never get here, but include it to keep the compiler happy
        return new Epsilon();
}
}
```

```
// NFA.java
// Implement a JFrame in which to draw a NFA

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import java.util.*;              // for iterator

/** Nondeterministic Finite Automaton, with graphic paint method
 *  implemented as a JFrame window.
 *  Permits user to build an NFA, with states and transitions, using menu.
 *
 *  @author <a href="mailto:bergmann@rowan.edu">Seth Bergmann and Greg Safko
 * </a>
 *
 */
public class NFA extends JFrame
        implements ActionListener, MouseListener
{

/** This NFA knows which state, if any, is selected
 */
public State selectedState = null;
/** User is in the process of adding a transition; a state has already
 *  been selected, and the user will then click on the target state.
 */
public boolean addingTrans;            // currently adding a transition

State fromState;                       // remember origin of a transition

char transChar;                        // a transition's input symbol
TransPanel transPanel;                 // Panel to get a transition

/** The starting state of this NFA
 */
public State start;

/** An array of the states contained in this NFA.
 */
public State [ ] states;

/** The number of states contained in this NFA.
 */
public int nStates = 0;
private static int maxStates = 100;
```

```
// NFA.java - continued
/** Create an NFA, titled "NFA", with null layout manager
 */
public NFA ()
{  super ("NFA");
   setSize (600,400);
   getContentPane().setLayout (null);                  // turn off layout mgr.
   setDefaultCloseOperation (EXIT_ON_CLOSE);
   setUpMenu();
   transPanel = new TransPanel (this);
   transPanel.setVisible (false);
   states = new State [maxStates];
   WindowUtilities.openInJFrame (transPanel, 200, 175);
}
String [ ] buildActions =                  // actions for the build menu
   {
        "Add State ...",
        "Delete State ...",
        "Modify State ...",
        "Add Transition ...",
        "Delete Transition ...",
        "Show Regular Expression..."
   };
/** Create the menus and add them to this NFA
 *  Open, Save not yet implemented.
 *  Delete State, Delete Transition not yet implemented
 */
public void setUpMenu()
{       JMenuItem menuItem;
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu ("File");
        JMenu buildMenu = new JMenu ("Build NFA");
        fileMenu.add ("Open ...");
        fileMenu.add ("Save");
        fileMenu.add ("Save As ...");
        for (int i=0; i<buildActions.length; i++)
        {       menuItem = new JMenuItem (buildActions[i]);
                buildMenu.add (menuItem);
                menuItem.addActionListener (this);
        }
        menuBar.add (fileMenu);
        menuBar.add (buildMenu);
        setJMenuBar (menuBar);
        fileMenu.addActionListener (this);
        buildMenu.addActionListener (this);
}
```

```
// NFA.java - continued
/** Implement the menu actions
 * Open, Save not yet implemented.
 * Deletion not yet implemented.
 * @param evt The menu event which invoked this method
 */
public void actionPerformed (ActionEvent evt)
{ if (evt.getActionCommand() == "Open ...") openFile();
  if (evt.getActionCommand() == "Save")     saveFile();
  if (evt.getActionCommand() == "Save As ...")     saveFileAs();
  for (int i=0; i<buildActions.length; i++)
        if (evt.getActionCommand().equals (buildActions[i]))
          switch (i)
           {
           case 0: addState();  break;
                case 1: delState();            break;
                case 2: modifyState();         break;
                case 3: addTrans();            break;
                case 4: delTrans();            break;
                case 5: showRE();              break;
           }
}
/** Add a state to this NFA
 */
void addState()
{
   if(nStates < maxStates)
   {
                State s;
                s = new State(100,100,this);
                getContentPane().add (s);
                states [nStates] = s;
                s.index = nStates;
                nStates++;
                repaint();
        }
        else
                System.out.println ("Too many states.  Max is "+ maxStates);
}
// Allow the user to set the attributes of the selected State.
// Eventually, grey out modify if no state is selected.
void modifyState()
{
        if (selectedState!=null) selectedState.setAttributes();
        repaint();
}
```

```
// NFA.java - continued
/** Add a transition to a state of this NFA.  A state should be selected first, by
 * clicking on it with the mouse.
 */
void addTrans()
{
	transPanel.setVisible (true);
	fromState = selectedState;
	addingTrans = true;
}

/** Delete a transition from this NFA
 */
void delTrans()
{
}

/** Show the regular expression corresponding to this NFA on stdout
 */
public void showRE()
{
  Union u = new Union (new Empty(), new Empty());
  Union newu = u;
  Iterator it = start.transitions.iterator();
  Transition trans;
  while (it.hasNext())
  {
	trans = (Transition) it.next();
	if(trans.toState.isFinal)
	{
	newu.setRight(new Union( r(start.index,trans.toState.index, nStates - 1),
		new Empty()));
	newu = (Union) u.getRight();
	}
  }
  System.out.println ("result unreduced is : " + u);
  System.out.println ("result reduce is    : " + u.reduce());
  // Show the Regular Expression
  String reString = new String();
  String actionString = new String();
  String filename = new String();
  filename = JOptionPane.showInputDialog("Enter a file name (for lex input)");
  actionString = JOptionPane.showInputDialog("Enter an action for this NFA");
  reString = u.toString();
  System.out.println(reString);
  PrintWriter lexfile = null;
```

```
// NFA.java - continued
  try
        {
              lexfile = new PrintWriter(new FileOutputStream(filename));
        }
  catch(FileNotFoundException ex)
        {
              System.out.println("Error opening the output file");
              System.exit(0);
        }
        lexfile.println("%%");
        lexfile.println(reString + " " + actionString);
        lexfile.println("%%");
        lexfile.close();
        System.exit(0);

}

/** The recursive algorithm from Hopcroft & Ullman
 * @param i Origin state
 * @param j Destination state
 * @param k Limiting state
 * @return A regular expression for all strings which will take
 * the machine from state i to state j, without going through
 * a state higher than state k.  State numbers begin with 0.
 */
public Expr r (int i, int j, int k)
  {
        if (k==-1)                    // base case
          {
              Iterator it = states[i].transitions.iterator();
              Union u = new Union(new Empty(), new Empty());
          Transition trans;
          Union newu = u;
          while (it.hasNext())
          {
              trans = (Transition) it.next();
                    if (trans.toState.index == j)
                    {
              newu.setRight(new Union(new Alpha(new Character (trans.inp)),
                    new Empty()));
                          newu = (Union) newu.getRight();
                    }
              }
              if (i==j)
                    newu.setRight(new Epsilon());
```

```
// NFA.java - continued
                return u.reduce();
                }       // end k==-1


        return new Union(new Concat((new Concat(r(i,k,k-1).reduce(),
        new Star(r(k,k,k-1).reduce())).reduce()),
        r(k,j,k-1).reduce()).reduce().reduce(), r(i,j,k-1)).reduce();

  }

/** Delete a State from this  NFA.  Not yet implemented.
 */
void delState()
{
}

/** Open a new NFA, not yet implemented.
 */
void openFile()
{
}

/** Save this NFA, not yet implemented.
 */
void saveFile()
{
}

/** Save this NFA, with a new name, not yet implemented.
 */
void saveFileAs()
{
}

/** Handle the Mouse clicked event if on a state
 */
public void mouseClicked (MouseEvent evt)
{
}

/** Handle the Mouse entered event if on a state
 */
public void mouseEntered (MouseEvent evt)
{
}
```

```
// NFA.java - continued
/** Handle the Mouse exited event if on a state  */
public void mouseExited (MouseEvent evt)
{
}
/** Handle the Mouse pressed event if on a state  */
public void mousePressed (MouseEvent evt)
{
}
/** Handle the Mouse released event if on a state  */
public void mouseReleased (MouseEvent evt)
{
}

/** Display all the components of this NFA on Graphics g
 *  For each state, check to see if it is a start and/or final state.
 *  Also display all the transitions from the state.  */
public void paint (Graphics g)
{  super.paint (g);
   Graphics2D graphics2D = (Graphics2D) g;

   JComponent comp;
   Color oldColor;
   Arrow startArrow;          // for start state
   State state;
   int i;
   for (i=0; i<getContentPane().getComponentCount(); i++)
     {   comp = (JComponent) getContentPane().getComponent(i);
         oldColor = graphics2D.getColor();
         if (comp instanceof JLabel)
                graphics2D.setColor (Color.magenta);          // magenta?
         if (comp instanceof State)                           // kludge for start states
           { state = (State) comp;
            if (state.isStart)
             {     startArrow = new Arrow (state.locX, state.locY,
                             state.locX+state.r/1.414, state.locY+state.r/1.414,
                             Color.blue);
                   startArrow.paint (graphics2D);
             }
           state.moveLabels();  // adjust the positions of the labels on the transitions
        }
         comp.paint(graphics2D);            // paint each component
         graphics2D.setColor (oldColor);
     }
}
}
```

```
// State.java
// class for an individual state of an NFA

import java.awt.*;            // for graphics
import java.awt.geom.*;       // for Arc2D
import javax.swing.*;         // for graphics
import java.util.*;           // for HashSet
import java.awt.event.*;      // for Mouse events

/** State of an NFA, implemented as a JComponent.
 *  May be added to an NFA.
 *  May possess 0 or more transitions to other states.
 *  May be final or start state.
 *  Implements MouseListener so that States can be selected and dragged
 */
public class State extends JComponent
                   implements MouseListener
{
/** The NFA (i. e. JFrame) which contains this state
 */
NFA nfa;                              // nfa containing this state
/** Radius of all states, when drawn as a circle
*/
public static int r = 20;                        // radius when drawn as a circle
/** final state is an accepting state,
 *  Accessed in the Attributes class.
 */
public boolean isFinal, isStart;

/** The index number of this state (used for the array states in NFA)
 */
public int index;

/** This state has a set of transitions to other states
*/
public HashSet transitions;
/** A state may be labelled.  Not yet implemented.
 */
public char label;

/** location of this state on the frame
 */
public int locX, locY;          // location on the frame, upper left corner
```

```java
// State.java - continued
/** Create a state
 *  @param x x coordinate of this state on the frame
 *  @param y y coordinate of this state on the frame
 *  @param n The NFA which contains this state
 */
public State (int x, int y, NFA n)
{  locX = x;
   locY = y;
   setBounds (locX, locY-3*r, 2*r, 2*r);              // kludge on y coordinate
   nfa = n;
   transitions = new HashSet();
   isStart = false;
   isFinal = false;
   addMouseListener (this);
}

/** Allow the user to set the attributes of this State, using a panel with
 *  checkboxes.
 */
public void setAttributes()
{
   Attributes att = new Attributes(this);
   WindowUtilities.openInJFrame (att,200,175);
}

/** Save mouse click location, for dragging of states
 */
int downAtX, downAtY;

/** A selected state can be modified or given a transition
 */
public boolean isSelected;

/** Respond to a MouseClicked event, by selecting this state.  Also check to
 *  see if it was clicked to select a transition target.
 *  @param evt The location of the event.
 */
```

```
// State.java - continued
public void mouseClicked (MouseEvent evt)
{
   Transition trans;
   selectThis();                    // this state is selected
   if (nfa.addingTrans)                          // adding a transition?
     {
        trans = new Transition (this, nfa.fromState, nfa.transChar);
        nfa.fromState.transitions.add (trans);
        nfa.addingTrans = false;
        nfa.repaint();
     }
}

/** Respond to a MouseEntered event
 *  @param evt The location of the event.
 */
public void mouseEntered (MouseEvent evt)
{
}

/** Respond to a MouseExited event
 *  @param evt The location of the event.
 */
public void mouseExited (MouseEvent evt)
{
}

/** Respond to a MousePressed event
 *  @param evt The location of the event.
 */
public void mousePressed (MouseEvent evt)
{  downAtX = evt.getX();              // save location of click
   downAtY = evt.getY();
   selectThis();                    // select this state
}

/** Respond to a MouseReleased event.
 *  This State has been dragged.
 *  @param evt The location of the event.
 */
```

```
// State.java - continued
public void mouseReleased (MouseEvent evt)
{  int x,y;
   int xDiff, yDiff;
   x = evt.getX();
   y = evt.getY();
   xDiff = x - downAtX;
   yDiff = y - downAtY;
   locX = locX + xDiff;                // new location of this State
   locY = locY + yDiff;
   setLocation (locX, locY-3*r);       // kludge on y coordinate
   moveLabels ();                // move the labels also
   nfa.repaint();
}

void moveLabels ()
{  Iterator it = transitions.iterator();
   Transition trans;
   while (it.hasNext())
     {    trans = (Transition) it.next();
          trans.lbl.setLocation ();
          trans.lbl.setOpaque (false);
     }
}

//  Deselect the selected state and select this state instead
void selectThis()
{  if (nfa.selectedState!=null) nfa.selectedState.isSelected = false;
   isSelected = true;
   nfa.selectedState = this;
}

/** Display this state on the given graphics object g.  Color it red if
 *  selected.  Draw an inner circle if it is a final state.  Draw an arrow
 *  coming in if it is a start state.  Draw all the transitions coming out
 *  of this state.
 * @param g The Graphics object on which it is being displayed.
 */
```

```
// State.java - continued
public void paint (Graphics g)
{ super.paint (g);
  Graphics2D graphics2D = (Graphics2D) g;

   if (isSelected) graphics2D.setColor (Color.red);
   graphics2D.draw (new Arc2D.Double (locX, locY, 2.0*r, 2.0*r, 0.0, 360.0,
                    Arc2D.OPEN));
   // draw inner circle for final states
   if (isFinal)
         graphics2D.draw (new Arc2D.Double (locX+0.2*r, locY+0.2*r,
                    1.6*r, 1.6*r, 0.0, 360.0, Arc2D.OPEN));
   // draw arrow to start state(s)
   Arrow startArrow;
   double diff;
   if (isStart)
     {   diff = r / 1.414;
         startArrow = new Arrow (locX, locY, locX+diff, locY+diff, Color.blue);
     }

   graphics2D.setColor (Color.black);


  // draw the transitions emanating from this State
  Iterator it = transitions.iterator();
  Transition trans;
  while (it.hasNext())
    {    trans = (Transition) it.next();
         trans.paint(g);
    }


}
}
```

```
// Transition.java
// Transition class to store a transition
//   store the state to which it goes, and the input symbol.

import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

/** Transition which originates from a particular
  * <a href="State.html">State</a>.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Transition extends JComponent
{

/** The states to and from  which this transition is directed.
 */
   public State toState, fromState;

/** The label (input symbol) on this transition
 */
  public TransLabel lbl;

/** The input symbol of this transition.
 */
   public char inp;                      // Input symbol
   private static int lblWidth=18;
   private static int lblHeight=20;

/** Create a transition
 * @param to The State to which this transition is directed.
 * @param i The input symbol for this transition.
 */
public Transition (State to, State from, char i)
{        toState = to;
         fromState = from;
         inp = i;

  String s = (new Character (inp)).toString();
  lbl = new TransLabel (s,this);
}
```

```
// Transition.java - continued
/** For debugging purposes
 */
public String toString()
{  return "Transition, toState=" + toState + ", inp=" + inp;
}

/** Paint this Transition as an Arrow with a label.  If the transition is to
 *  itself, draw it as an arc.
 */
public void paint (Graphics g)
{  super.paint (g);
   Graphics2D graphics2D = (Graphics2D) g;

   Arrow arrow;                    // arrow used to draw this transition
   double arcArrowX, arcArrowY;
   if (toState == fromState)              // arc arrow
     {    arcArrowX = toState.locX+0.5*State.r;
          arcArrowY = toState.locY-0.5*State.r;
          graphics2D.draw (new Arc2D.Double (arcArrowX, arcArrowY,
                  (double) State.r, (double) State.r,
                  -0.01*State.r, 180+0.01*State.r,Arc2D.OPEN));
          lbl.setLocation (toState.locX+2*State.r, toState.locY-4*State.r);
     }
   else
     {    arrow = new Arrow (fromState.locX+State.r, fromState.locY+State.r,
                          toState.locX+State.r, toState.locY+State.r);
          arrow.paint(g);
     }
}


}
```

```
// Arrow.java
// Arrow class to draw straight line segments with arrow heads.

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
// import java.awt.geom.Point2D;

/** Arrow class with graphic paint method, implemented as a JComponent
 *
 *  @author <a href="mailto:bergmann@rowan.edu">Seth Bergmann and Greg Safko
 * </a>
 *
 */
public class Arrow extends JComponent
{

/** x1,y1 is first endpoint of this arrow.
 *  x2,y2 is second endpoint of this arrow.
 *  color is the color of this arrow, default is black.
 */
public double x1,y1,x2,y2;            // endpoints of this Arrow
Color color;                          // color of this Arrow

/** arrowhead's angle with respect to the shaft
 */
public static final double angle = Math.PI/10; // arrowhead angle to shaft
/** length of arrowhead
 */
public static final double len = 10;          // arrowhead length

/** Create an arrow with given endpoints
 *  @param xOne x coordinate of first endpoint
 *  @param yOne y coordinate of first endpoint
 *  @param xTwo x coordinate of second endpoint
 *  @param yTwo y coordinate of second endpoint
 */
public Arrow (double xOne, double yOne, double xTwo, double yTwo)
{  x1 = xOne;
   x2 = xTwo;
   y1 = yOne;
   y2 = yTwo;
}
```

```
// Arrow.java - continued
/** Create an arrow with given endpoints and given color
 *  @param xOne x coordinate of first endpoint
 *  @param yOne y coordinate of first endpoint
 *  @param xTwo x coordinate of second endpoint
 *  @param yTwo y coordinate of second endpoint
 *  @param c color of this arrow
 */
public Arrow (double xOne, double yOne, double xTwo, double yTwo, Color c)
{  x1 = xOne;
   x2 = xTwo;
   y1 = yOne;
   y2 = yTwo;
   color = c;
}

/** Private fields needed for arrowhead endpoints
 *
 */
double ax1,ay1, ax2, ay2;                    // coordinates of arrowhead endpoints

/** Display this arrow on the given Graphics object g.
 *
 */
public void paint (Graphics g)
{ super.paint (g);
  Graphics2D graphics2D = (Graphics2D) g;
  Color oldColor;
  oldColor = graphics2D.getColor();
  graphics2D.setColor (color);

  // paint the shaft
  graphics2D.draw (new Line2D.Double (x1,y1,x2,y2));

  // paint arrowhead
        arrHead (x1,y1,x2,y2);
        graphics2D.draw (new Line2D.Double (x2,y2,ax1,ay1));
        graphics2D.draw (new Line2D.Double (x2,y2,ax2,ay2));
        graphics2D.setColor (oldColor);
}
/**  Compute the arrowhead endpoints
 *  arrow is pointing from x1,y1 to x2,y2
 *  angle to shaft is angle.
 *  length of arrowhead is len.
 *  return endpoints in ax1, ay1, ax2, ay2.
 */
```

```
// Arrow.java - continued
private void arrHead (double x1, double y1, double x2, double y2)
{  double c,a,beta,theta,phi;
  c = Math.sqrt ((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
  if (Math.abs(x2-x1) < 1e-6)
        if (y2<y1) theta = Math.PI/2;
        else theta = - Math.PI/2;
  else
   { if (x2>x1)
          theta = Math.atan ((y1-y2)/(x2-x1)) ;
        else
          theta = Math.atan ((y1-y2)/(x1-x2));
   }
  a = Math.sqrt (len*len  + c*c - 2*len*c*Math.cos(angle));
  beta = Math.asin (len*Math.sin(angle)/a);
  phi = theta - beta;
  ay1 = y1 - a * Math.sin(phi);              // coordinates of arrowhead endpoint
  if (x2>x1)
        ax1 = x1 + a * Math.cos(phi);
  else
        ax1 = x1 - a * Math.cos(phi);
  phi = theta + beta;                        // second arrowhead endpoint
  ay2 = y1 - a * Math.sin(phi);
  if (x2>x1)
        ax2 = x1 + a * Math.cos(phi);
  else
        ax2 = x1 - a * Math.cos(phi);
}

}
```

```
// TransLabel.java

import java.awt.*;
import javax.swing.*;

/** A TransLabel labels a transition with a symbol from the NFA's input alphabet.
  * <a href="State.html">State</a>.
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class TransLabel extends JLabel
{

/** The Transition which this label is attached to.
 */
   public Transition trans;
   State toState, fromState;

   static int lblWidth=18;
   static int lblHeight=18;

/** Create a transition
 * @param t  The transition being labelled.
 * @param s The input symbol for this transition.
 */
public TransLabel (String s, Transition t)
{       super (s);
        trans = t;
        toState = trans.toState;
        fromState = trans.fromState;
        setHorizontalAlignment (JLabel.LEFT);
        setVerticalAlignment (JLabel.BOTTOM);
        setBounds (lblX(), lblY(),  lblWidth, lblHeight);
        setOpaque (false);                      // transparent
        trans.toState.nfa.getContentPane().add (this);
}

/**  Establish the x,y coordinates of the upperleft corner of this label.
 */
public void setLocation ()
{  super.setLocation (lblX(), lblY());
}
```

```
// Translabel.java - continued
public int lblX ()
{ double theta;
 if (Math.abs (toState.locX-fromState.locX) < 1)     // Vertical transition
        return round (fromState.locX + State.r);
 else if (toState.locX > fromState.locX)                    // Transition to the right?
   {    theta = Math.atan ((fromState.locY-toState.locY)/(toState.locX-fromState.locX));
        return  round (fromState.locX + State.r + 3.5*State.r*Math.cos(theta));
   }
 else                                          // Transition to the left
   {    theta = Math.atan ((fromState.locY-toState.locY)/(fromState.locX-toState.locX));
        return round (fromState.locX - 1.5*State.r*Math.cos(theta));
   }
}

public int lblY ()
{  double theta;
  if (Math.abs (toState.locX-fromState.locX) < 1)    // Vertical transition
   if (toState.locY>fromState.locY)
        return round (fromState.locY + 0.8*State.r);          // downward
   else return round (fromState.locY - 2.3*lblHeight);        // upward
  else if (toState.locX > fromState.locX)             // Transition to the right
   {    theta = Math.atan ((fromState.locY-toState.locY)/(toState.locX-fromState.locX));
        return round (fromState.locY - lblHeight - 3.5*State.r*Math.sin(theta));
   }
  else                                 // Transition to the left
   {    theta = Math.atan ((fromState.locY-toState.locY)/(fromState.locX-toState.locX));
        return round (fromState.locY - lblHeight - 1.5*State.r*Math.sin(theta));
   }
}

// Round to nearest int.
static int round (double d)
{  return (new Double (d+0.5)).intValue();
}

}
```

```
// TransPanel.java

import javax.swing.*;
import java.awt.event.*;

/** A Panel to obtain the character on a transition being added to the nfa.

 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */

public class TransPanel extends JPanel implements ActionListener
{

JButton okButton;
JTextField txtField;
NFA nfa;

public TransPanel (NFA n)
{  okButton = new JButton ("OK");
   add (okButton);
   txtField = new JTextField ("Enter a character");
   add (txtField);
   okButton.addActionListener (this);
   nfa = n;
   setVisible (true);
}


/** OK button has been clicked.  Get the char and hide this panel.
 */
public void actionPerformed (ActionEvent evt)
{  nfa.transChar = txtField.getText().charAt(0);
   setVisible (false);
}


}
```

```
// Attributes.java
import javax.swing.*;
import java.awt.event.*;
/** Obtain attributes of a state from the user (start, final).
 * Starting state, and final state are attributes.
 * Use a JPanel containing two checkBoxes.
 * @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 * @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a> */
public class Attributes extends JPanel
        implements ItemListener, ActionListener
{ JCheckBox startBox, finalBox;
  State state;                    // the selected state

/** Set up the JPanel containing two checkboxes
 * @param st The state whose attributes are being modified. */
public Attributes(State st)
{
  startBox = new JCheckBox ("Start state");
  finalBox = new JCheckBox ("Final (accepting) state");
  startBox.setContentAreaFilled (false);
  finalBox.setContentAreaFilled (false);
  startBox.addActionListener(this);
  finalBox.addActionListener(this);
  startBox.setSelected (st.isStart);
  finalBox.setSelected (st.isFinal);
  add (startBox);
  add (finalBox);
  state = st;
  setVisible (true);
}

/** This method is invoked when a box is checked  */
public void actionPerformed (ActionEvent evt)
{
  state.isStart = startBox.isSelected();
  if (state.isStart) state.nfa.start = state;
  state.isFinal = finalBox.isSelected();
}

public void itemStateChanged (ItemEvent evt)
{
}

}
```

```
// Driver.java
// Driver for the 2D Graphics implementations of NFA class


/** Test the NFA class
 *
 *  @author <a href="mailto:bergmann@rowan.edu"> Seth Bergmann </a>
 *  @author <a href="mailto:gspyo@jersey.net"> Greg Safko </a>
 */
public class Driver
{
public static void main (String [ ] args)
  {
        NFA nfa = new NFA();                    // NFA is a JFrame
        nfa.setVisible (true);
  }
}
```

```
// ExitListener.java
import java.awt.*;
import java.awt.event.*;

/** A listener that you attach to the top-level JFrame of
 * your application, so that quitting the frame exits the
 * application.
 *
 * Taken from Core Web Programming from
 * Prentice Hall and Sun Microsystems Press,
 * http://www.corewebprogramming.com/.
 * &copy; 2001 Marty Hall and Larry Brown;
 * may be freely used or adapted.
 */
public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    // System.exit(0);
  }
}
```

```
// WindowsUtilities.java
import javax.swing.*;
import java.awt.*;  // For Color and Container classes.

/** A few utilities that simplify using windows in Swing.
 *
 *  Taken from Core Web Programming from
 *  Prentice Hall and Sun Microsystems Press,
 *  http://www.corewebprogramming.com/.
 *  &copy; 2001 Marty Hall and Larry Brown;
 *  may be freely used or adapted.
 */

public class WindowUtilities {

  /** Tell system to use native look and feel, as in previous
   *  releases. Metal (Java) LAF is the default otherwise.
   */

  public static void setNativeLookAndFeel() {
    try {
     UIManager.setLookAndFeel(
       UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Error setting native LAF: " + e);
    }
  }

  public static void setJavaLookAndFeel() {
    try {
     UIManager.setLookAndFeel(
       UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Error setting Java LAF: " + e);
    }
  }

   public static void setMotifLookAndFeel() {
    try {
      UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    } catch(Exception e) {
      System.out.println("Error setting Motif LAF: " + e);
    }
  }
```

```java
// WindowsUtilities.java – continued
 /** A simplified way to see a JPanel or other Container. Pops
  *  up a JFrame with specified Container as the content pane.
  */

 public static JFrame openInJFrame(Container content,
                                   int width,
                                   int height,
                                   String title,
                                   Color bgColor) {
   JFrame frame = new JFrame(title);
   frame.setBackground(bgColor);
   content.setBackground(bgColor);
   frame.setSize(width, height);
   frame.setContentPane(content);
   frame.addWindowListener(new ExitListener());
   frame.setVisible(true);
   return(frame);
 }

 /** Uses Color.white as the background color. */

 public static JFrame openInJFrame(Container content,
                                   int width,
                                   int height,
                                   String title) {
   return(openInJFrame(content, width, height,
                       title, Color.white));
 }

 /** Uses Color.white as the background color, and the
  *  name of the Container's class as the JFrame title.
  */

 public static JFrame openInJFrame(Container content,
                                   int width,
                                   int height) {
   return(openInJFrame(content, width, height,
                       content.getClass().getName(),
                       Color.white));
 }
}
```

References

Aho, A.V., R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, Mass: Addison-Wesley, 1986.

Bergmann, S.D., *Compiler Design: Theory, Tools, and Examples.* Glassboro, NJ: Rowan University, 1998. Unpublished manuscript.

Ibid., Wm. C. Brown Publishers, 1994

Estier, T., *What is BNF Notation*, http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html downloaded from the Internet on Dec 19, 2001

Hein, J.L., *Theory of Computation.* Boston: Jones and Bartlett, 1996.

Hopcroft, J.E, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Reading, Mass: Addison-Wesley, 1986.

Hutton, B., *Computer Science 07.330 Language Implementation, Lecture Notes.* Auckland, New Zealand: University of Auckland Press, 1987.

Levine, J.R., T. Mason, and D. Brown, *Lex & yacc.* Cambridge, Mass: O'Reilly, 1992.

Parsons, T., *Introduction to Compiler Construction.* New York: W.H. Freeman, 1992.

Sudkamp, T., *Languages and Machines, An Introduction to the Theory of Computer Science.* Reading, Mass: Addison-Wesley 1991.