Lexical Analysis

Outline

Scanning Tokens

The first step in compiling a program is to break it into tokens (aka lexemes); for example, given the *j--* program

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
    }

    static int n = 5;
}
```

we want to produce the sequence of tokens `package`, `pass`, `;`,`import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`,`class`, `Factorial`, `{`,`public`, `static`, `int`,`factorial`, `(`,`int`, `n`,`)`, `{`, `if`, `(`, `n`,`<=`, `0`, `)`, `}`, `return`, `1`,`;`, `else`,`return`, `n`, `*`, `factorial`, `(`, `n`, `-`, `1`, `)`, `}`, `;`, `}`, `public`, `static`, `void`, `main`, `(`, `String`, `[`, `]`, `args`, `)`, `}`, `{`, `int`, `x`, `=`, `n`, `;`, `System`, `.`, `out`, `.`, `println`, `(`, `x`, `+`, `"!="`, `+`, `factorial`, `(`, `x`, `)`, `)`, `}`, `;`, `}`, `static`, `int`, `n`, `=`, `5`,`;`, and `}`

Scanning Tokens

We separate the lexemes into categories; in our example program

- `public`, `class`, `static`, and `void` are reserved words
- `Factorial`, `main`, `String`, `args`, `System`, `out`, and `println` are all identifiers
- The string `!=` is a literal, a string literal in this instance
- The rest are operators and separators

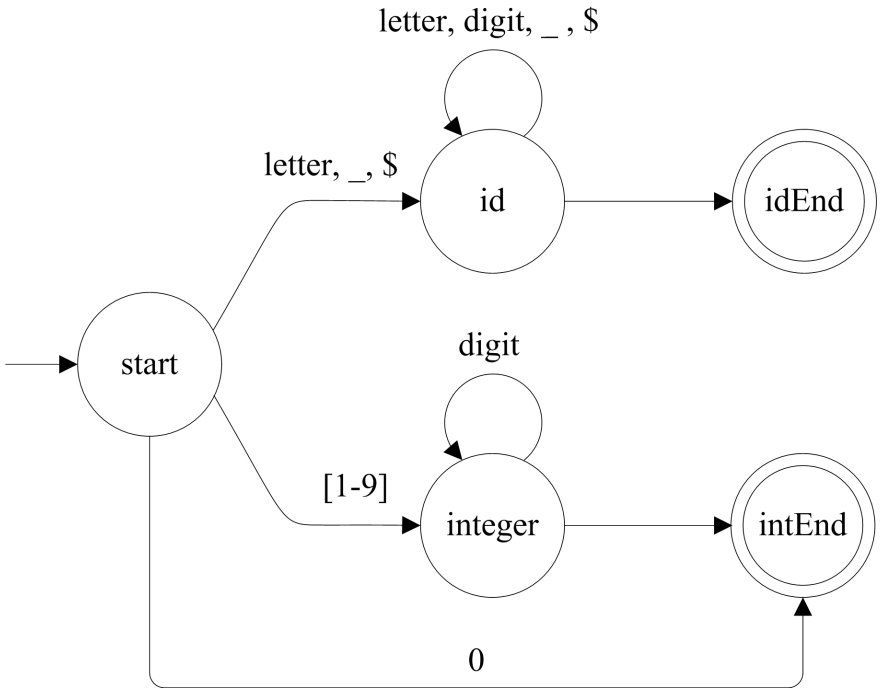
The program that breaks the input stream of characters into tokens is called a lexical analyzer or a scanner

A scanner may be hand-crafted or it may be generated automatically from a specification consisting of a sequence of regular expressions

A state transition diagram is a natural way of describing a scanner

Scanning Tokens

A state transition diagram for identifiers and integers and the corresponding code



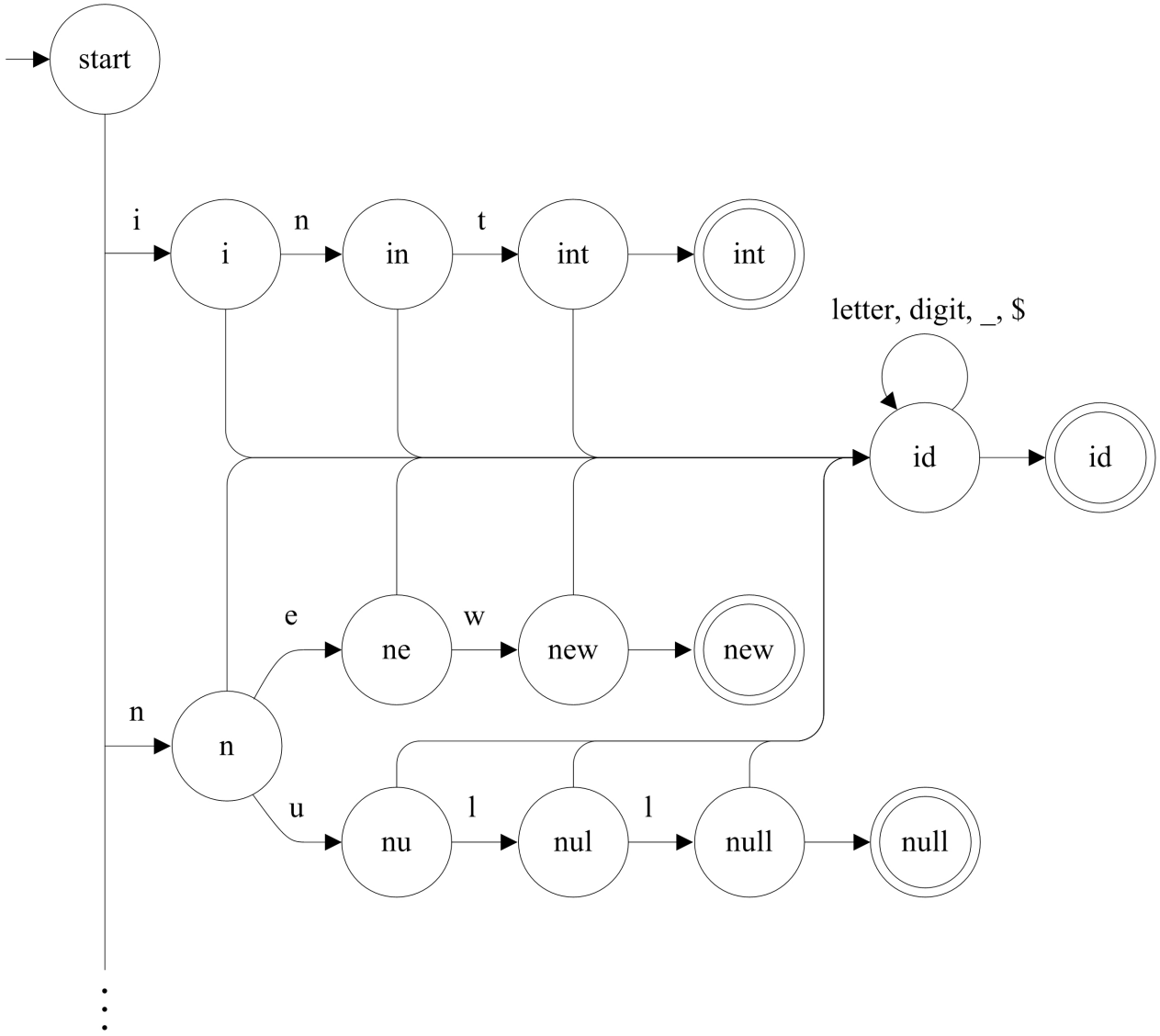

```
    if (isLetter(ch) || ch == '_' || ch == '$') {
        buffer = new StringBuffer();
        while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
            buffer.append(ch);
            nextCh();
        }
        return new TokenInfo(IDENTIFIER, buffer.toString(), line);
    }
```

Scanning Tokens

```
    else if (ch == '0') {
        nextCh();
        return new TokenInfo(INT_LITERAL, "0", line);
    }
    else if (isDigit(ch)){
        buffer = new StringBuffer();
        while (isDigit(ch)) {
            buffer.append(ch);
            nextCh();
        }
        return new TokenInfo(INT_LITERAL, buffer.toString(), line);
    }
```

Scanning Tokens

A state transition diagram for reserved words and the corresponding code



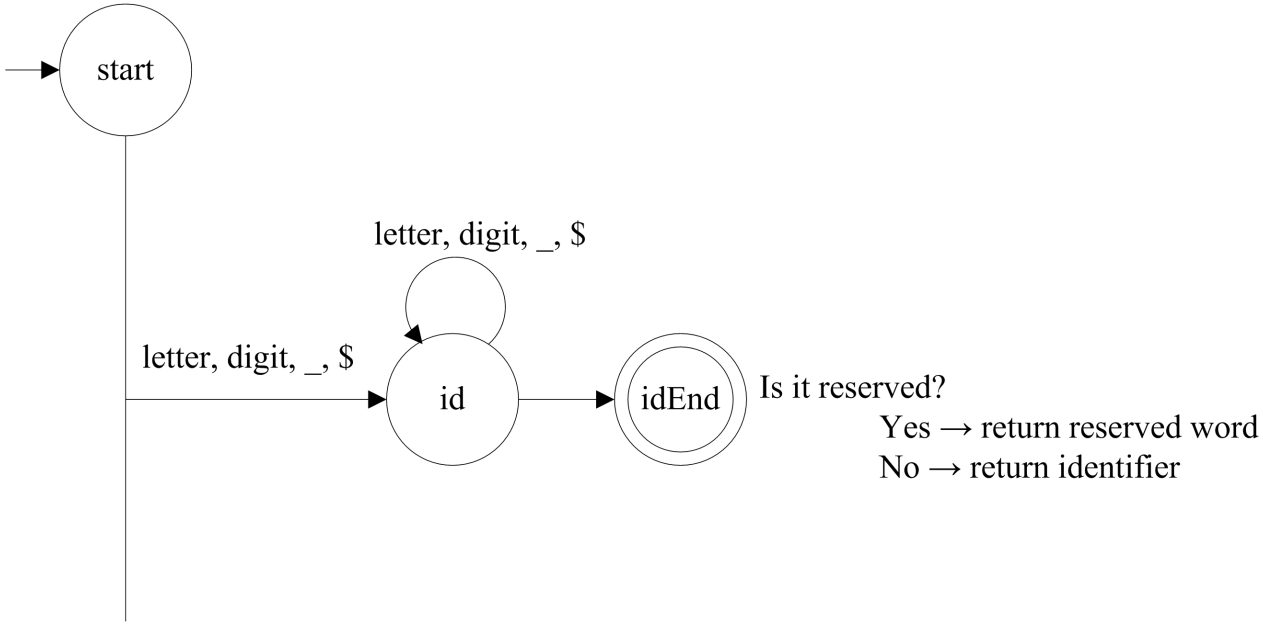
Scanning Tokens

```
...
else if (ch == 'n') {
    buffer.append(ch);
    nextCh();
    if (ch == 'e') {
        buffer.append(ch);
        nextCh();
        if (ch == 'w') {
            buffer.append(ch);
            nextCh();
            if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch != '$') {
                return new TokenInfo(NEW, line);
            }
        }
    }
    else if (ch == 'u') {
        buffer.append(ch);
        nextCh();
        if (ch == 'l') {
            buffer.append(ch);
            nextCh();
            if (ch == 'l') {
                buffer.append(ch);
                nextCh();
                if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch != '$') {
                    return new TokenInfo(NULL, line);
                }
            }
        }
    }
```

Scanning Tokens

```
    while (isLetter(ch) || isDigit(ch) ||
            ch == '_' || ch == '$') {
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(), line);
}
else ...
```

A better approach for recognizing reserved words



Scanning Tokens

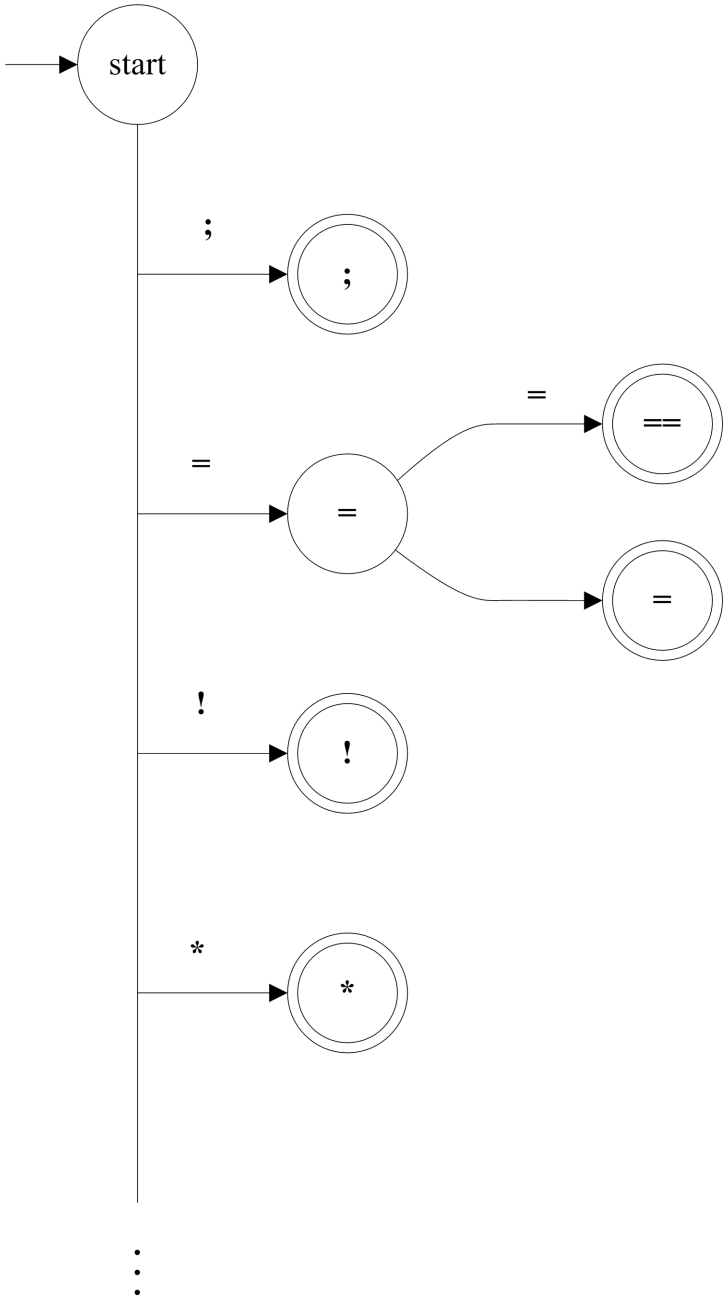
```
    if (isLetter(ch) || ch == '_' || ch == '$') {
        buffer = new StringBuffer();
        while (isLetter(ch) || isDigit(ch) ||
               ch == '_' || ch == '$'){
            buffer.append(ch);
            nextCh();
        }
        String identifier = buffer.toString();
        if (reserved.containsKey(identifier)) {
            return new TokenInfo(reserved.get(identifier), line);
        }
        else {
            return new TokenInfo(IDENTIFIER, identifier, line);
        }
    }
```

The above approach relies on a map (hash table), `reserved`, mapping reserved identifiers to their representations:

```
reserved = new Hashtable<String, Integer>();
reserved.put("abstract", ABSTRACT);
reserved.put("boolean", BOOLEAN);
reserved.put("char", CHAR);
...
reserved.put("while", WHILE);
```

Scanning Tokens

A state transition diagram for separators and operators and the corresponding code

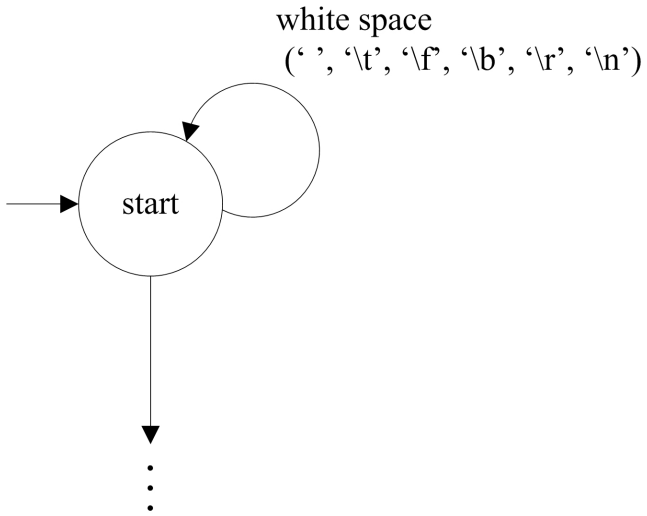


Scanning Tokens

```
switch (ch) {
...
case ';':
    nextCh();
    return new TokenInfo(SEMI, line);
case '=':
    nextCh();
    if (ch == '=') {
        nextCh();
        return new TokenInfo(EQUAL, line);
    }
    else {
        return new TokenInfo(ASSIGN, line);
    }
case '!':
    nextCh();
    return new TokenInfo(LNOT, line);
case '*':
    nextCh();
    return new TokenInfo(STAR, line);
...
}
```

Scanning Tokens

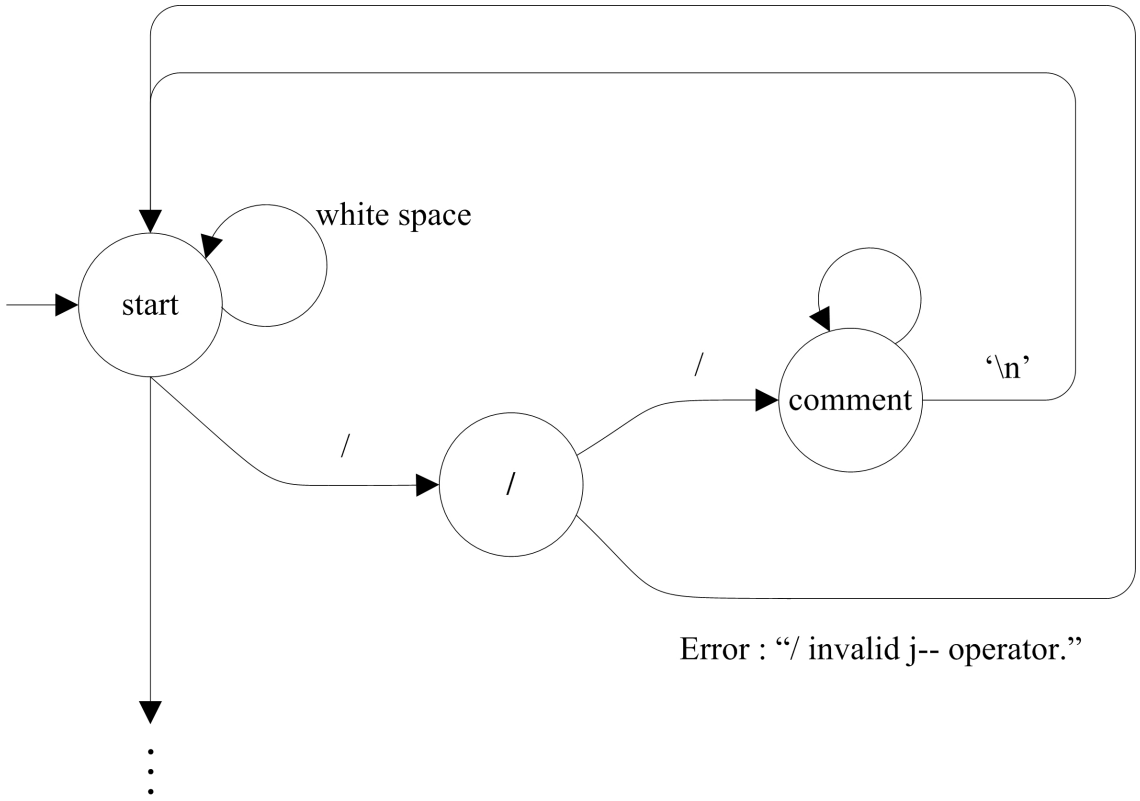
A state transition diagram for whitespace and the corresponding code




```
while (isWhitespace(ch)) {
    nextCh();
}
```

Scanning Tokens

A state transition diagram for comments and the corresponding code



Scanning Tokens

```
boolean moreWhiteSpace = true;
while (moreWhiteSpace) {
    while (isWhitespace(ch)) {
        nextCh();
    }
    if (ch == '/') {
        nextCh();
        if (ch == '/') {
            // CharReader maps all new lines to '\n'
            while (ch != '\n' && ch != EOFCH) {
                nextCh();
            }
        }
        else {
            reportScannerError("Operator / is not supported in j--.");
        }
    }
    else {
        moreWhiteSpace = false;
    }
}
```

Regular Expresssions

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression defines a language of strings over an alphabet, and may take one of the following forms

1. If a is in our alphabet, then the regular expression a describes the language $L(a)$ consisting of the string a
2. If r and s are regular expressions then their concatenation rs is also a regular expression describing the language $L(rs)$ of all possible strings obtained by concatenating a string in the language described by r, to a string in the language described by s
3. If r and s are regular expressions then the alternation $r|s$ is also a regular expression describing the language $L(r|s)$ consisting of all strings described by either r or s
4. If r is a regular expression, the repetition (aka the Kleene closure) $r*$ is also a regular expression describing the language $L(r*)$ consisting of strings obtained by concatenating zero or more instances of strings described by r together
5. ϵ is a regular expression describing the language containing only the empty string
6. If r is a regular expression, then (r) is also a regular expression denoting the same language

Regular Expresssions

For example, given an alphabet $\{a, b\}$

1. $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
2. $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
3. $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab

As another example, in a programming language such as Java

1. Reserved words may be described as `abstract | boolean | char | ... | while`
2. Operators may be described as `= | == | > | ... | *`
3. Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language

A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

  $$m(r, a) = s \text{ where } r, s \in S, a \in \Sigma$$

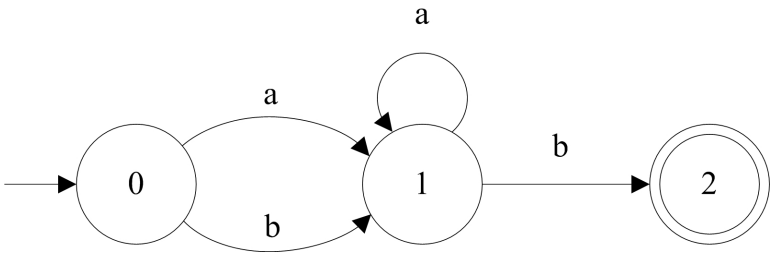
  read as, "if one is in state r, and the next input symbol is a, scan the a and move into state s
- $F \in S$ is a set of final states

Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$ that describes the language consisting of all strings starting with either an a or a b, followed by zero or more a's, and ending with a b

An FSA F that recognizes the language is $F = (\Sigma, S, s_0, M, F)$ where $\Sigma = \{a, b\}, S = \{0, 1, 2\}, s_0 = 0, M = \{m(0, a) = 1, m(0, b) = 1, m(1, a) = 1, m(1, b) = 2\}, F = \{2\}$

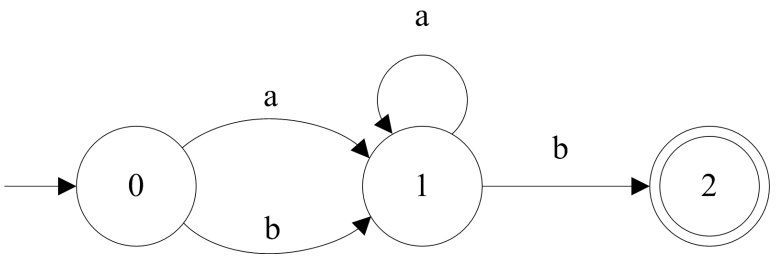
The corresponding transition diagram is shown below



An FSA recognizes strings in the same way that state transition diagrams do

For the previous FSA, given the input sentence $baaab$ and beginning in the start state 0, the following moves are prescribed

- $m(0, b) = 1 \implies$ in state 0 we scan a b and go into state 1
- $m(1, a) = 1 \implies$ in state 1 we scan an a and go back into state 1
- $m(1, a) = 1 \implies$ in state 1 we scan an a and go back into state 1 (again)
- $m(1, a) = 1 \implies$ in state 1 we scan an a and go back into state 1 (again)
- $m(1, b) = 2 \implies$ finally, in state 1 we scan a b and go into the final state 2



Non-deterministic (NFA) Versus Deterministic Finite State Automata (DFA)

A deterministic finite state automaton (DFA) is an automaton without ϵ-moves, and there is a unique move from any state on an input symbol a, ie, there cannot be two moves $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

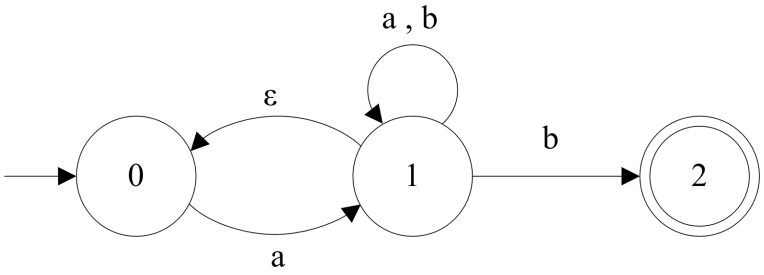
A non-deterministic finite state automaton (NFA) is an automaton that allows either of the following

- More than one move from the same state, on the same input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$, which says we can move from state r to state s without scanning any input symbols

Non-deterministic (NFA) Versus Deterministic Finite State Automata (DFA)

For example, an NFA that recognizes all strings of a's and b's that begin with an a and end with a b is $N = (\Sigma, S, s_0, M, F)$ where $\Sigma = \{a, b\}$, $S = \{0, 1, 2\}$, $s_0 = 0$, $M = \{m(0, a) = 1, m(1, a) = 1, m(1, b) = 1, m(1, \epsilon) = 0, m(1, b) = 2\}$ and, $F = \{2\}$

The corresponding transition diagram is shown below

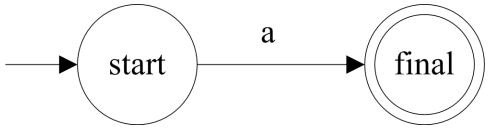


An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

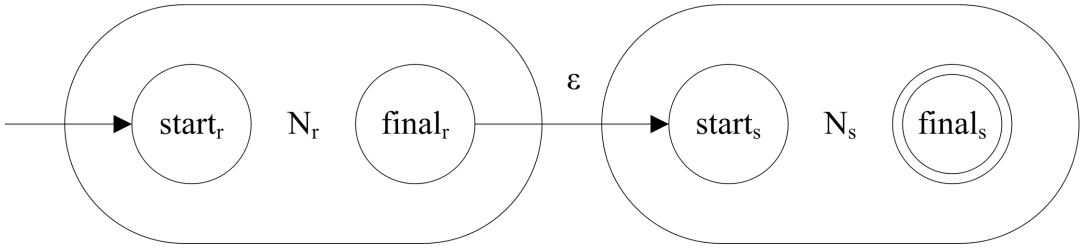
Regular Expresssions to NFA

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

(Rule 1) If the regular expression r takes the form of an input symbol, a, then the NFA that recognizes it has two states: a start state and a final state, and a move on symbol a from the start state to the final state

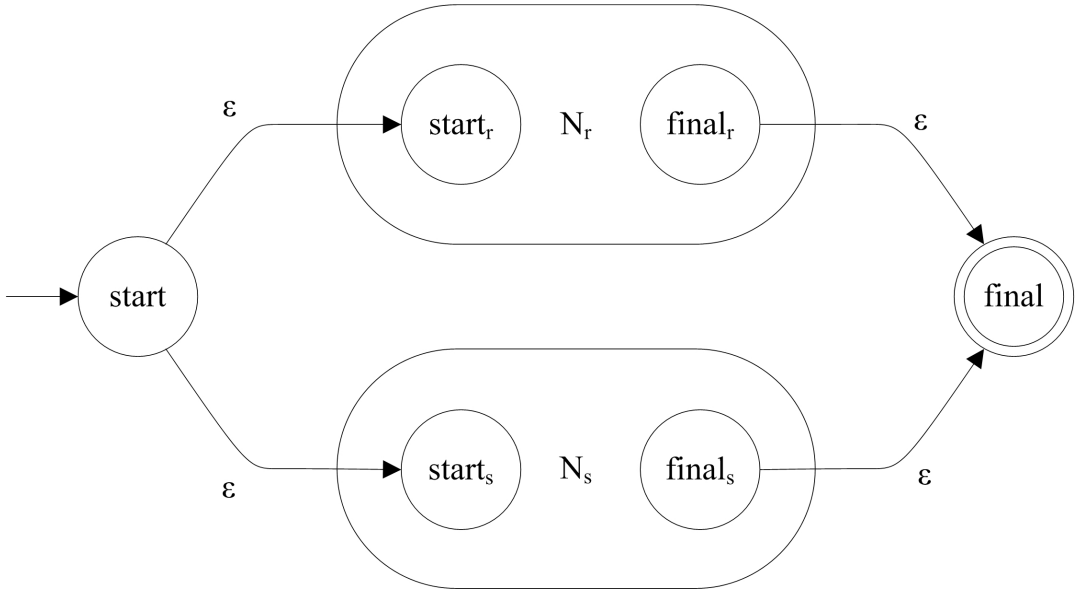


(Rule 2) If N_r and N_s are NFA recognizing the languages described by the regular expressions r and s respectively, then we can create a new NFA recognizing the language described by rs as follows: we define an ϵ-move from the final state of N_r to the start state of N_s, then choose the start state of N_r to be our new start state, and the final state of N_s to be our new final state



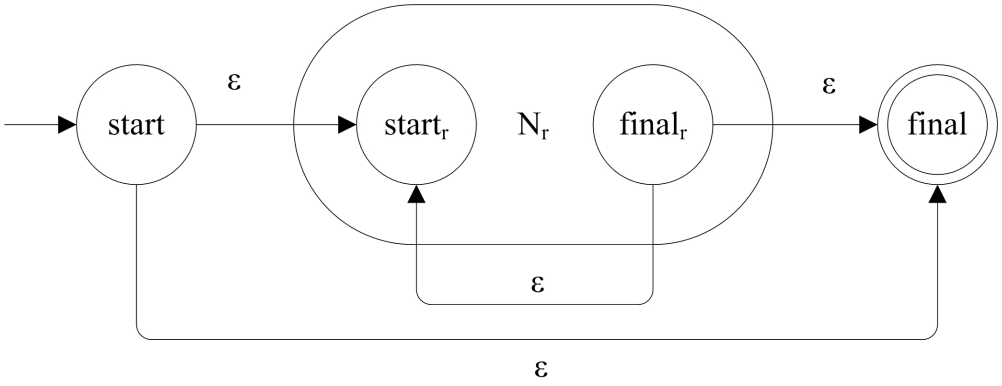
Regular Expresssions to NFA

(Rule 3) If N_r and N_s are NFA recognizing the languages described by the regular expressions r and s respectively, then we can create a new NFA recognizing the language described by $r|s$ as follows: we define a new start state, having ϵ-moves to each of the start states of N_r and N_s, and we define a new final state and add ϵ-moves from each of the final states of N_r and N_s to this state

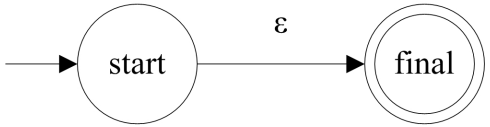


Regular Expresssions to NFA

(Rule 4) If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows: we add an ϵ-move from N_r's final state back to its start state, define a new start state and a new final state, add ϵ-moves from the new start state to both N_r's start state and the new final state, and define an ϵ-move from N_r's final state to the new final state



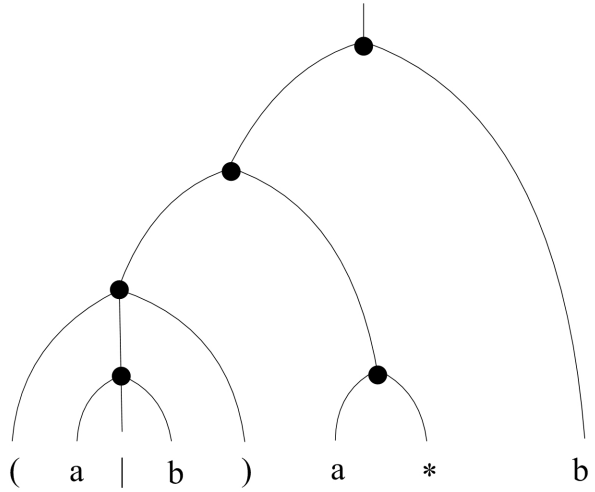
(Rule 5) If r is ϵ then we just need an ϵ-move from the start state to the final state



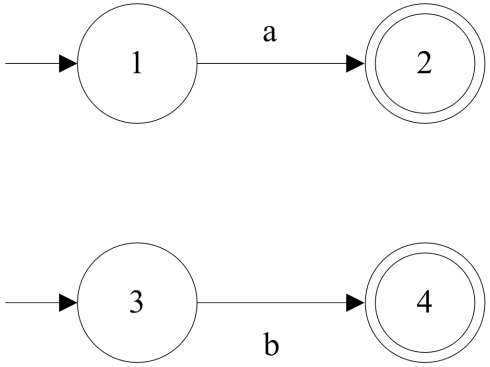
(Rule 6) If N_r is our NFA recognizing the language described by r, then N_r also recognizes the language described by (r)

Regular Expresssions to NFA

As an example, let's construct an NFA for the regular expression $(a|b)a*b$, which has the following syntactic structure

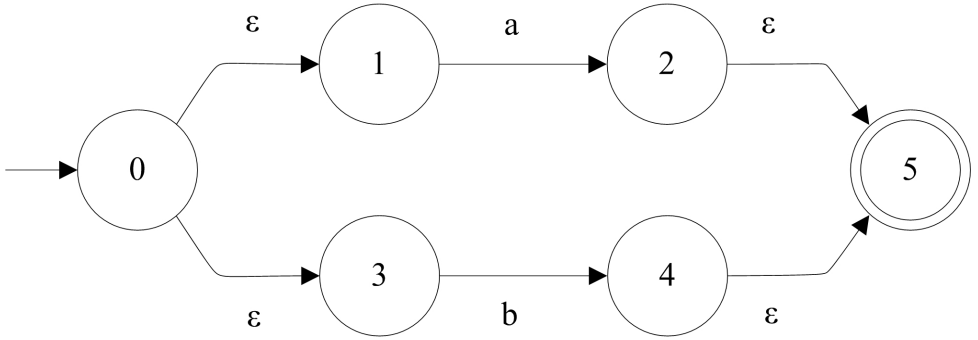


We start with the first a and b; the automata recognizing these are easy enough to construct using Rule 1



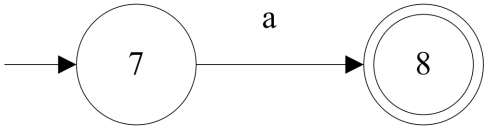
Regular Expresssions to NFA

We then put them together using Rule 3 to produce an NFA recognizing $a|b$

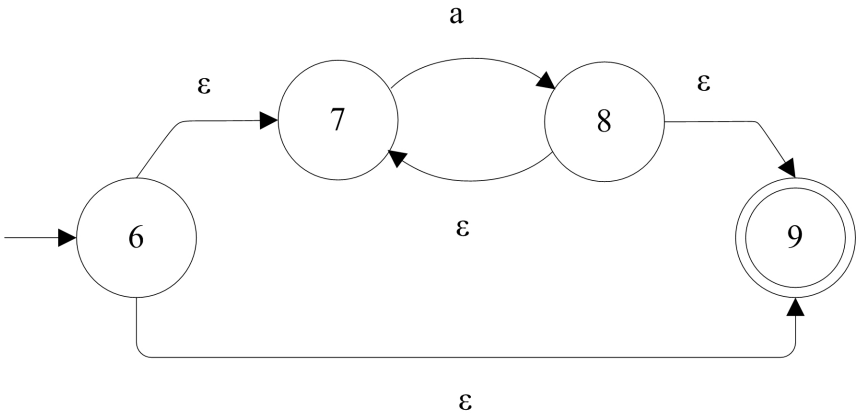


The NFA recognizing $(a|b)$ is the same as that recognizing $a|b$, by Rule 6

An NFA recognizing the second instance of a is simple enough, by Rule 1 again

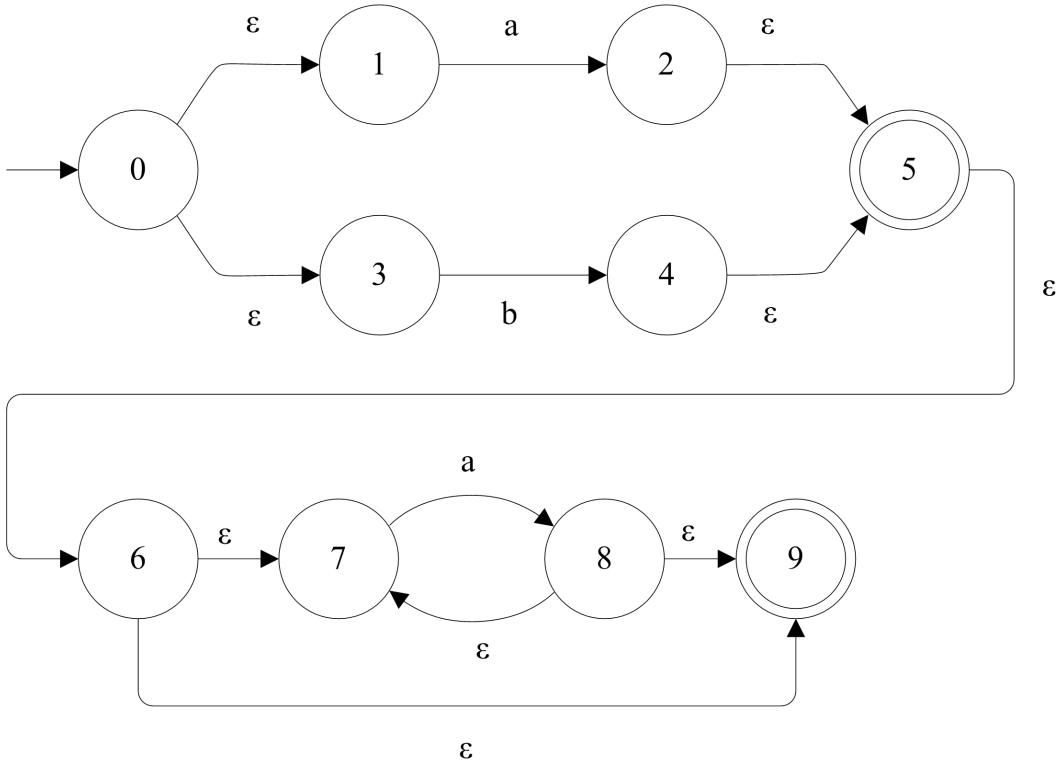


The NFA recognizing $a*$ can be constructed from the NFA for a, by applying Rule 4



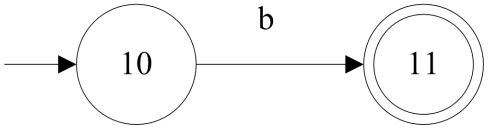
Regular Expresssions to NFA

We then apply Rule 2 to construct an NFA recognizing the concatenation $(a|b)a*$

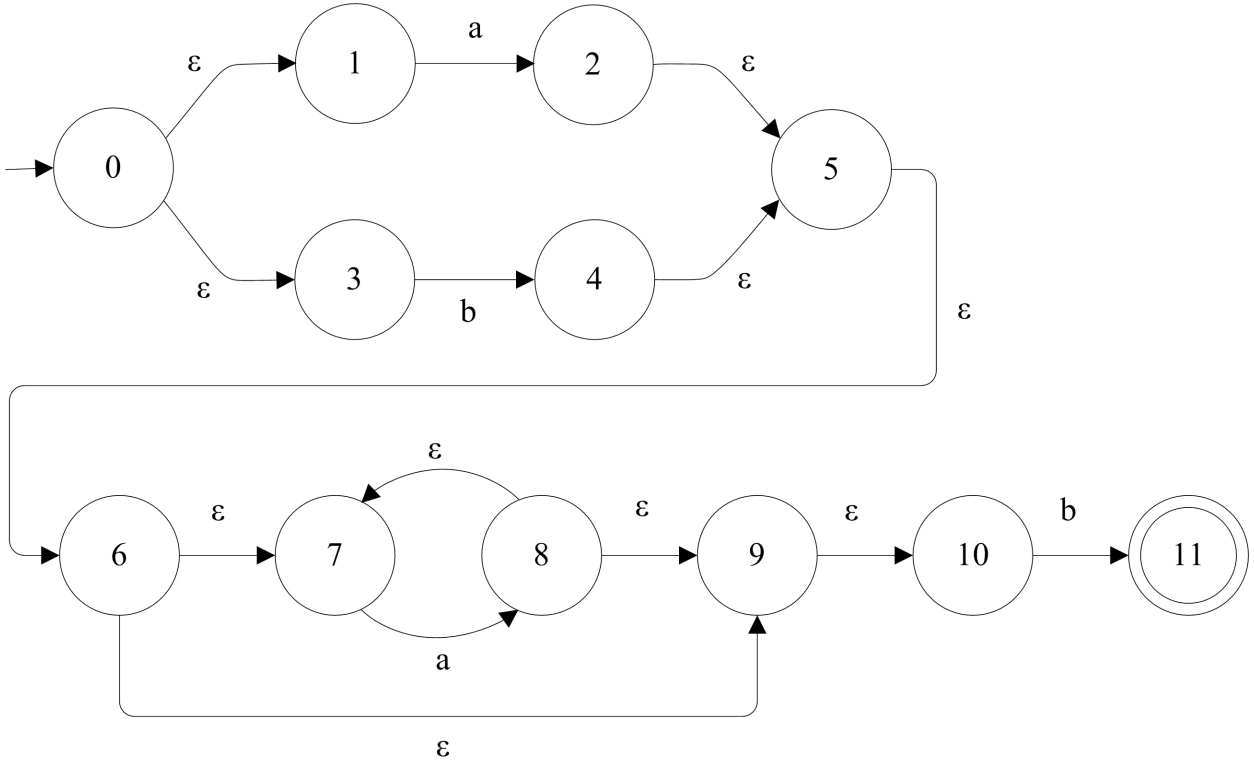


Regular Expresssions to NFA

An NFA recognizing the second instance of b is simple enough, by Rule 1 again



Finally, we can apply Rule 2 again to produce an NFA recognizing $(a|b)a*b$



NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

The DFA that we construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input

The computation of all states reachable from a given state s based on ϵ-moves alone is called taking the ϵ-closure of that state

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie, ϵ-closure(s) $= \{s\} \cup \{r \in S|$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes S and all states reachable from any state $s \in S$ using ϵ-moves alone

NFA to DFA

Algorithm ϵ-closure(S) for a Set of States S

Input: a set of states, S
Output: ϵ-closure(S)

```
Stack P.addAll(S) // a stack containing all states in S
Set C.addAll(S) // the closure initially contains the states in S
while ! P.empty() do
    s ← P.pop()
    for r in m(s, ε) do
        // m(s, ε) is a set of states
        if r ∉ C then
            P.push(r)
            C.add(r)
        end if
    end for
end while
return C
```

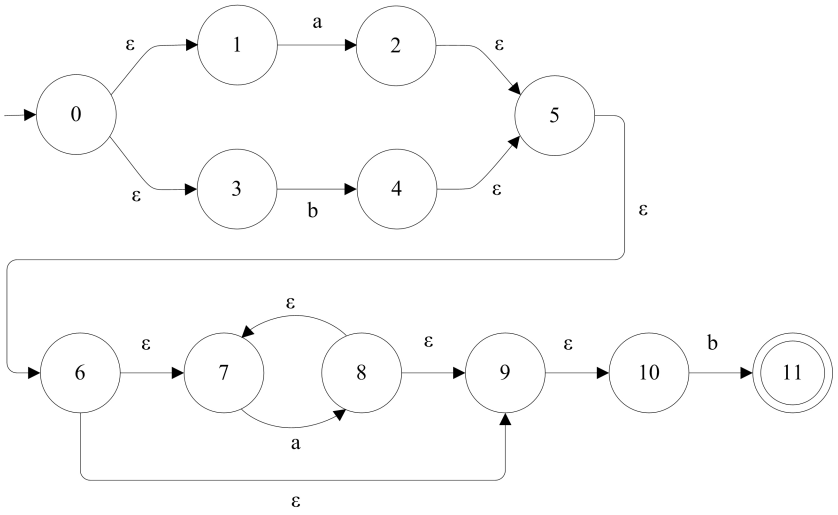
Algorithm ϵ-closure(s) for a State s

Input: a state, s
Output: ϵ-closure(s)

```
Set S.add(s) // S = {s}
return ε-closure(S)
```

NFA to DFA

NFA for the regular expression $(a|b)a*b$ from before



In the corresponding DFA, from the start state s_0, and scanning the symbol a, we want to go into a state that reflects all the states we could be in after scanning an a in the NFA: 2, and then (via ϵ-moves) 5, 6, 7, 9 and 10

$$m(s_0, a) = s_1, \text{ where}$$
$$s_1 = \epsilon\text{-closure}(2) = \{2, 5, 6, 7, 9, 10\}$$

Similarly, scanning a symbol b in state s_0, we get

$$m(s_0, b) = s_2, \text{where}$$
$$s_2 = \epsilon\text{-closure}(4) = \{4, 5, 6, 7, 9, 10\}$$

NFA to DFA

From state s_1, scanning an a, we have to consider where we could have gone from the states $\{2, 5, 6, 7, 9, 10\}$ in the NFA; from state 7, scanning an a, we go into state 8, and then (by ϵ-moves) 7, 9, and 10

$$m(s_1, a) = s_3, \text{where}$$
$$s_3 = \epsilon\text{-closure}(8) = \{7, 8, 9, 10\}$$

Now, from state s_1, scanning b, we have

$$m(s_1, b) = s_4, \text{where}$$
$$s_4 = \epsilon\text{-closure}(11) = \{11\}$$

NFA to DFA

From state s_2, scanning an a takes us into a state reflecting 8, and then (by ϵ-moves) 7, 9 and 10, generating state $\{7, 8, 9, 10\}$, which is s_3

Scanning a b from state s_2 takes us into a state reflecting 11, generating state $\{11\}$, which is s_4

$$m(s_2, a) = s_3$$
$$m(s_2, b) = s_4$$

From state s_3, scanning an a takes us back into s_3, and scanning a b takes us into s_4

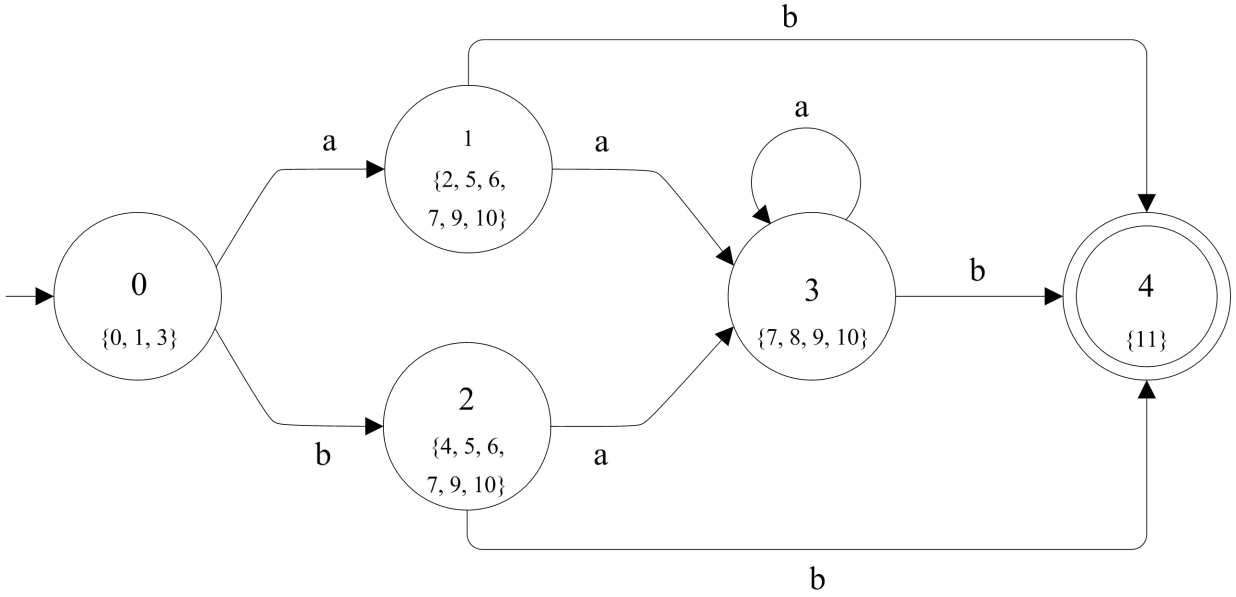
$$m(s_3, a) = s_3$$
$$m(s_3, b) = s_4$$

There are no moves at all out of state s_4, so we have found all of our transitions and all of our states

Any state reflecting a final state in the NFA is a final state in the DFA; in our example, only s_4 is a final state because it contains (the final) state 11 from the original NFA

NFA to DFA

The DFA derived from our NFA for the regular expression $(a|b)a*b$ is shown below



NFA to DFA

Algorithm NFA to DFA Construction

Input: an NFA, $N = (\Sigma, S, s_0, M, F)$
Output: DFA, $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

```
Set s_D0 ← ε-closure(s_0)
Set S_D.add(s_D0)
Moves M_D
Stack stk.push(s_D0)
i ← 0
while ! stk.empty() do
    t ← stk.pop()
    for a in Σ do
        s_Di+1 ← ε-closure(m(t, a));
        if s_Di+1 ≠ {} then
            if s_Di+1 ∉ S_D then
                S_D.add(s_Di+1) // We have a new state
                stk.push(s_Di+1)
                i ← i + 1
                M_D.add(m_D(t, a) = i)
            else if ∃j, s_j ∈ S_D and s_Di+1 = s_j then
                M_D.add(m_D(t, a) = j) // In the case the state already exists
            end if
        end if
    end for
end while
```

Algorithm NFA to DFA Construction (contd.)

Set F_D
for s_D in S_D **do**
  for s in s_D **do**
    if $s \in F$ **then**
      F_D.add(s_D)
    end if
  end for
end for
return $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

A Minimal DFA

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

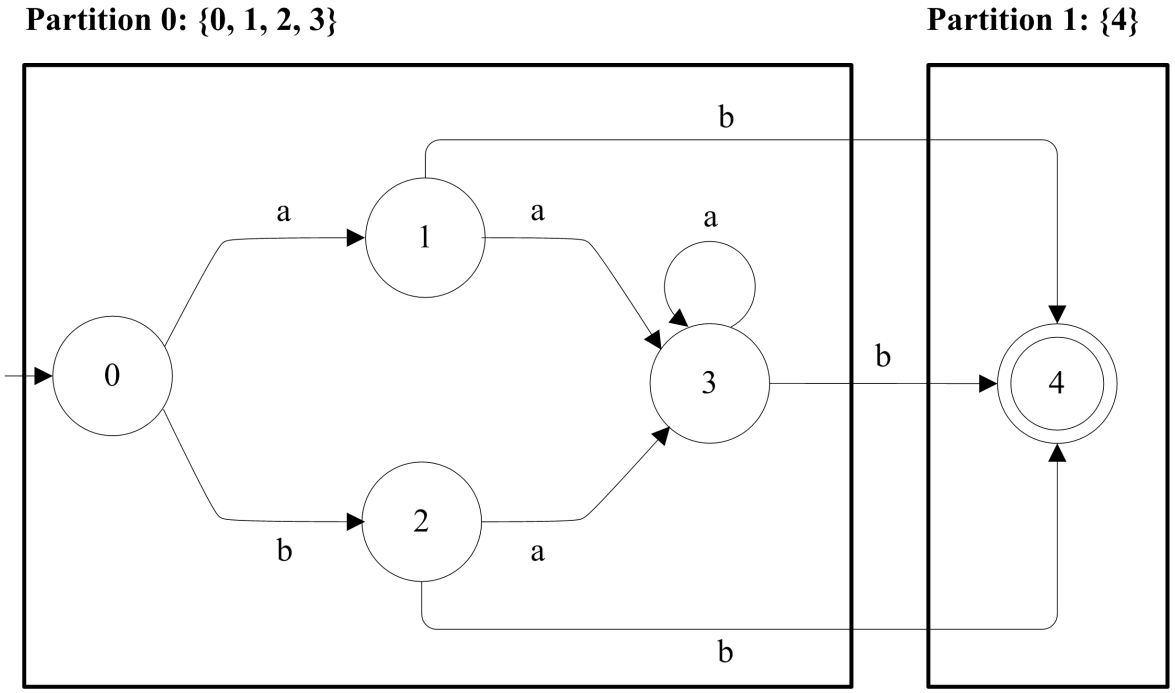
We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

A Minimal DFA

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



The two states in this new DFA consist of the start state, $\{0, 1, 2, 3\}$ and the final state $\{4\}$

We must make sure that from a particular partition, each input symbol must move us to an identical partition

A Minimal DFA

Beginning in any state in the partition $\{0, 1, 2, 3\}$, an a takes us to one of the states in $\{0, 1, 2, 3\}$

$$m(0, a) = 1$$
$$m(1, a) = 3$$
$$m(2, a) = 3$$
$$m(3, a) = 3$$

So, our partition $\{0, 1, 2, 3\}$ is fine so far as moves on the symbol a are concerned

For the symbol b,

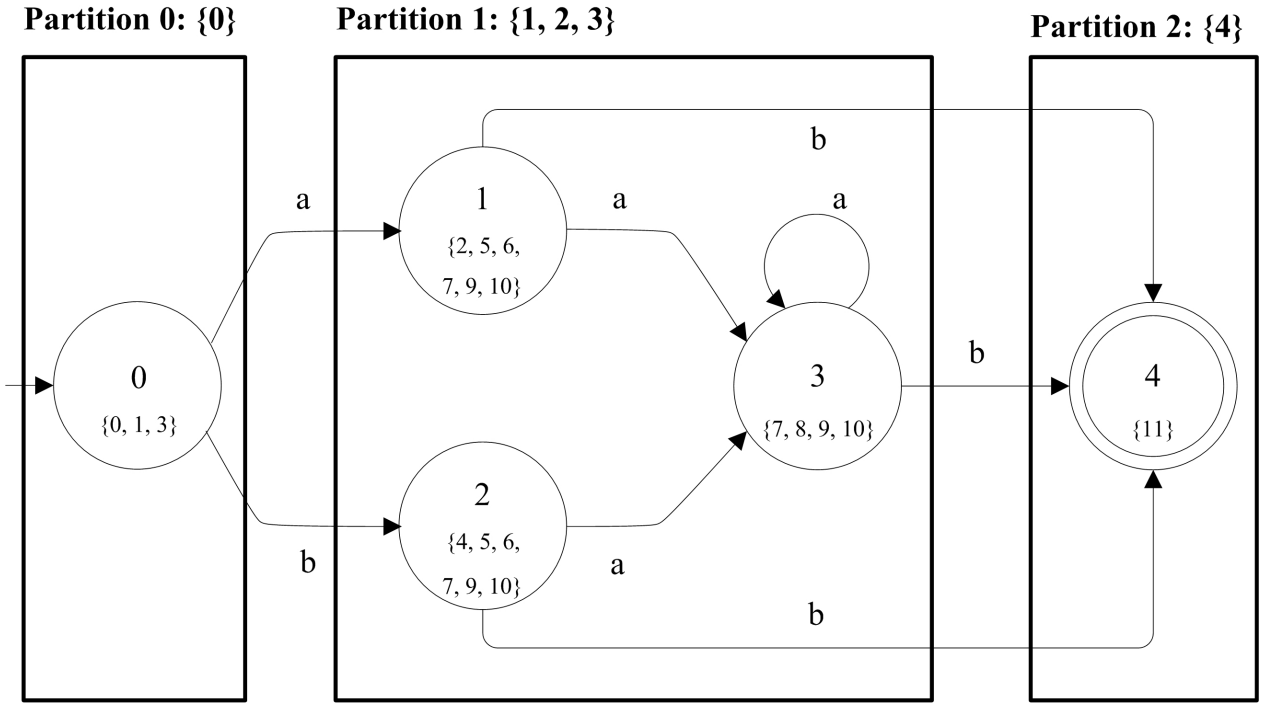
$$m(0, b) = 2$$
$$m(1, b) = 4$$
$$m(2, b) = 4$$
$$m(3, b) = 4$$

So we must split the partition $\{0, 1, 2, 3\}$ into two new partitions, $\{0\}$ and $\{1, 2, 3\}$

If we are in state s, and for an input symbol a in our alphabet there is no defined move $m(s, a) = t$, we invent a special dead state d, so that we can say $m(s, a) = d$

A Minimal DFA

We are left with a partition into three sets: $\{0\}$, $\{1, 2, 3\}$ and $\{4\}$, as shown below



A Minimal DFA

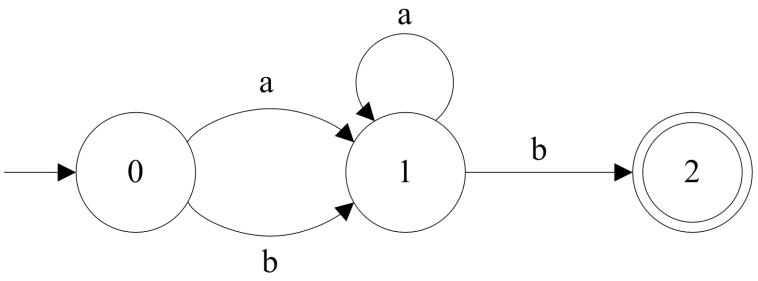
We need not worry about $\{0\}$ and $\{4\}$ as they contain just one state and so correspond to (those) states in the original machine

We consider $\{1, 2, 3\}$ to see if it is necessary to split it

$$m(1, a) = 3$$
$$m(2, a) = 3$$
$$m(3, a) = 3$$

$$m(1, b) = 4$$
$$m(2, b) = 4$$
$$m(3, b) = 4$$

Thus, there is no further state splitting to be done, and we are left with the following smaller DFA



A Minimal DFA

Algorithm Minimizing a DFA

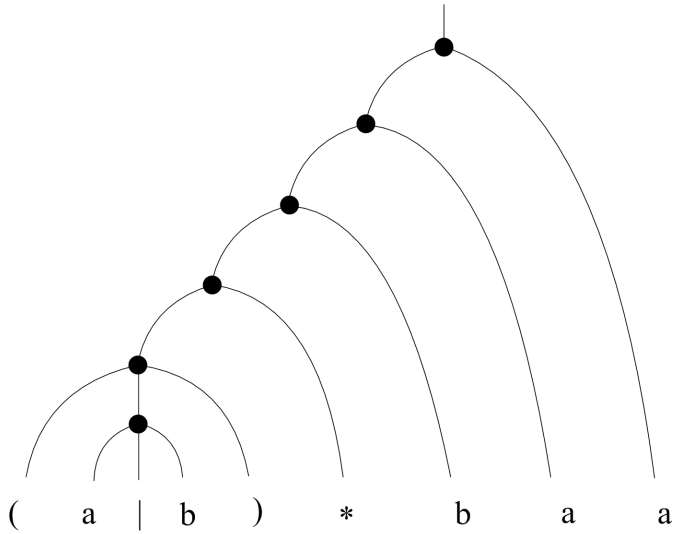
Input: a DFA, $D = (\Sigma, S, s_0, M, F)$
Output: a partition of S

```
Set partition ← {S − F, F} // start with two sets: the non-final states and the final states
// Splitting the states
while splitting occurs do
    for Set set in partition do
        if set.size() > 1 then
            for Symbol a in Σ do
                // Determine if moves from this 'state' force a split
                State s ← a state chosen from set
                targetSet ← the set in the partition containing m(s, a)
                Set set1 ← {states s from set S, such that m(s, a) ∈ targetSet}
                Set set2 ← {states s from set S, such that m(s, a) ∉ targetSet}
                if set2 ≠ {} then
                    // Yes, split the states.
                    replace set in partition by set1 and set2 and break out of the for-loop
                    to continue with the next set in the partition
                end if
            end for
        end if
    end for
end while
```

A Minimal DFA

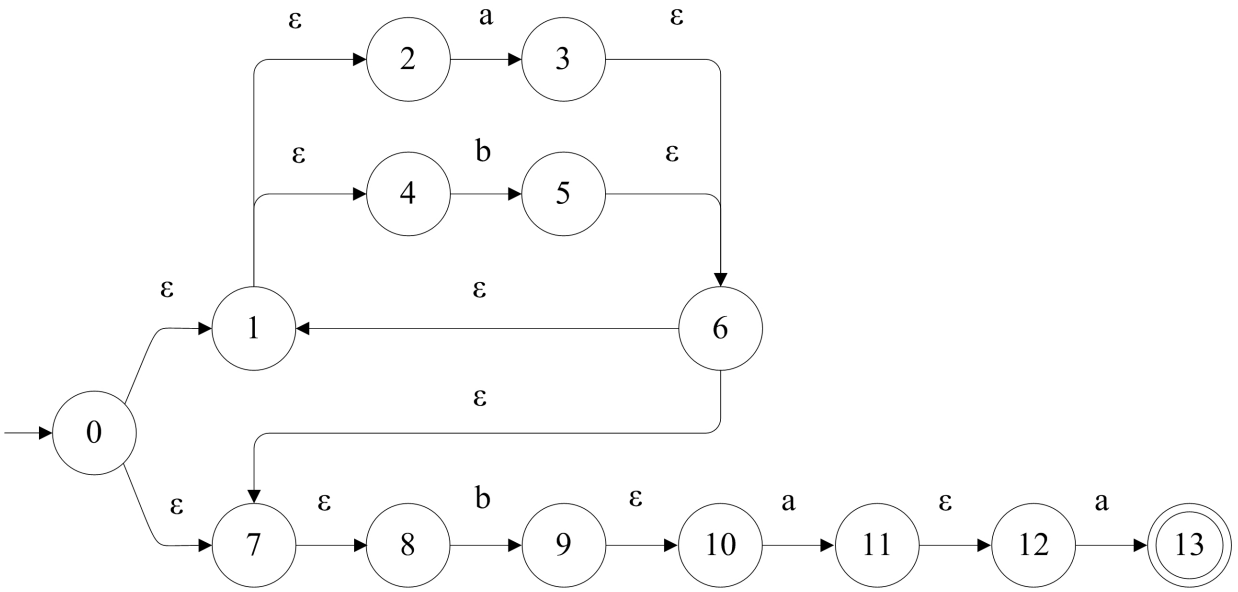
Let us run through another example, starting from a regular expression, producing an NFA, then a DFA, and finally a minimal DFA

Consider the regular expression $(a|b) * baa$ having the following syntactic structure



A Minimal DFA

We apply the Thompson's construction procedure to produce the NFA shown below



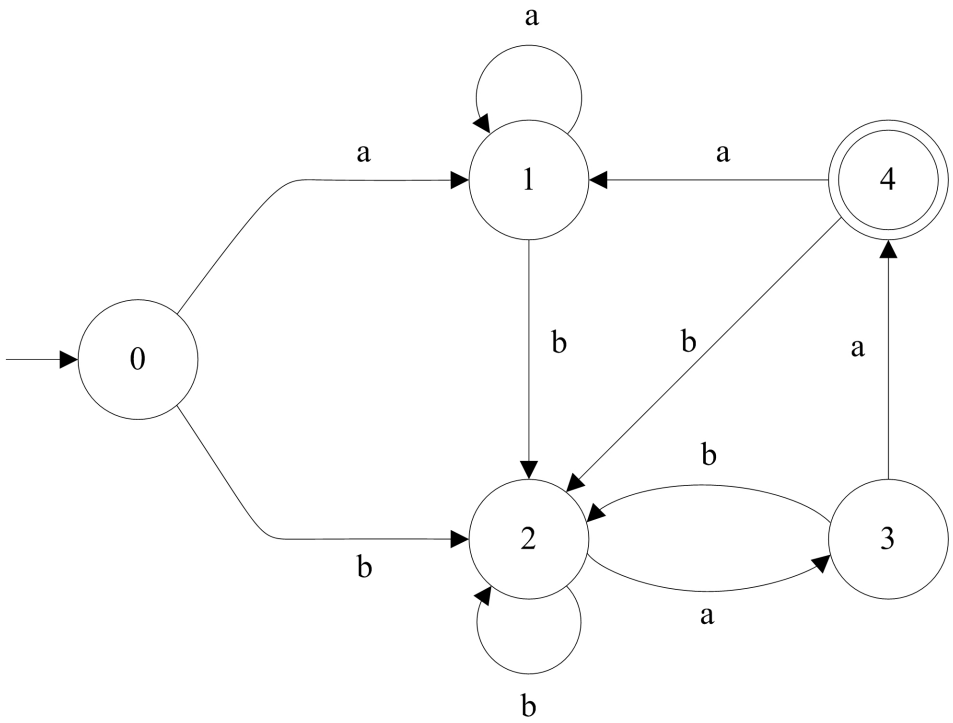
A Minimal DFA

Using the powerset construction method, we derive a DFA having the following states

$$
\begin{aligned}
&s_0 : \{0, 1, 2, 4, 7, 8\} \\
&m(s_0, a) : \{1, 2, 3, 4, 6, 7, 8\} = s_1 \\
&m(s_0, b) : \{1, 2, 4, 5, 6, 7, 8, 9, 10\} = s_2 \\
&m(s_1, a) : \{1, 2, 3, 4, 6, 7, 8\} = s_1 \\
&m(s_1, b) : \{1, 2, 4, 5, 6, 7, 8, 9, 10\} = s_2 \\
&m(s_2, a) : \{1, 2, 3, 4, 6, 7, 8, 11, 12\} = s_3 \\
&m(s_2, b) : \{1, 2, 4, 5, 6, 7, 8, 9, 10\} = s_2 \\
&m(s_3, a) : \{1, 2, 3, 4, 6, 7, 8, 13\} = s_4 \\
&m(s_3, b) : \{1, 2, 4, 5, 6, 7, 8, 9, 10\} = s_2 \\
&m(s_4, a) : \{1, 2, 3, 4, 6, 7, 8\} = s_1 \\
&m(s_4, b) : \{1, 2, 4, 5, 6, 7, 8, 9, 10\} = s_2
\end{aligned}
$$

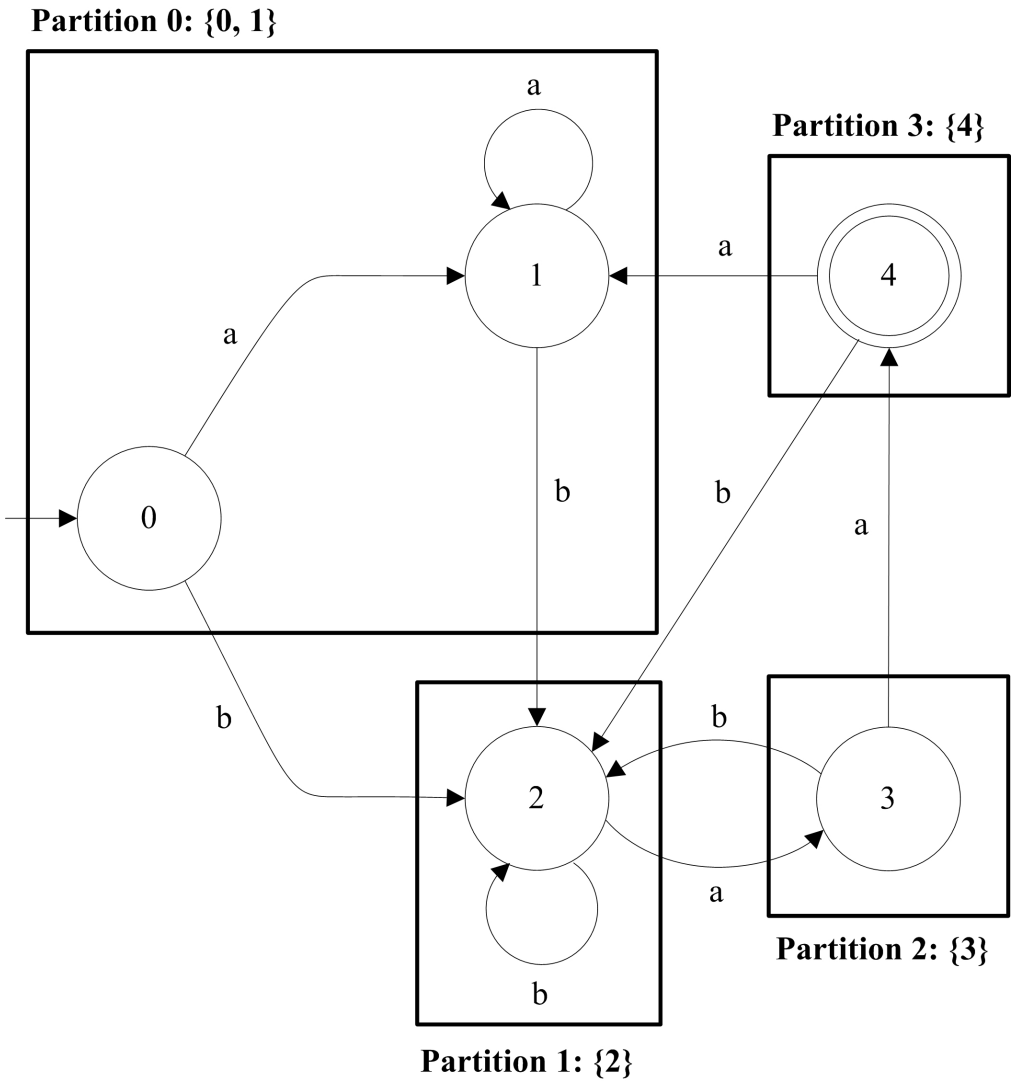
A Minimal DFA

The DFA itself is shown below



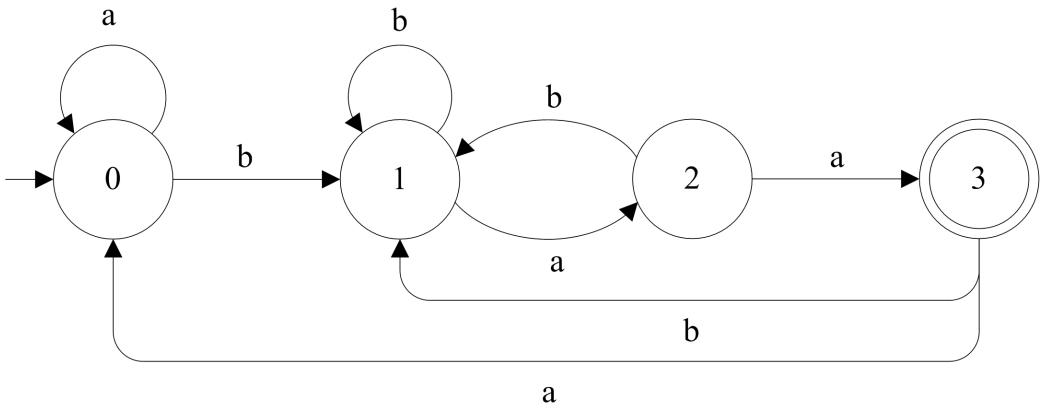
A Minimal DFA

Finally, we use partitioning to produce the minimal DFA shown below



A Minimal DFA

We re-number the states to produce the equivalent DFA shown below



JavaCC: a Tool for Generating Scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating lexical analyzers from regular expressions and parsers from context-free grammars

A lexical grammar specification consists a set of regular expressions and a set of lexical states; from any particular state, only certain regular expressions may be matched in scanning the input

There is a standard `DEFAULT` state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. `SKIP`: throws away the matched string
2. `MORE`: continues to the next state, taking the matched string along
3. `TOKEN`: creates a token from the matched string and returns it to the parser (or any caller)
4. `SPECIAL_TOKEN`: creates a special token that does not participate in the parsing

JavaCC: a Tool for Generating Scanners

For example, a SKIP can be used for ignoring white space

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

We can deal with single-line comments with the following regular expressions

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT>
MORE: { < ~[] > }
```

An alternative regular expression dealing with single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Reserved words and symbols are specified by simply spelling them out; for example

```
TOKEN: {
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
...

| < COMMA: "," >
| < DOT: "." >
}
```

JavaCC: a Tool for Generating Scanners

A token for scanning identifiers

```
TOKEN: {
  < IDENTIFIER: (<LETTER>|"_"|"$") (<LETTER>|<DIGIT>|"_"|"$")* >
| < #LETTER: ["a"-"z","A"-"Z"] >
| < #DIGIT: ["0"-"9"] >
}
```

A token for scanning literals

```
TOKEN: {
  < INT_LITERAL: ("0" | <NON_ZERO_DIGIT> (<DIGIT>)*) >
| < #NON_ZERO_DIGIT: ["1"-"9"] >
| < CHAR_LITERAL: "'" (<ESC> | ~["'","\\","\n","\r"]) "'" >
| < STRING_LITERAL: "\"" (<ESC> | ~["\"","\\","\n","\r"])* "\"" >
| < #ESC: "\\" ["n","t","b","r","f","\\","'","\""] >
}
```

JavaCC takes a specification of the lexical syntax and produces several Java files, one of which is `TokenManager.java`, a program that implements a state machine; this is our scanner

The entire lexical grammar for *j--* is specified in `$j/j--/lexicalgrammar`