From regular expressions to lexer generators

CS 321 Languages and Compiler Design I, Winter Term 2014
Department of Computer Science, Portland State University

These notes describe an implementation of abstract syntax trees for regular expressions as well as algorithms for parsing simple forms of regular expressions and for generating corresponding NFAs and DFAs from them. As such, the code that is described here could be seen as a (somewhat simple-minded) first step towards an implementation of a general purpose regular expression matching library, or a more specialized tool such as lexical analyzer generators like `lex`, `flex`, and `jflex`.

Primary learning objectives: Upon successful completion, students will be able to:

- Describe and apply mechanisms for defining the lexical structure of a programming language.
- Explain the role of abstract syntax trees, and syntax analysis techniques in a compiler for regular expressions and in the operation of a simple lexer generator.

1 Regular Expressions

We will work with the simple language of regular expressions that is described in the following table:

Name	Regular expression	Matches
`Epsilon` (empty)	`%`	Matches only the empty string. This is sometimes written as ϵ, but we have used `%` here instead because it is easier to enter on a standard keyboard.
`Char` (single character)	c	Matches a specific single character string; c stands for an arbitrary single character. For example, the regular expression `x` matches only the single character string whose first (and only) character is `x`.
`Seq` (sequence)	$r_1\,r_2$	Matches any string of the form $v\,w$, the concatenation of strings v and w, such that r_1 matches v and r_2 matches w.
`Alt` (alternatives)	$r_1 \mid r_2$	Matches all strings that match either r_1 or r_2 (or both).
`Rep` (repetition)	$r{\star}$	Matches all strings of the form $v_1\,v_2\,\ldots\,v_n$ for any $n \geq 0$ so long as r matches all of the strings v_1, v_2, …, and v_n. This is sometimes summarized by saying that $r{\star}$ matches the concatenation of *zero or more* strings, each of which matches r.

For example, the regular expression `x(y|z)*` will match any sequence of characters that begins with an `x` and is followed by a sequence of zero or more `y` or `z` characters. This includes strings like `x`, `xyyy`, `xzzz`, and `xyzyz`, but excludes the empty string as well as any string, like `abc`, that does not begin with `x`. There are also situations where it is useful to talk about *partial matching*. For example, the regular expression `x(y|z)*` does not match the full string `xyzabc`, but it does allow a partial match against the initial portion, `xyz`, leaving the remainder, `abc`, unmatched.

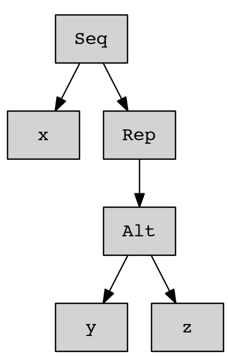
More formally, we can describe the set of all regular expressions, built from the five constructs shown above, using the following context-free grammar:

$$
\begin{array}{lcl}
r & \rightarrow & \texttt{\%} \\
r & \rightarrow & \texttt{c} \\
r & \rightarrow & r\,r \\
r & \rightarrow & r \;\texttt{|}\; r \\
r & \rightarrow & r\,\star \\
r & \rightarrow & \texttt{(}\; r \;\texttt{)}
\end{array}
$$

This grammar is *ambiguous*, meaning that it allows some input strings to be parsed in more than one way. And, unfortunately, in many of these cases, different parses will lead to different interpretations of the regular expression. For example, according to this grammar, the regular expression `xy|z` could be interpreted as either `x(y|z)` or `(xy)|z`. We will work around these problems in Section 3.2 by giving a refined grammar that enforces particular precedences (or order of operations). For the time being, however, we will avoid such issues by writing explicit sets of parentheses around individual subexpressions (made possible by the sixth production in the grammar above) wherever necessary to avoid ambiguity.

2 Abstract Syntax for Regular Expressions

Thinking in terms of abstract syntax, we can capture the underlying structure of any regular expression in the language described above by using a set of abstract syntax trees with five different node types—`Epsilon`, `Char`, `Seq`, `Alt`, and `Rep`—corresponding to the five different forms of regular expression described above. The following diagram, for example, shows the abstract syntax tree structure for the regular expression `x(y|z)*`:

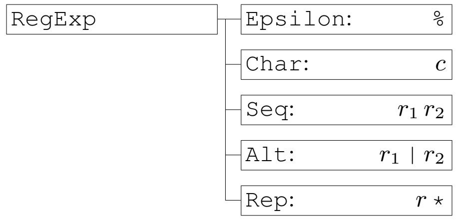


Interpretation in words: `x(y|z)*` describes a sequence (`Seq`) of a single character (`x`) and a repetition (`Rep`) of an alternative (`Alt`) between the two characters `y` and `z`.

(Note that we label each `Char` node with the specific character to be matched; all other nodes are labeled instead with their type with links to their subtree components where appropriate.)

2.1 Implementation in Java

To build an implementation of these ideas in Java, we can define a small collection of classes, using an abstract `RegExp` base class with five subclasses corresponding to each of the five possible regular expression forms described above. The resulting hierarchy can be illustrated by the following class diagram:



In this diagram, subclasses are drawn to the right of the base classes that they extend, and classes corresponding to specific forms of regular expressions are represented by boxes that contain the name of a class and a regular expression fragment suggesting the corresponding concrete syntax.

For example, the overall structure of the `Char` class is as follows, extending `RegExp` and adding a single attribute, `c`, that represents the particular character to be matched:

regexp/Char.java

```java
/** Represents a single character regular expression.
 */
class Char extends RegExp {
    private int c;
    Char(int c) {
        this.c = c;
    }
    ...
}
```

In a similar way, the `Seq` class extends `RegExp` and provides a representation for sequences, including components for two subexpressions, `r1` and `r2`:

regexp/Seq.java

```java
/** Represents a sequence regular expression of the form r1 r2, which
 *  matches a string matching r1 followed by a string matching r2.
 */
class Seq extends RegExp {
    private RegExp r1;
    private RegExp r2;
    Seq(RegExp r1, RegExp r2) {
        this.r1 = r1;
        this.r2 = r2;
    }
    ...
}
```

Each of the remaining abstract syntax classes—`Epsilon`, `Alt`, and `Rep`—are defined in a very similar way.

2.2 Printing Fully Parenthesized Regular Expressions

In the previous section, we described a small collection of classes that allow us to construct abstract syntax trees representing regular expressions. Once we have built up these syntax trees, we will, of course, also want to define functions that operate on these trees. As a simple example, we will define an operation `r.fullParens()` that takes an arbitrary `RegExp` abstract syntax tree, `r`, and returns a string value that contains a fully parenthesized version of that same regular expression. (By fully parenthesized, we mean that every use of a sequence, alternative, or repetition construct is enclosed in parentheses so that there can be no ambiguity or uncertainty about the order in which the different operators should be applied.)

Of course, the implementation of `fullParens()` will need to produce a different output string for each different type of abstract syntax tree `r`. To make this work, we start by defining an 'abstract' method in the `RegExp` base class that specifies the type of `fullParens()` (in this case, the fact that it has no parameters and returns a value of type `String`):

regexp/RegExp.java

```
/** Return a string with a fully parenthesized version of this
 *  regular expression.
 */
public abstract String fullParens();
```

After this, we must add an appropriate implementation of `fullParens()` in each of the subclasses of `RegExp`. For `Epsilon` and `Char`, this is particularly straightforward: we just need to return a single character, and there is no need for any parentheses:

regexp/Epsilon.java

```
public String fullParens() {
    return "%";
}
```

regexp/Char.java

```
public String fullParens() {
    return Character.toString((char)c);
}
```

For the remaining cases, we make recursive calls to `fullParens()` on each of the `RegExp` components of the tree. The resulting strings are then combined in an appropriate way, including a pair of enclosing parentheses, as shown in the following code fragments.

regexp/Seq.java

```
public String fullParens() {
    return "(" + r1.fullParens() + r2.fullParens() + ")";
}
```

regexp/Alt.java

```
public String fullParens() {
    return "(" + r1.fullParens() + "|" + r2.fullParens() + ")";
}
```

regexp/Rep.java

```
public String fullParens() {
    return "(" + r.fullParens() + "*)";
}
```

Together, these six definitions provide a full definition for the `fullParens()` that can now be used on any `RegExp` value.

A quick test for the code to print fully parenthesized regular expressions. We use the abstract syntax classes to build the representation of a particular regular expression and then call `fullParens()` to display the result:

regexp/FullParensTest.java

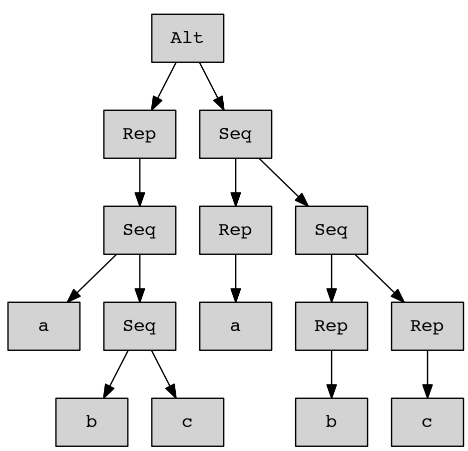
```java
11  public class FullParensTest {
12      public static void main(String[] args) {
13          // Build a representation of the regular expr (abc)*|(a*b*c*)
14          RegExp  a = new Char('a');
15          RegExp  b = new Char('b');
16          RegExp  c = new Char('c');
17          RegExp  r = new Alt(new Rep(new Seq(a, new Seq(b,c))),
18                              new Seq(new Rep(a),
19                                      new Seq(new Rep(b),
20                                              new Rep(c))));
21          System.out.println("r = " + r.fullParens());
22          r.toDot("ast.dot");
23      }
24  }
```

The code in Lines 14–20 shows how the constructors for the five main `RegExp` classes can be used to build up the abstract syntax tree data structure for a given regular expression; in this case, it is `(abc)*|(a*b*c*)`. Note also, that in addition to the `fullParens()` method, described above and called in Line 21, there is also a call to a `toDot()` method in Line 22. This particular call writes a dot description of the AST structure into the specified output file `ast.dot`, which can then be processed using the AT&T GraphViz tools to obtain a corresponding AST diagram. The `toDot()` method is implemented in much the same way as `fullParens()`, placing an abstract definition in the `RegExp` base class, and then providing an appropriate concrete implementations for each distinct AST node type. We will not discuss this further here, but instead encourage the reader to study the accompanying software and explore further details of how this works.

A simple test run of `FullParensTest` produces the following output:

```
$ java regexp.FullParensTest "(abc)*|(a*b*c*)"
r = (((a(bc))*)|((a*)((b*)(c*))))
$
```

In addition, the call to `toDot()` generates a dot file that results in the following AST diagram:



Of course, for more general use, the code on Lines 14–20 of `FullParensTest.java` does not provide a very concise or flexible way to construct abstract syntax trees. In particular, we would need to rewrite and recompile this code every time we wanted to use a different regular expression. As such, apart from providing an opportunity to test some of our earlier code, this example also suggests that it might be more convenient if we had a parser that could be used to construct `RegExp` data structures directly from text strings, without needing to use the explicit constructors. Moreover, it would be useful if our parser made appropriate use of rules for precedence and grouping so that we could write our regular expressions without having to use the fully parenthesized form shown above.

3 Syntax Analysis for Regular Expressions

In this section, we will describe the implementation of a simple lexical analyzer (Section 3.1), and of a parser for our language of regular expressions (Section 3.2). These components can be combined to implement syntactic analysis for regular expressions, and to provide operations that return an appropriate `RegExp` structure for a regular expression that is specified in a text string. Finally, we describe some simple methods that simplify the tasks of creating and connecting a lexer and a parser in the appropriate way so that we can feed in raw input strings and extract useful AST structures (Section 3.3).

3.1 A Lexer for Regular Expressions

Working directly from the grammar, we can see that there are only a few different token types. These include parentheses (the `(` and `)` tokens); special symbols (the `|`, `*`, and `%` tokens, the latter being how we represent the ε symbol in our concrete syntax); and single characters, which we will represent using the `c` character. Starting from these observations, we can construct a lexical analyzer as follows:

regexp/RegExpLexer.java

```
/** A hand-coded lexer for use in reading regular expressions.
 */
public class RegExpLexer extends SourceLexer {
    public RegExpLexer(Handler handler, Source source) {
        super(handler, source);
    }

    /** Return a code describing the next token in the input
     *  stream, and returning 0 at the end of the input.  We
     *  use the ASCII values of the characters (, ), *, |, and
     *  % as the codes for the corresponding tokens (% is used
     *  in place of epsilon); we use the character c as the code
     *  for a single character regular expression (saving the
     *  actual character value in tokChar); and we use a 0 code
     *  to signal the end of the input.
     */
    public int nextToken() {
        for (;;) {
          switch (c) {
              case EOF  : return token=0;  // end of input
              case ' '  :                  // skip whitespace
              case '\t' :
              case '\r' : c = nextChar();
                          continue;
              case EOL  : nextLine();      // skip newlines
                          continue;
              case '('  :
              case ')'  :
```

```java
                case '*'   :
                case '|'   :
                case '%'   : token = c;
                             c     = nextChar();
                             return token;
                case '\\'  : c = nextChar();
                             if (c==EOF) {
                                 report(new Failure("Missing character"));
                                 return '\\';
                             }
                             /* intentional fall-thru */
                default    : tokChar = c;
                             c       = nextChar();
                             return token = 'c';
            }
          }
      }

      /** Stores the value of the most recently read single character token.
       */
      private int tokChar;

      /** Return a single character regular expression for matching the most
       *  recently read single character.
       */
      public RegExp getSemantic() { return new Char(tokChar); }
  }
```

Note that our lexical analyzer makes two concessions to practical use. First, it allows whitespace to be included in a regular expression so that the input can be spread over several lines or laid out with embedded spaces so that it is easier to read. Second, we provide an ‘escape’ mechanism that allows special characters like `(` or `*` to be treated like any other character if they are preceded by a backslash. In particular, a single backslash can be entered using `\\`. Note that the lexer does not distinguish between an input character, such as `x`, and an escaped version of the same character, such as `\x`: in both cases, the lexer returns `c` as the token type, and sets the `tokChar` variable to `'x'` so that our parser will be able to determine, not just that a single character token was found, but also to determine exactly *which* single character token was found.

To demonstrate that our lexer is working, we will use the following small test program, which reads a regular expression from the strings passed in on the command line argument and outputs the corresponding sequence of token characters. Note that this program uses the `StringArraySource` class from the `compiler` package to construct a suitable source input for the lexer:

regexp/LexerTest.java

```java
public class LexerTest {
    public static void main(String[] args) {
        Handler        handler = new SimpleHandler();
        Source         source  = new StringArraySource(handler,"input",args);
        RegExpLexer    lexer   = new RegExpLexer(handler, source);

        while (lexer.nextToken()!=0) {
          System.out.print((char)lexer.getToken());
        }
        System.out.println();
    }
}
```

The following session shows a simple test run[1]:

```
$ java regexp.LexerTest "(abc)*|(a*b*c*)"
(ccc)*|(c*c*c*)
$
```

The output here reflects the structure of the input, but replaces each single character in the regular expression with the character `c`. Of course, this is just the symbol that our lexer uses to represent such tokens.

3.2 A Parser for Regular Expressions

Our next task, building on the lexer that was introduced in the previous section, is to implement a parser for regular expressions. The technique that we choose for this is known as *recursive descent parsing*, which is a form of *top-down parsing*, and often works well for small, handwritten parsers. We will discuss these ideas further in future weeks; for the purposes of these notes, however, it will suffice just to develop an intuitive understanding of how the parser works.

Our first task is to figure out how we should resolve the ambiguities in the original grammar given in Section 1. Following standard conventions, we will treat repetition as having high precedence and alternatives as having low precedence with sequencing in between. In addition, we will treat both sequencing and alternatives as grouping to the right. These design choices are reflected in the following rewritten (and now unambiguous) grammar for regular expressions:

$$
\begin{array}{lcl}
regexp & \rightarrow & seq \mid regexp \\
regexp & \rightarrow & seq \\
 & & \\
seq & \rightarrow & repseq \\
seq & \rightarrow & rep \\
 & & \\
rep & \rightarrow & rep * \\
rep & \rightarrow & atom \\
 & & \\
atom & \rightarrow & \varepsilon \\
atom & \rightarrow & \mathtt{c} \\
atom & \rightarrow & (\, regexp\,)
\end{array}
$$

Now, following the structure of this grammar, we can construct a recursive descent parser, including one parse function (`regexp()`, `seq()`, `rep()`, and `atom()`) in our code for each of the nonterminals ($regexp$, seq, rep, and $atom$) in the grammar.

regexp/RDRegExpParser.java

```java
    /** Parse a regexp: regexp = seq | seq '|' regexp
     */
    private RegExp regexp() {
        RegExp r = seq();              // read a sequence
        if (token=='|') {              // look for a '|'
            nextToken();               // ... followed by another
            r = new Alt(r, regexp());  //     regexp
        }
        return r;
    }
```

[1] We must include quotes around the regular expression argument so that the command line interpreter doesn't try to treat the `|` character as a pipe symbol!

```java

    /** Parse a seq:   seq = rep  |  rep seq
     */
    private RegExp seq() {
        RegExp r = rep();              // read a rep
        if (token=='c' || token=='%' || token=='(') {
                                       // if an atom could come next,
            r = new Seq(r, seq());     // then look for another seq()
        }
        return r;
    }

    /** Parse a rep:   rep = atom  |  rep '*'
     */
    private RegExp rep() {
        RegExp r = atom();             // read an atom
        while (token=='*') {           // followed by zero or more '*'s
            nextToken();
            r = new Rep(r);
        }
        return r;
    }

    /** Parse an atom:  atom  =  '(' regexp ')'  |  'c'  |  '%'
     */
    private RegExp atom() {
        if (token=='c') {               // check for single character
            RegExp r = lexer.getSemantic();
            nextToken();
            return r;
        } else if (token=='%') {        // check for an epsilon
            nextToken();
            return new Epsilon();
        } else if (token=='(') {        // look for a parenthesized
            nextToken();                // expression ...
            RegExp r = regexp();
            if (token==')') {
                nextToken();
            } else {
                report(new Failure(lexer.getPos(),
                                   "missing close parenthesis"));
            }
            return r;
        }
        report(new Failure(lexer.getPos(),
                           "syntax error in regular expression"));
        return new Epsilon(); // represents missing regular expression
    }
```

Note that all of this code is placed in a class called `RDRegExpParser` (short for 'recursive descent regular expression parser'). The same file also includes some small fragments of code, not shown here, to ensure that the parser will read its input from a `RegExpLexer` object, `lexer`, that is passed in as an argument of the `RDRegExpParser` class.

3.3 Putting it Together

To make the code in the previous two sections a little easier to use, we will define the following convenience methods. These functions package up a lexer and a parser in an appropriate way to convert the text for a regular expression in to a corresponding `RegExp` structure.

regexp/RDRegExpParser.java

```
    /** Convenience method that combines a lexer and a parser to parse
     *  a sequence of strings as a regular expression.
     */
    public static RegExp parse(Handler handler, String[] args) {
        RDRegExpParser parser  = new RDRegExpParser(handler);
        Source         source  = new StringArraySource(handler,"input",args);
        RegExpLexer    lexer   = new RegExpLexer(handler, source);
        return parser.parseRegExp(lexer);
    }

    /** Parse a single string to extract the AST for a regular expression.
     */
    public static RegExp parse(Handler handler, String arg) {
        return parse(handler, new String[] { arg });
    }
```

The first method takes its input from an array of strings, which might be useful when the regular expression to be read is spread over multiple lines (or, as we will use it here, multiple command line arguments). The second method is a simple variation that takes its input from a single string. Note that this code uses a `StringArraySource` object, which is defined in the `compiler` package. This provides a simple way to construct an input source from an array of strings instead of reading it from a file (as the `JavaSource` that we saw in the first lab would do, for example).

Once again, we use a small test program to experiment with our implementation. The following code combines elements from the previous `FullParensTest` and `LexerTest` programs, but differs in two respects. First, unlike `FullParensTest`, it constructs a `RegExp` from the strings passed in on the command line argument instead of using the abstract syntax constructors like `Seq`. And second, unlike `LexerTest`, it passes the `lexer` that it constructs to a `parser` instead of reading the sequence of token codes directly.

regexp/RDParserTest.java

```
public class RDParserTest {
    public static void main(String[] args) {
        RegExp r = RDRegExpParser.parse(new SimpleHandler(), args);
        System.out.println("r = " + r. fullParens());
        r.toDot("ast.dot");
    }
}
```

In the following session, for example, we use the combined lexer and parser to make sense of the input regular expression, and then output an equivalent, but fully parenthesized version of the same regular expression:

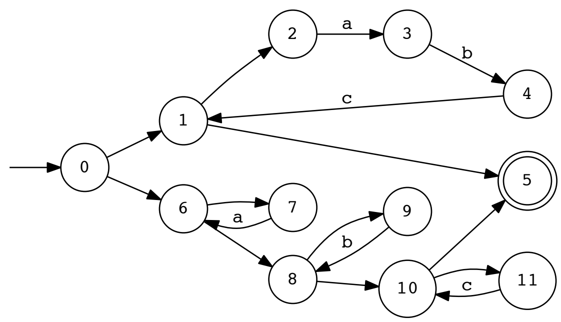
```
$ java regexp.RDParserTest "(abc)*|(a*b*c*)"
r = (((a(bc))*)|((a*)((b*)(c*))))
$
```

Behind the scenes, this code also generates a dot file, `ast.dot`; we do not include the resulting tree structure here because, for the specific example shown here, it it just the same as the AST diagram that was included in Section 2.2.

[Aside: At this point, some readers may enjoy the exercise of defining a new method, `minParens()` on each of the `RegExp` classes that behaves like `fullParens()` except that it includes only the minimal number of parentheses. If this is implemented correctly, then it should be possible to take an arbitrary (non-null) `RegExp` value, `r`, convert it to a string using `minParens()`, and then use the parser in `RDRegExpParser` to construct a new `RegExp` value that has exactly the same structure as the original structure `r`.]

4 Representing NFAs

Having developed an abstract syntax and supporting syntax analysis for regular expressions, we would like to add mechanisms for "compiling" regular expressions into NFAs (nondeterministic finite automata, or machines), as illustrated in the following diagram:



This NFA recognizes strings in the language described by the regular expression `(abc)*|(a*b*c*)`. Each of the circles represents an NFA state; the numbers inside each state are used only for the purposes of identification and do not carry any deeper meaning. Matching begins with the leftmost state, numbered 1, and terminates when we reach the accept state, numbered 5, and indicated by the double circle. In between these points, the NFA can make transitions between states as indicated by the arrows. On arrows that have an associated label, the NFA must match and consume the next character in the input against the character in the label to make the transition. On arrows that have no associated label, however, the NFA can make the transition without matching any input. These latter transitions are sometimes referred to as ε-transitions (or, by some authors, as λ-transitions).

Given these preliminaries, we can provide a representation for individual states using the follow class, which stores an array of (zero or more) outgoing transitions for the state as well as a code to distinguish normal states from accept states (the latter being the only kind of state in which the machine is permitted to terminate).

regexp/State.java

```
class State {
    /** The set of transitions associated with this state.
     */
    Transition[] trans = null;

    /** An accept code for this state.  A zero value indicates that
     *  this is not an accepting state.  A positive value indicates
     *  an accept state.  We can use multiple distinct positive
```

```
     *  accept codes within a given machine to represent different
     *  "reasons" for being able to accept.
     */
    int accept = 0;

    ...
}
```

Individual transitions, corresponding to arrows in NFA diagrams, are also easy to represent in Java by using values of the following `Transition` class. Because transitions are stored in arrays in the states in which they originate, we only need to store the destination state and an associated label for each `Transition`:

regexp/Transition.java

```
class Transition {
    /** Captures the input symbol/character for this transition.
     *  The epsilon code defined below is used to identify epsilon
     *  transitions (i.e., transitions that do not consume any
     *  input).
     */
    int   on;

    /** The target state for this transition.
     */
    State target;

    /** Construct a transition.
     */
    Transition(int on, State target) {
        this.on = on;
        this.target = target;
    }

    /** Special code used to signal an epsilon transition.
     */
    static final int epsilon = '\0';

    /** Construct an epsilon transition to a specified target.
     */
    Transition(State target) {
        this(epsilon, target);
    }
}
```

In this implementation, ε-transitions are encoded by using the value `epsilon` as the value for the `on` label, and we provide a special constructor for ε-transitions, at the end of the class definition, that requires only a target for the transition.

We can generate a textual description of an individual state using the following code:

regexp/State.java

```
    /** Output a description of this machine state.
     */
    void display() {
        System.out.println("State no: " + num);
        if (accept>0) {
            System.out.println("Accept state! [code="+accept+"]");
```

```
        }
        for (int i=0; i<trans.length; i++) {
            int    c  = trans[i].on;
            State to = trans[i].target;
            if (c==Transition.epsilon) {
                System.out.println("Epsilon transition to " + to.num);
            } else {
                System.out.println("Transition on " + (char)c +
                                   " to state " + to.num);
            }
        }
        System.out.println();
    }
```

One immediate problem here is the assumption that we have access to distinct `num` fields for identifying individual `State` values. The following code provides a definition for such `num` fields, together with some logic for setting them to appropriate values. The first step in this process is to initialize the `num` field to `(-1)` when the associated state is created. Then, given the start state for a newly constructed NFA, we can perform a depth-first search of the whole machine, assigning consecutive positive numbers to each new `State` that we find:

regexp/State.java

```
    /** Holds the number/unique identifier of this state.  A negative
     *  value here indicates that the state has not yet been assigned
     *  a number.
     */
    int num = (-1);

    /** Perform a depth-first search of the automaton, starting at
     *  this state, so that every reachable state receives a distinct
     *  identifier.  The returned result indicates the number of
     *  distinct states that were encountered.
     */
    public int numberStates(int n) {
        if (num<0) {
            num = n++;
            for (int i=0; i<trans.length; i++) {
                n = trans[i].target.numberStates(n);
            }
        }
        return n;
    }
```

Note that the `numberStates()` method threads a parameter, `n`, through the search to reflect the next unused state number at each stage. In particular, if we have an NFA with start state `s`, then, in addition to assigning each state a distinct number, the call to `s.numberStates(0)` will also return the total number of states that are reachable from `s`.

Numbering the states solves one problem, but there is still some work to do before we can print out a description of a generated NFA. Of course, it would be sufficient simply to call the `display()` method on each of the states in the NFA, but this is not easy to do with the code shown so far because we have no way to iterate through all of the states.

Once again, depth-first search comes to our rescue, this time in the form of a method `collectStates()` that passes an array of `State` values around as it traverses a freshly generated and numbered NFA. The entries in the array are initially `null`, but when the search encounters a state with some particular `num` for the first time, then it will store (a pointer to) that state in the `num`th entry of the array. This also serves as a way to flag that particular state as having been 'visited' so that this second depth first search will terminate properly. As the search progresses, it will eventually fill in the appropriate `State` values for each possible array index.

regexp/State.java

```
    /** A follow-up to numberStates that performs a second
     *  depth-first search, building up an array of all the
     *  states in this machine.
     */
    public void collectStates(State[] states) {
        if (states[num]==null) {
            states[num] = this;
            for (int i=0; i<trans.length; i++) {
                trans[i].target.collectStates(states);
            }
        }
    }
```

For example, given the start state, `m`, of a newly created NFA, we can use code of the following form to construct an array that includes all of its reachable states, and then produce a textual description of the full NFA.

```
    int count      = m.numberStates(0);
    State[] states = new State[count];
    m.collectStates(states);
    System.out.println("number of states = " + states.length);
    for (int st=0; st<states.length; st++) {
        states[st].display();
    }
    DotOutput.toDot(states, "nfa.dot");
```

As the last line of code suggests here, the accompanying software also includes code for generating diagrams, in dot format, of the NFAs that we produce: the NFA diagram at the start of this section, for example, was produced in this way. We will not describe this feature further in these notes, but readers are welcome to consult the software for further details.

5 Generating NFAs from Regular Expressions

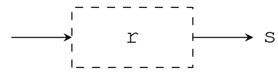
Now that we have concrete representations for both regular expressions and NFAs, we can implement standard algorithms that take regular expressions as input and produce NFAs that match the same language as output.

The overall structure of the implementation we provide here follows the same approach as the implementation of the `fullParens()` method in Section 2.2. This requires the definition of a basic method signature in the `RegExp` base class, together with the addition of specific implementations for each of the different `RegExp` subclasses.

More specifically, we rely on calls of the form `r.toNFA(s)` where `r` is a regular expression, and `s` is a follow-on state; the intention is that this expression will return (a pointer to) the start state of a machine that recognizes a string in the language described by `r`, and then continues on to the state `s`:

regexp/RegExp.java

```java
/** Construct an NFA that will recognize a string matching this
 *  regular expression and then transition to the follow-on NFA
 *  with start state s.
 */
abstract State toNFA(State s);
```



In the case of an `Epsilon` regular expression, there is no need to create a new state; we simply return the pointer to the follow-on state `s`:

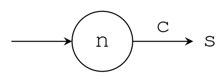
regexp/Epsilon.java

```java
State toNFA(State s) {
    return s;
}
```


For a single character, `Char`, regular expression, we return a pointer to a new state, `n`, that has a single transition, on the specified character, to `s`:

regexp/Char.java

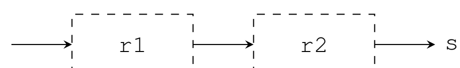
```java
State toNFA(State s) {
    State n = new State();
    n.trans = new Transition[] {
      new Transition(c, s)
    };
    return n;
}
```



The case for sequences, `r1 r2`, is particularly concise and elegant: we use recursive calls to create a machine that will: recognize a string in the language described by `r1`; then transition to a machine that will recognize a string in the language described `r2`; and only then transition to the original follow-on state, `s`:

regexp/Seq.java

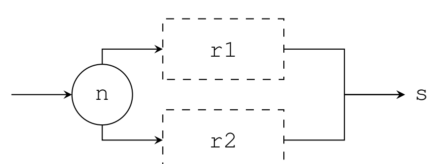
```java
State toNFA(State s) {
    return r1.toNFA(r2.toNFA(s));
}
```



For alternatives, `r1|r2`, we create a new state, `n`, with ε-transitions to two new machines, one that will recognize a string in the language for `r1` before transitioning to `s`, and one that will recognize a string in the language for `r2`, again before transitioning to `s`:

regexp/Alt.java

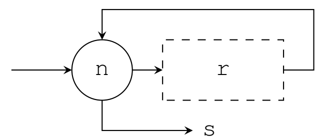
```java
State toNFA(State s) {
    State n = new State();
    n.trans = new Transition[] {
      new Transition(r1.toNFA(s)),
      new Transition(r2.toNFA(s))
    };
    return n;
}
```



Finally, the case for repetitions, `r*`, involves creating a new state, `n`, with two ε-transitions. One of these transitions takes us directly to the follow-on state, `s`; this corresponds to the case where we have matched zero occurrences of strings in the language for `r`. The other transition takes us to a machine, `r.toNFA(n)`, that recognizes one string in the language for `r` and then loops back to `n`, allowing for further iterations:

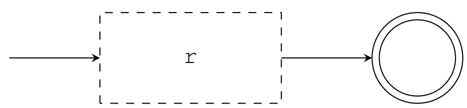
regexp/Rep.java

```
State toNFA(State s) {
    State n = new State();
    n.trans = new Transition[] {
      new Transition(r.toNFA(n)),
      new Transition(s)
    };
    return n;
}
```



As an aside, notice that the code in these method bodies corresponds very closely to the instructions that you might give somebody if you were explaining how to draw the appropriate NFA by hand: Each `new State()` call corresponds to drawing a new state node and each `new Transition(..)` call corresponds to drawing an arrow between states.

The `toNFA()` method implementations defined above are designed to be used when we are generating a small fragment of a larger machine before transitioning on to some appropriate follow-on state. If the goal instead is just to recognize a particular regular expression and then transition to an accept state, then we just need to build a machine of the following form:



In terms of our current implementation, we can describe this by the following variant of the `toNFA()` method:

regexp/RegExp.java

```
/** Construct an NFA that will recognize a string matching this
 *  regular expression and then transition to a state that will
 *  accept with the given integer code.
 */
public State toNFA(int accept) {
   State   s = new State();        // Build a new state
   s.accept = accept;              // with specified accept code
   s.trans  = new Transition[0]; // and no outgoing transitions.
   return this.toNFA(s);           // Generate recognizer.
}
```

Note that the parameter for this version of `toNFA()` is not a follow-on state, but just an integer that specifies an accept code (which, for simple cases, would just be the integer `1`).

6 Generating DFAs from NFAs

Although we will not describe the code in detail in these notes, the accompanying software includes an implementation of the "subset construction" that can generate a deterministic finite automaton or DFA that recognizes the same language as an input NFA. The key idea is that each state in the generated DFA is labeled with a set of NFA states; these are precisely the states that the NFA could be in after any sequence of input characters that leads to the given DFA state. The implementation of this feature is spread across three classes: `DFAState` is an extension of the previously described `State` class that allows each `DFAState` to be labeled with a set of numbers (corresponding to the set of NFA state numbers); the `SubsetConstruction` class implements the main algorithm for building a DFA; and the `DFATrans` class is used during this process to build up lists of transitions.

The complete process of generating a DFA for a given regular expression, `r`, can then be implemented by the following code sequence:

```
1	// Generate an NFA, count its states, and then collect the set
2	// of all NFA states in an appropriately sized array:
3	State m        = r.toNFA(1);
4	int count      = m.numberStates(0);
5	State[] states = new State[count];
6	m.collectStates(states);
7
8	// Run the subset construction to generate a corresponding DFA:
9	State start = new SubsetConstruction(states).getDFA();
10	State[] dfa = new State[start.numberStates(0)];
11	start.collectStates(dfa);
```

The key steps here are the calls `r.toNFA(1)` in Line 3, and `new SubsetConstruction(states).getDFA()` in Line 9. The first of these generates the NFA, while the second produces the corresponding DFA. In between, the remaining steps just use the `numberStates()` and `collectStates()` methods, introduced in Section 4, to build arrays of NFA and DFA states suitable for displaying the generated machines.

7 Matching using a DFA

Regular expressions are widely used as a notation for describing patterns of characters within strings, but in all that we have done so far, we have focused instead on issues of how they are represented, parsed, and used to generate NFAs and the DFAs. In this section, we will, at last, present some code that uses a DFA to perform string matching!

We will implement our string matching algorithm as a method of the `State` class that takes a `String` input, `s`, and a starting position, `pos`, within that string. The return result is an integer value that is either `-1`, if there is no match, or else a positive number, which is the furthest position that the matching process was able to reach before it either reached the end of the string or else got stuck. In particular, if the return value is `pos`, then we can determine that the empty string was a valid match, and if the return value is `s.length()`, then the whole string, from `pos` onwards, was a valid match.

The `match()` method implements the longest lexeme/maximal munch rule. In other words, `match()` does not return at the first accept state state it reaches, and instead tracks the position at which the last accept state was reached (in the `acceptPos` variable, together with a pointer to the state itself in the `acceptState` field), only returning when the input has been consumed or when the machine gets stuck.

regexp/State.java

```
/** Attempt to match the given string, beginning at the specified position,
 *  using this state as the start state.  Matching may not succeed if the
 *  machine is non-deterministic because it may choose the "wrong"
 *  transitions; this is one of the reasons why we prefer to use a DFA
 *  instead of an NFA.
 */
```

```
    public int match(String s, int pos) {
        State current    = this;      // track current state in DFA
        int   acceptPos = (-1);       // position of last accept state
        acceptState     = null;
        while (current!=null) {
            if (current.accept>0) {   // is this an accept state?
                acceptState = current;
                acceptPos   = pos;
            }
            if (pos>=s.length()) {    // finished reading input?
                break;
            }
            Transition[] trs = current.trans;
            int    c = s.charAt(pos);
            current = null;           // look for matching transition
            for (int i=0; i<trs.length; i++) {
                if (c==trs[i].on) {
                    current = trs[i].target;
                    pos++;
                    break;
                }
            }
        }
        return acceptPos;
    }

    /** Records the last accept state that was encountered during matching.
     */
    public static State acceptState = null;

```

8 A Simple Demonstration

The following code brings together many of the features that we have described in previous sections as a simple test program that: accepts an input regular expression; generates a corresponding NFA; produces an equivalent DFA; and then begins a loop, allowing the user to enter a sequence of strings, and then using the DFA to match each one against the original regular expression.

regexp/RegExpTest.java

```
public class RegExpTest {
    public static void main(String[] args) {
        // Read regular expression:
        Handler handler = new SimpleHandler();
        RegExp  r       = RDRegExpParser.parse(handler, args);

        // Print out in fully parenthesized form:
        System.out.println("r = " + r. fullParens());
        r.toDot("ast.dot");

        // Now build an NFA for r, number its states, and
        // collect them together in an array.
        State m         = r.toNFA(1);
        int count       = m.numberStates(0);
        State[] states = new State[count];
```

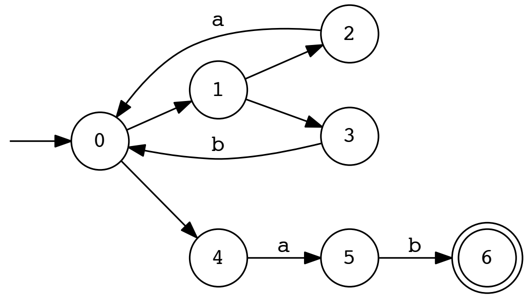
```java
        m.collectStates(states);

        // Output a description of the machine, including the
        // transitions from each state.
        System.out.println("number of NFA states = " + states.length);
        for (int st=0; st<states.length; st++) {
            states[st].display();
        }
        DotOutput.toDot(states, "nfa.dot");

        // Run the subset construction to generate a corresponding
        // DFA and then output a description of that DFA.
        State start = new SubsetConstruction(states).getDFA();
        State[] dfa = new State[start.numberStates(0)];
        start.collectStates(dfa);
        System.out.println("number of DFA states = " + dfa.length);
        for (int st=0; st<dfa.length; st++) {
            dfa[st].display();
        }
        DotOutput.toDot(dfa, "dfa.dot");

        // Read input lines and use the generated DFA to match them
        // against the original regular expression.
        Source input = new StdinSource(handler);
        String line;
        System.out.println("Enter text to match, or an empty line to end:");
        while ((line=input.readLine())!=null && line.length()>0) {
            int n = dfa[0].match(line, 0);
            if (n<0) {
                System.out.println("No match!");
            } else {
                for (int i=0; i<n; i++) {
                    System.out.print(" ");
                }
                System.out.println("^");
            }
        }
    }
```

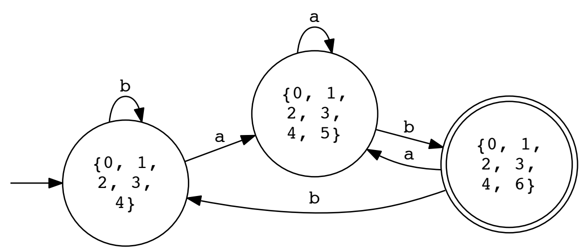
The last section of the code shown here uses a `StdinSource` object (from the `compiler` package) to read input lines from the user, and then uses `match()` on each one to determine whether it matches the regular expression that was entered on the command line and used to generate the DFA. In the case of a (partial) match, it prints a caret/up-arrow symbol under the first character that could not be matched. (If the whole string was matched, then the caret appears after the last character in the string.) The program is terminated by entering an empty line or the end of file character (a control Z on most Windows machines, and a control D on most Unix/Linux/MacOS X boxes).

The following transcript shows the output that is produced by running `regexp.RegExpTest` with the regular expression `(a|b)*ab` as input.

```
$ java regexp.RegExpTest "(a|b)*ab"
r = (((a|b)*)(ab))
number of NFA states = 7
State no: 0
Epsilon transition to 1
```

```
Epsilon transition to 4

State no: 1
Epsilon transition to 2
Epsilon transition to 3

State no: 2
Transition on a to state 0

State no: 3
Transition on b to state 0

State no: 4
Transition on a to state 5

State no: 5
Transition on b to state 6

State no: 6
Accept state! [code=1]

number of DFA states = 3
{0, 1, 2, 3, 4}State no: 0
Transition on b to state 0
Transition on a to state 1

{0, 1, 2, 3, 4, 5}State no: 1
Transition on b to state 2
Transition on a to state 1

{0, 1, 2, 3, 4, 6}State no: 2
Accept state! [code=1]
Transition on b to state 0
Transition on a to state 1

Enter text to match, or an empty line to end:
ababab
      ^
abababaaaabaa
           ^
baaaaaaaaa
No match!
abaaaaaab
         ^
$
```



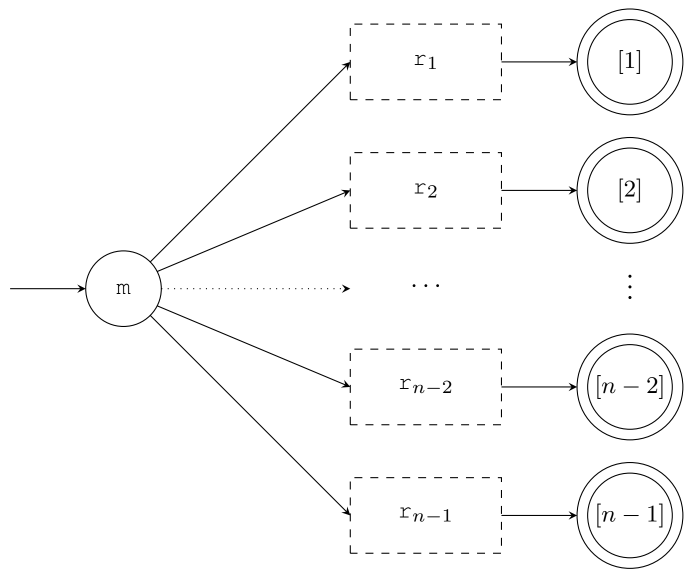


The diagrams on the right above, of course, show the structure of the generated NFA and DFA in graphical form, and are included here as an aid to interpreting the textual output. The output ends with some examples that use the DFA for matching text inputs against the original regular expression.

9 A Simple Lexer Generator

As a final demonstration, we end these notes with the description of a program that captures key parts of the operation of a simple lexer generator. Full code is included in the `LexerGenTest` class, which is part of the accompanying software. We will only present fragments of that implementation here.

The `LexerGenTest` program takes a sequence of n regular expressions, $\mathtt{r}_1$, $\mathtt{r}_2$, ..., $\mathtt{r}_{n-2}$, $\mathtt{r}_{n-1}$, and uses these to construct an NFA of the following form:



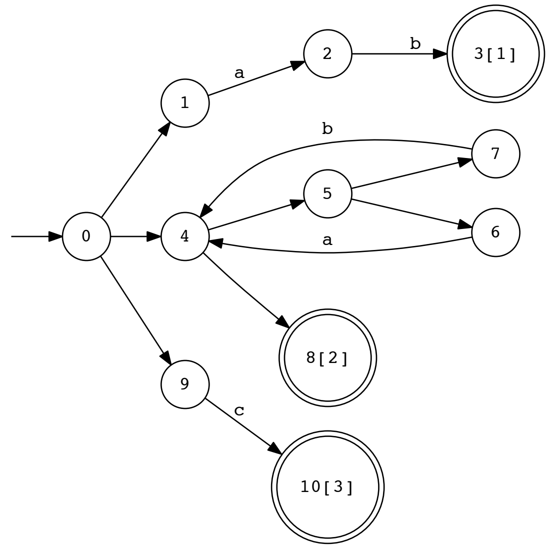
The intention here is that the regular expressions correspond to the list of patterns that might be provided on the left hand side of each of the lexical rules in a typical `lex` or `jflex` specification. Note also that each of the accept states in this diagram is labeled with a different accept code, indicated by the numbers inside the square brackets. In the context of a real lexer generator, these might be used as references to the actions associated with each of the lexical rules. The code for constructing this particular NFA—including the start state, `m`, and the associated ε-transitions to the NFAs for each component regular expression—is straightforward given the methods that we have already implemented:

regexp/LexerGenTest.java

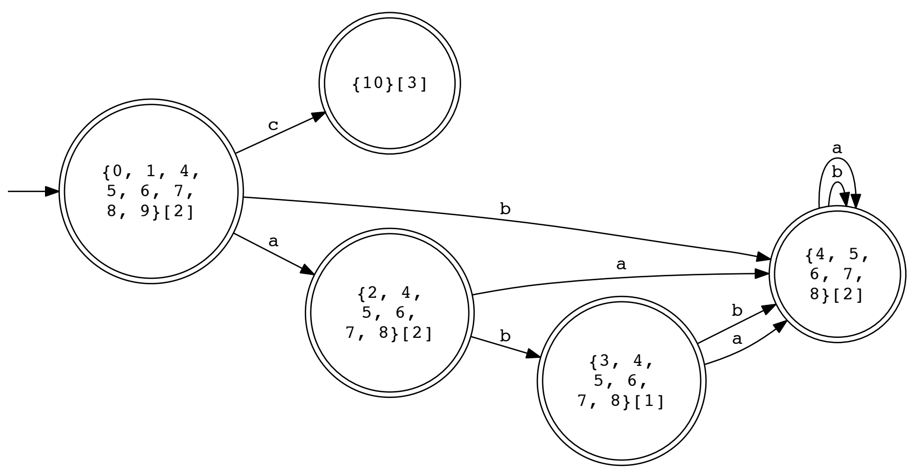
```
        State m = new State();
        m.trans = new Transition[args.length];
        for (int i=0; i<args.length; i++) {
            // Build an accepting machine that will recognize the
            // ith regular expression and then accept with code i+1.
            RegExp r = RDRegExpParser.parse(handler, args[i]);
            m.trans[i] = new Transition(r.toNFA(i+1));
        }
```

From this point, we can proceed as before to construct a DFA that can be used for matching. The only complication is that we might now end up with DFA states that contain multiple accepting NFA states with distinct accept codes. To resolve this, we use the same policy as typical lexer generators by giving priority to rules that appear early in the input over rules that appear later. In particular, this means that we will always pick the lowest possible (nonzero) accept code when there is a choice to be made between multiple accepting NFA states.

As a concrete example, running `java regexp.LexerGenTest "ab" "(a|b)*" "c"` produces the following NFA, in which the separate machines for each of the three regular expressions, as provided on the command line, are still clearly visible:



This, in turn, can be used to produce a DFA, effectively merging the separate branches of the NFA to obtain the following structure:



Note here, for example, that the path from the start state that corresponds to the string `ab` has accept code 1, indicating that it has chosen the first regular expression, `ab`, even though the second, `(a|b)*`, would also have matched. On the other hand, if we follow the path for the string `abb`, then we arrive in a state with accept code 2, having now ruled out the first regular expression by finding a longer lexeme that matches the second.

Finally, we can reuse the code for matching to implement a simple lexical analyzer that can break an input string in to multiple distinct lexemes, each of which matches one of the original three regular expressions. The key idea here is that, having found one lexeme that spans from positions `pos` to `newpos`, we begin our search for the next token at position `newpos`. This process stops only when `newpos` has not advanced beyond `pos` (either because `newpos` is `-1`, indicating a failure to match, or because the longest possible match at this point is the empty string, so continuing with further calls to `match()` would only result in an infinite loop):

regexp/LexerGenTest.java

```java
        Source input = new StdinSource(handler);
        String line;
        System.out.println("Enter text to tokenize, or an empty line to end:");
        while ((line=input.readLine())!=null && line.length()>0) {
            int pos = 0;
            int newpos;
            while ((newpos = dfa[0].match(line, pos)) > pos) {
                System.out.println("Matched \""
                                  + line.substring(pos, newpos)
                                  + "\", accept code = "
                                  + State.acceptState.accept);
                pos = newpos;
            }
            if (newpos<line.length()) {
                System.out.println("Unmatched trailing input: "
                                  + line.substring(newpos));
            }
        }
```

The following script illustrates the behavior of this code by showing how it breaks down several input strings into sequences of lexemes (the input strings are entered on Lines 2, 7, and 11):

```
1  Enter text to tokenize, or an empty line to end:
2  aaabbcabc
3  Matched "aaabb", accept code = 2
4  Matched "c", accept code = 3
5  Matched "ab", accept code = 1
6  Matched "c", accept code = 3
7  abcd
8  Matched "ab", accept code = 1
9  Matched "c", accept code = 3
10 Unmatched trailing input: d
11 bbabcab
12 Matched "bbab", accept code = 2
13 Matched "c", accept code = 3
14 Matched "ab", accept code = 1
```

Of course, this provides only the basic components of a lexer generator. A more realistic system, for example, would typically also include code to:

- minimize the size of the generated DFA;
- support a broader range of regular expressions;
- allow the definition and use of named regular expressions;
- label transitions with sets of characters (character classes) rather than requiring a separate transition for each distinct character in the input alphabet;
- integrate actions and other code fragments written in the target language of the lexer generator.