Overview of Scanning   Chapter 2

- The function of a scanner, called also lexical analyzer, is to:
  - Read characters from the source file
  - Group input characters into meaningful units, called **tokens**
- The scanner takes care of other things as well:
  - Removal of comments and white space
  - Keeping track of current line number and character position
    - Required for reporting error messages
  - Case conversions of identifiers and keywords
    - Simplifies searching if the language is not case-sensitive
  - Interpretation of compiler directives
    - Flags are internally set to direct code generation
  - Communication with the symbol or literal table
    - Identifiers can be entered in the symbol table
    - String literals can be entered in the literal table

Tokens and Lexemes

- Consider the following statement:

  if distance >= rate * (time1 – time0) **then** distance := maxdist ;

  - Contains 5 identifiers: distance, rate, time1, time0, maxdist
- For parsing purposes, all identifiers are alike
  - It is enough to tell the parser that the next token is an identifier
  - However, the code generator needs the name of the identifier
- Similarly, for parsing purposes all relational operators are alike
  - The syntactic structure would not change if >= were changed to > or <=
  - However, the code generator needs to know exactly what operator is used
- We make the following distinction:
  - A **token** is a logical entity described as part of the syntax of a language
  - A **lexeme** is a special instance of the token, which is the **string value**
- For the above statement, the scanner should return the following tokens:

  if id relop id * (id – id) then id := id ;

Formal Languages

- An **alphabet** is a finite set of characters, denoted by the Greek symbol Σ (sigma)
  - The alphabet can be the ASCII set, a subset of ASCII, or so ASCII
- A **string** over some alphabet is a sequence of symbols drawn from the alphabet
  - Example: 01001 is string over the alphabet {0,1}
- The empty or **null string** ε is a special string of length zero
  - When ε is concatenated with any string s yields s. That is, $s\,\varepsilon = \varepsilon\, s = s$
- A **formal language** is a set of strings (possibly infinite) over some alphabet
  - Just a set of strings. No specific relationship with a programming language.
  - Example: L = {00, 01, 10, 11} is the set of all 2-character strings over $\Sigma = \{0,1\}$
- The **concatenation** of two languages L_1 and L_2 (sets of strings) is:
  - Obtained by concatenating every string in L_1 with every string in L_2
  - $L = L_1 L_2 = \{s = s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2\}$
  - $L_1 = \{\varepsilon, 0, 00\}$, $L_2 = \{\varepsilon, 1, 11\}$, $L = L_1 L_2 = \{\varepsilon, 1, 11, 0, 01, 011, 00, 001, 0011\}$

Formal Languages – cont'd

- The **exponentiation** of a language is defined as follows:
  - $L^0 = \{\varepsilon\}$ and $L^{i\text{-}1} = L^{i\text{-}1} L$
  - $L = \{0,1\}$, $L^0 = \{\varepsilon\}$, $L^1 = L = \{0,1\}$
  - $L^2 = \{00, 01, 10, 11\}$ and $L^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- The **Kleene closure** of a language L, denoted as L*, is defined as: $L^* = \bigcup_{i=0}^{\infty} L^i$
  - $L = \{0,1\}$, $L^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$
  - $\varepsilon \in L^*$ for any set L
- The **Positive closure** of a language L, denoted as L^+, is defined as: $L^+ = \bigcup_{i=1}^{\infty} L^i$
  - $L = \{0,1\}$, $L^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$
- Examples:
  - Let L = {A, B, …, Z, a, b, …, z} set of all English letters
  - Let D = {0, 1, …, 9} set of all digits
  - $L \cup D$ is the set of letters and digits
  - LD is the set of strings consisting of a letter followed by a digit
  - L^4 is the set of all four-letter strings
  - L* is the set of all strings of letters , including ε
  - $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter

Regular Expression

- Is a notation that denotes a set of strings that follows a certain pattern
  - A regular expression r corresponds to set of strings $L(r)$
  - $L(r)$ is called a **regular set** or a **regular language** and may be infinite
- A regular expression is defined as follows:
  - A basic regular expression **`a`** denotes the set $\{a\}$ where $a \in \Sigma$; $L(\mathbf{a}) = \{a\}$
  - The regular expression ε denotes the set {ε}
    - Technically, regular expression ε is different from string ε
  - If **`r`** and **`s`** are two regular expressions denoting the sets $L(r)$ and $L(s)$ then:
    - **`r|s`** is a regular expression denoting the union set: $L(r) \cup L(s)$
    - **`r s`** is a regular expression denoting the concatenation set: $L(r)\ L(s)$
    - **`r*`** is a regular expression denoting the Kleene closure set: $L(r)^*$
    - **`(r)`** is a regular expression denoting the set $L(r)$
- Regular expressions are of practical interest. They can be used to:
  - Specify the structure of tokens
  - Program a scanner generator

Examples of Regular Expressions

- **0|1** denotes the set {0,1}
- **0*** denotes the set {ε, 0, 00, 000, 0000, …}
- **(0|1)(0|1)** denotes the set {00, 01, 10, 11}
- **(0|1)*** denotes the set {ε, 0, 1, 00, 01, 10, 11, 000, 001, …}
- **0|0*1** denotes the set {0, 1, 01, 001, 0001, …}
- Consider the alphabet Σ = {*a*, *b*, *c*}, the set of all:
  - Strings containing exactly one *b* is represented by: **(a|c)*b(a|c)***
  - Strings containing at most one *b* is represented by: **(a|c)*|(a|c)*b(a|c)***
    - Alternative solution: **(a|c)*(b|ε)(a|c)***
    - There is NO unique answer, but we attempt to find a simple regular expression
  - Strings that contain NO two consecutive *b*'s is represented by:
    - **(*notb*|b *notb*)*(b|ε)** where ***notb* = (a|c)**
    - Equivalent to: **(a|c|b(a|c))*(b|ε) = (a|c|ba|bc)*(b|ε)**
    - Equivalent to: **(b|ε)(a|c|ab|cb)***
- Any finite set of strings is regular and can be represented by: **(s_1|s_2|…|s_k)**

Extensions to Regular Expressions

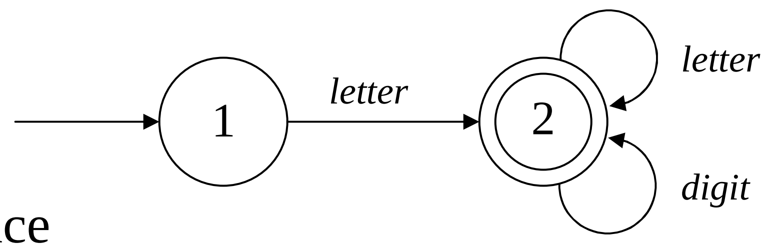
- The standard regular operations are sufficient to describe any regular expression
  - Standard regular operations are: alternation, concatenation, and Kleene closure
- In practice, additional operations are often utilized
- **`r+`** is a regular expression denoting the positive closure set: $L(r)+$
  - **`r+`** is read as **one or more** repetitions of **`r`**, while **`r*`** is **zero or more** repetitions of **`r`**
  - **`r+ = r r*`** and **`r* = r+|`**ε
- **`r?`** is a regular expression denoting the set $L(r) \cup \{\varepsilon\}$. **`r? = r|`**ε
- To specify a range of characters, we will use Lex notation as follows:
  - **`[0-9] = 0|1|2|...|9`**
  - **`[A-Za-z] = A|B|...|Z|a|b|...|z`**
- To exclude characters from the alphabet, we will use Lex notation as follows:
  - **`[^a]`** matches any character in the alphabet except *a*
  - **`[^abc]`** matches any character in the alphabet, which is not *a*, *b*, or *c*
  - **`[^0-9]`** matches any character in the alphabet which is not a digit

Regular Definitions

- We may assign a name to a regular expression to:
  - Use and Reuse the name in other (more complex) regular expressions
  - Enhance the readability of longer regular expressions
- Given the following regular definitions:
  - ***digit*** **= [0-9]**, ***letter*** **= [A-Za-z]**, ***eol*** **= \n**, and ***neol*** **= [^\n]**
  - We can use them to write complex regular expressions:
    - Integer Literal = ***digit+***
    - Fixed-Point Literal = ***digit+* "." *digit+***
    - Floating-Point Literal = ***digit+* "." *digit+*(e|E)(+|-)?*digit+***
    - Identifier = ***letter*(*letter*|*digit*)****
    - Ada Comment = **-- *neol** *eol***
- Not all infinite sets of strings are regular
  - The set $\{a^n b a^n \mid n \geq 0\}$ cannot be described by a regular expression
  - **a*ba*** does not guarantee the same number of *a*'s at the beginning and end

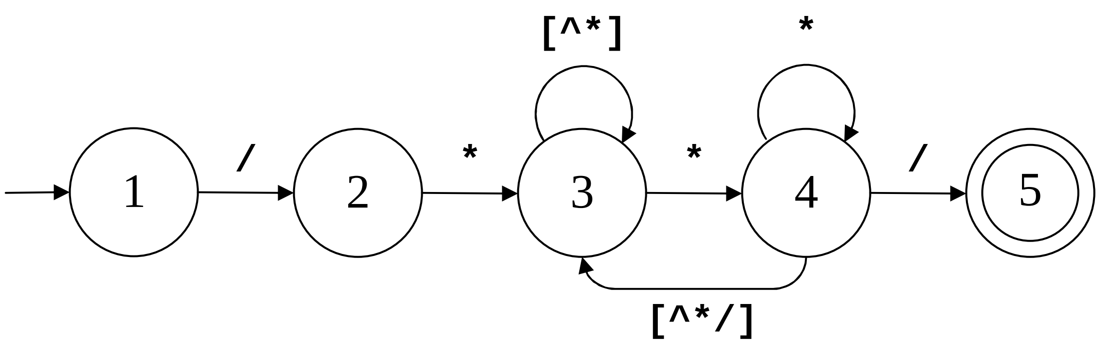
Finite Automata

- Used to recognize the tokens specified by a regular expression
- Can be converted to an algorithm for matching input strings
- A Finite Automaton (FA) consists of:
  - A finite set of **states**
  - A set of **transitions** (or moves) between states
    - The transitions are labeled by characters form the alphabet
  - A special **start state**
  - A set of **final** or **accepting states**
- A finite automaton for `letter(letter|digit)*` is shown below
- We may label a transition with more than one character for convenience
- We start at the start state
- We make a transition if next input character matches label on transition
- If no move is possible, we stop
- If we end in an accepting state then
  - input sequence of characters is valid
- Otherwise, we do not have a valid sequence



Deterministic Finite Automata (DFA)

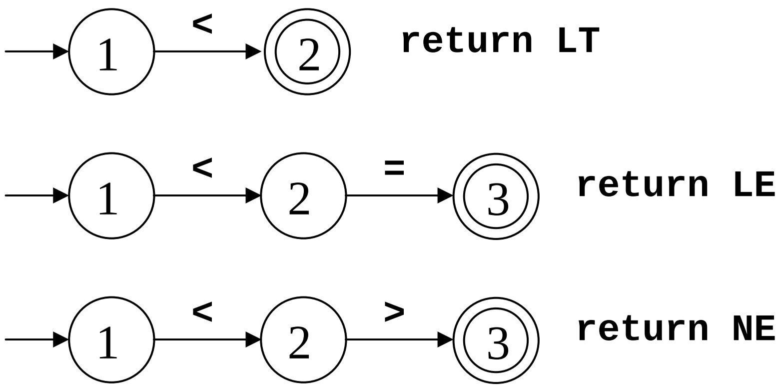
- Has a **unique** transition for every state and input character
- Can be represented by a **transition table T**
  - Table **T** is indexed by state *s* and input character *c*
  - **T**[*s*][*c*] is the next state to visit from state *s* if the input character is *c*
  - **T** can also be described as a **transition function**
  - **T**: $S \times \Sigma \rightarrow S$ maps the pair (*s*, *c*) to *next_s*
- DFA and transition table for a C comment are show below
  - Blank entries in the table represent an **error state**
  - A full transition table will contain one column for each character (may waste space)
  - Characters are combined into **character classes** when treated identically in a DFA

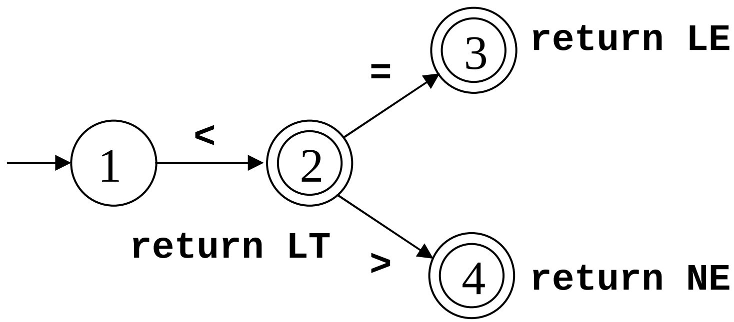


State	/	*	other
1	2		
2		3	
3	3	4	3
4	5	4	3
5			

Combining DFAs

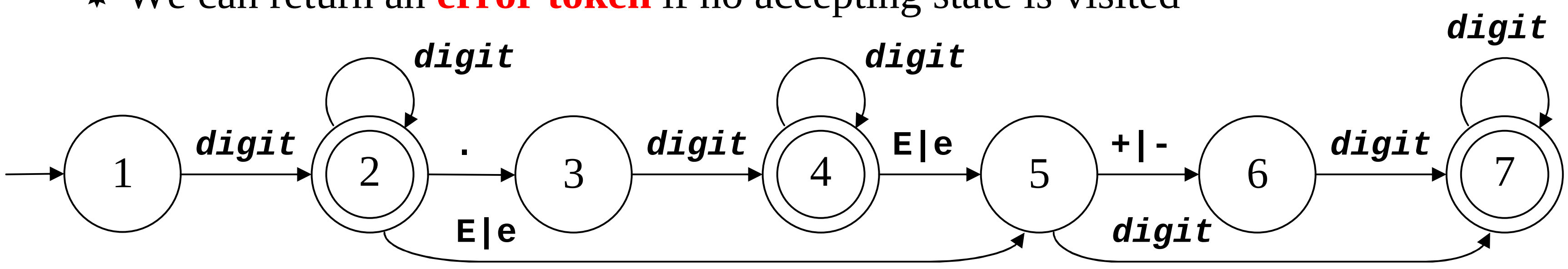
- In a programming language there are many tokens
- Each token is recognized by its own DFA
- We need to combine DFAs together into one large DFA
  - Unite the starting states of various DFAs into one starting state
  - Simple if each token begins with a different character
  - Becomes more complex if some tokens have a common prefix
- Consider the DFAs for **<**, **<=**, and **<>**
  - They share a common prefix **<**
  - They are combined into one DFA as shown on the right





Algorithmic Aspects of a DFA

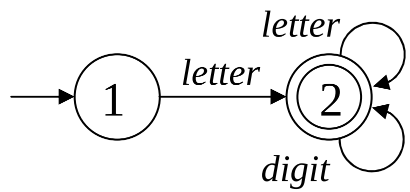
- A DFA diagram is just an **outline** of a scanning algorithm
- A DFA does NOT describe every aspect of the algorithm
- What happens when making a transition? A typical action is to
  - Save the character read in a string buffer belonging to a single token
  - The string value is the lexeme of the token
- What happens when we reach an accepting state?
  - If no further transition is possible, we return the token recognized
  - If further transitions are possible, we continue to **match the longest string**
- What happens when no transition exist from an non-accepting state?
  - We can **backtrack to the last accepting state**, if we visited one
    - The extra characters read, called **lookahead** characters, are returned back to input
  - We can return an **error token** if no accepting state is visited

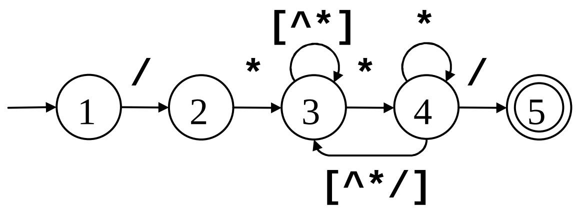


Converting a DFA into an Algorithm

- We can convert a DFA into an algorithm by:
  - Using a variable, *state*, to maintain the current state
  - Writing transitions as case statements inside a loop
    - The first case statement tests the current *state*
    - The nested case statements tests the input character *ch*
    - The *unput*(*ch*) statement returns *ch* back to input

```
state := 1; input(ch);
while not eof do
  case state of
    1: case ch of
         letter: state := 2; input(ch);
         else exit while;
       end case;
    2: case ch of
         letter, digit: input(ch);
         else unput(ch); exit while;
       end case;
  end case;
end while;
if state = 2 then return id; else error; end if;
```






```
state := 1; input(ch);
while not eof do
  case state of
    1: case ch of
          '/': state := 2; input(ch);
          else exit while;
       end case;
    2: case ch of
          '*': state := 3; input(ch);
          else exit while;
       end case;
    3: case ch of
          '*': state := 4; input(ch);
          else state := 3; input(ch);
       end case;
    4: case ch of
          '*': state := 4; input(ch);
          '/': state := 5; exit while;
          else state := 3; input(ch);
       end case;
  end case;
end while;
if state = 5 then accept_comment;
else error; end if;
```

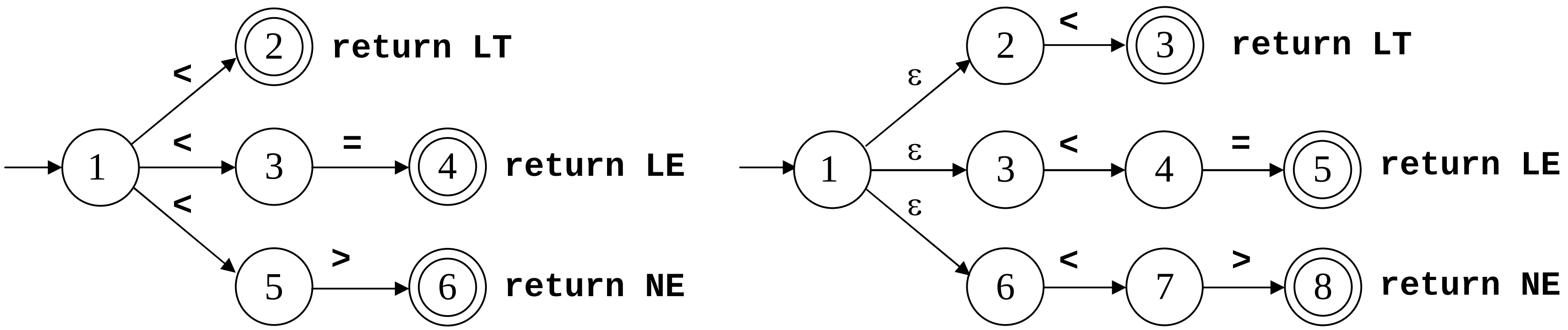
Table-Driven Generic Algorithm for a DFA

- A DFA can be implemented as a generic algorithm
  - Driven by a transition table
- Suitable for scanner generators such as Lex
- Advantages of a generic algorithm:
  - Size of code is reduced
  - Same code works with different DFAs
  - Transition table is only modified
  - Code is easier to change and maintain
- Disadvantages:
  - Transition table can be very large
  - Much of the table space is unused
  - Table compression is required

```
state := 1;
input(ch);
while not eof
   next_state := T[state][ch];
   if next_state = undefined then
      exit while;
   end if ;
   state := next_state;
   input(ch);
end while;
if final(state) then
   unput(ch);       -- extra char
   return token;
else if previous final state
   backtrack to previous final state
   return token;
else
   error;
end if ;
```

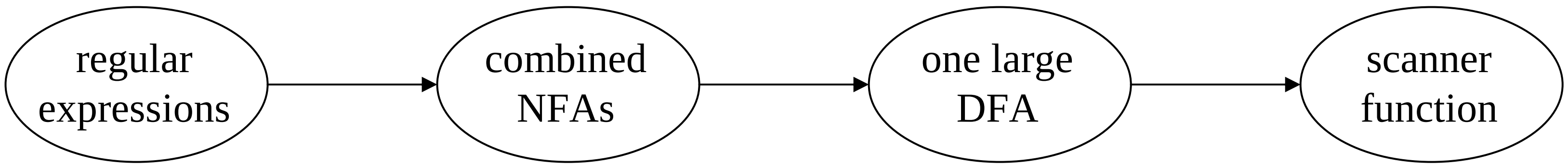
Nondeterministic Finite Automata (NFA)

- An NFA is similar to a DFA except that:
  - Multiple transitions labeled by same character from same state are allowed
  - ε-transitions are allowed
- ε-transitions are spontaneous. They occur without consuming any character
- An NFA can be converted to an algorithm, except that:
  - There can be many transitions that must be tried to match an input sequence of chars
  - Transitions that have not been tried must be stored to backtrack to them on failure
  - Resulting algorithm of NFA is slower than the one that corresponds to a DFA
- DFAs with common prefixes can be combined into one large NFA by:
  - Uniting their starting states as show on the left
  - Introducing a new start state and ε-transitions as shown on the right



From Regular Expressions to Scanner Function

- A scanner generator transforms regular expressions into a function
- First, regular expressions are transformed into NFAs
- Second, combined NFAs are converted into one large DFA
- Third, the DFA is converted into a scanner function

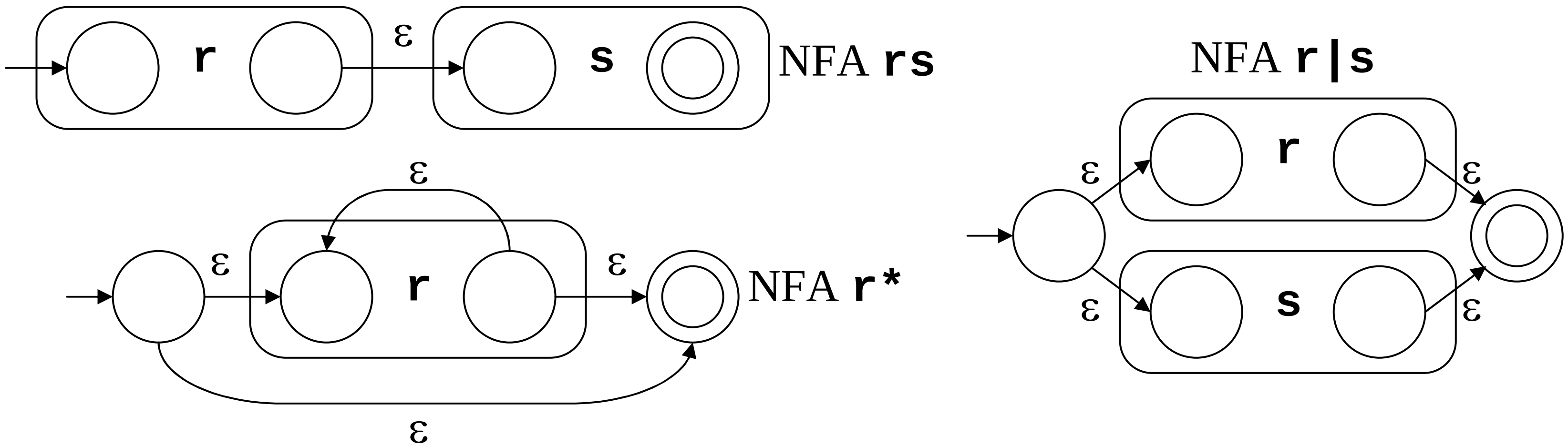


- Thompson's construction transforms regular expressions into NFA
  - Transforming regular expressions into a DFA directly is more complex
- Subset construction is used to transform an NFA into a DFA

From a Regular Expression to an NFA

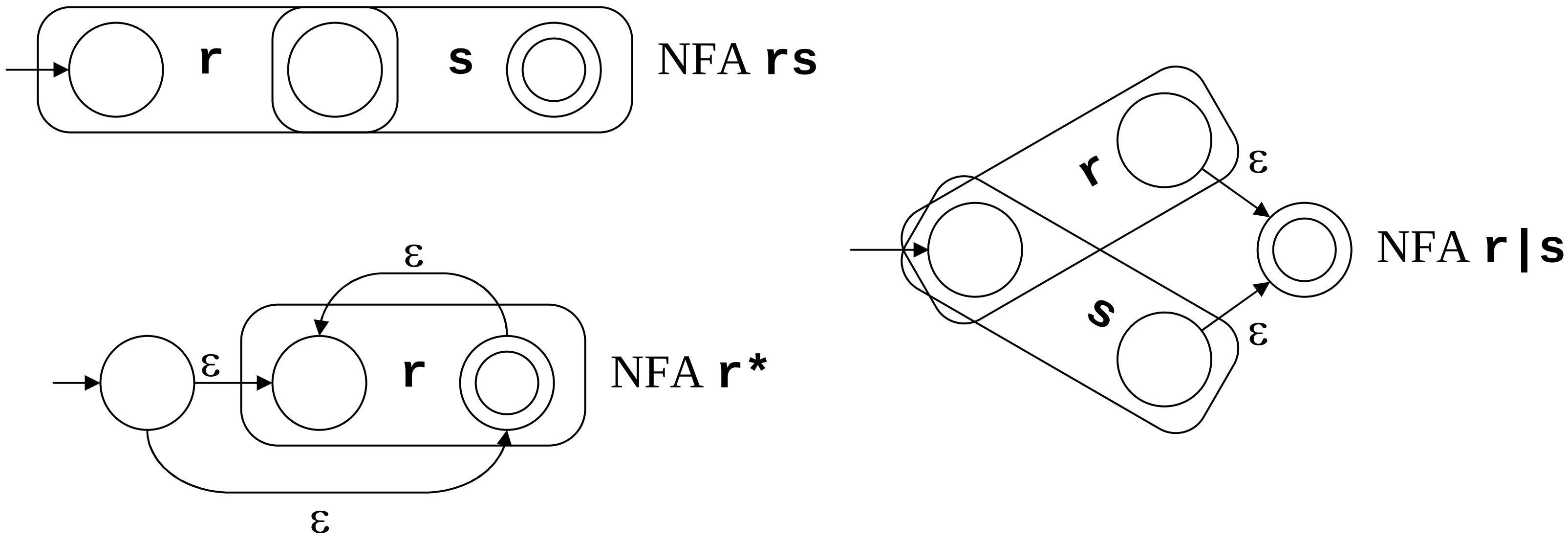
- Regular expressions are built out of:
  - Basic regular expressions **`a`** (where $a \in \Sigma$) and ε
  - Basic operations: concatenation **`r s`**, alternation **`r|s`**, and Kleene closure **`r*`**
- Regular expression for **`a`** and ε

- Thompson's construction of **`r s`**, **`r|s`**, and **`r*`**
  - The NFA of each regular expression **`r`** has one accepting state



Alternative Construction of an NFA

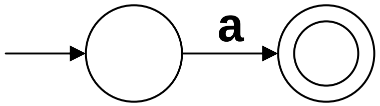
- The following is a variation of Thompson's construction
  - Less ε-transitions
  - Less states
  - The NFA of each regular expression **r** has one accepting state as before
- Construction of **r s**, **r|s**, and **r***



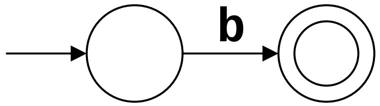
Example on NFA Construction

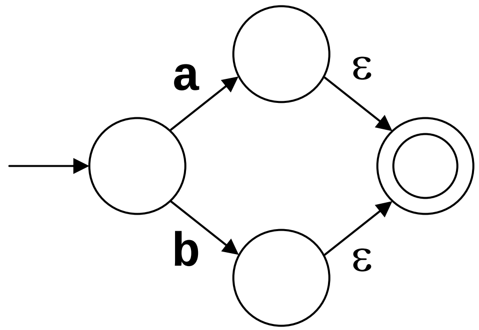
- Consider the construction of **`(a|b)*abb`**

NFA for **a**



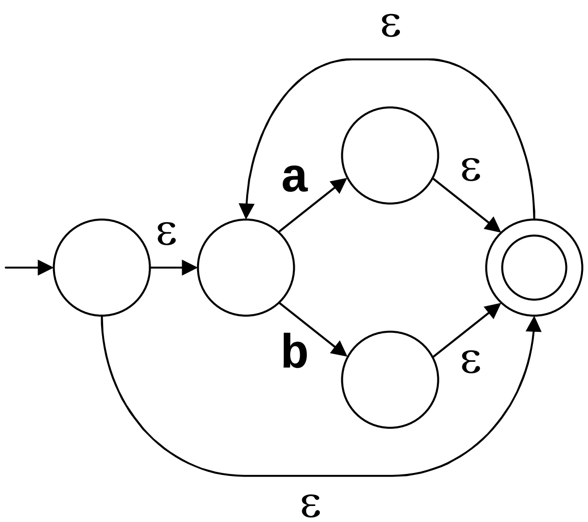
NFA for **b**



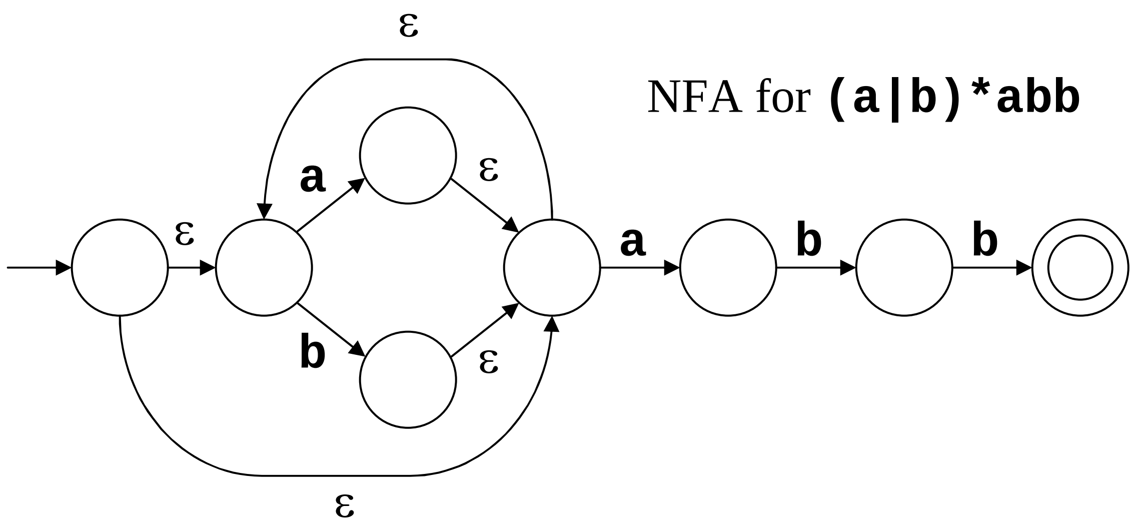


NFA for **`a|b`**

NFA for **`(a|b)*`**



NFA for **`(a|b)*abb`**

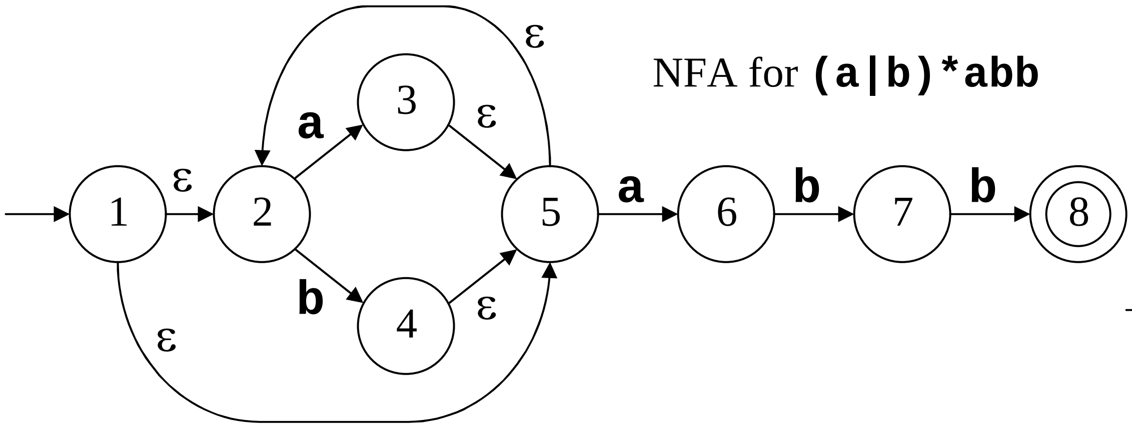


From an NFA to a DFA – Subset Construction

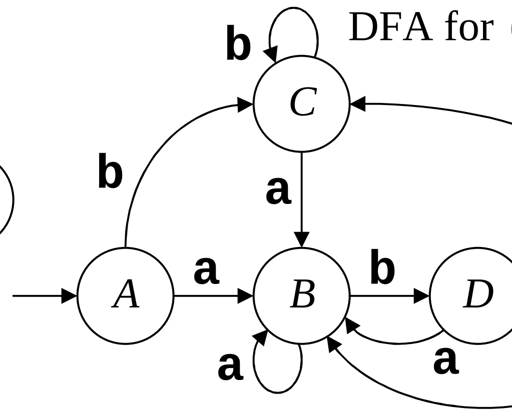
- For any NFA N, we can construct a DFA M equivalent to it
  - Each state of M corresponds to a **subset** of the states of N
  - M will be in state $\{s_1, s_2, s_3\}$ after reading an input string iff N can be in s_1, s_2, or s_3
  - The initial state of M is the subset of all states that N could be in initially
    - This is the set of states reachable from the initial state of N following only ε-transitions
  - The set of states reachable following only ε-transitions is called the ε-closure
    - ε-closure(state s) = $\{s\} \cup$ {all states reachable from s following only ε-transitions}
    - Start state of M = ε-closure(start state of N)
  - Once the start state of M is computed, we determine the successor states
    - Take any state S of M, S corresponds to a subset of states of N. $S = \{s_1, s_2, \ldots\}$
    - To compute S-successor under character c, we find the successors of $\{s_1, s_2, \ldots\}$ under c
    - The successors of $\{s_1, s_2, \ldots\}$ under c will be a new set of states $\{t_1, t_2, \ldots\}$
    - We compute T = ε-closure($\{t_1, t_2, \ldots\}$) ; $\quad \varepsilon\text{-closure(set of states } T) = \bigcup_{t \in T} \varepsilon\text{-closure}(t)$
    - T is included in M and a transition from S to T is labeled with c
  - We continue adding states and transitions to M until all possible successors are added
  - The process of adding new states to M must eventually terminate. Why?

Example on Subset Construction Algorithm

- The start state of the DFA is ε-closure({1}) = {1, 2, 5}; Call it state *A*
- *A*-successor under *a* is {3, 6}; ε-closure({3, 6}) = {3, 6, 5, 2}; Call it state *B*
- *A*-successor under *b* is {4}; ε-closure({4}) = {4, 5, 2}; Call it state *C*
- *B*-successor under *a* is {3, 6}; ε-closure({3, 6}) = {3, 6, 5, 2}; This is state *B*
- *B*-successor under *b* is {4, 7}; ε-closure({4, 7}) = {4, 7, 5, 2}; Call it state *D*
- *C*-successor under *a* is {3, 6}; ε-closure({3, 6}) = {3, 6, 5, 2}; This is state *B*
- *C*-successor under *b* is {4}; ε-closure({4}) = {4, 5, 2}; This is state *C*
- *D*-successor under *a* is {3, 6}; ε-closure({3, 6}) = {3, 6, 5, 2}; This is state *B*
- *D*-successor under *b* is {4, 8}; ε-closure({4, 8}) = {4, 8, 5, 2}; Call it state *E*
- *E*-successor under *a* is {3, 6}; ε-closure({3, 6}) = {3, 6, 5, 2}; This is state *B*
- *E*-successor under *b* is {4}; ε-closure({4}) = {4, 5, 2}; This is state *C*



NFA for **(a|b)*abb**



DFA for **(a|b)*abb**

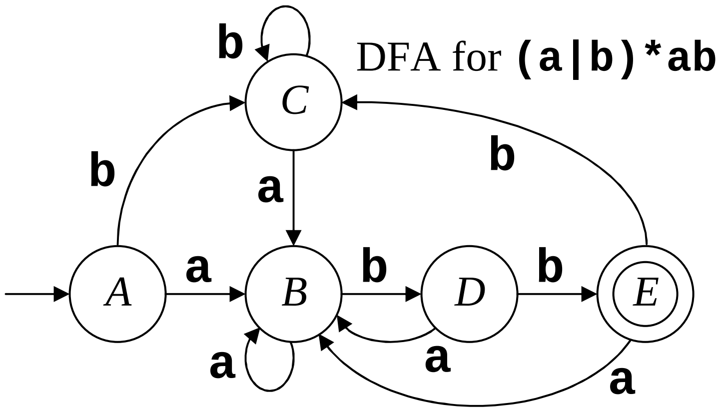
Minimizing the Number of States in a DFA

- The DFA obtained by the subset construction algorithm can be minimized
- State *s* can be **distinguished** from state *t* in a DFA when for some string *w*:
  - Starting at state *s* and reading string *w*, we end up in an accepting state
  - Starting at state *t* and reading string *w*, we end up in a non-accepting state
- An algorithm that produces a minimum-state DFA is given below:

  1. Construct an initial partition Π of the DFA set of states, *S*, with 2 groups:
     - The set of final states *F*
     - The set of non-final states $S - F$
  2. For each group *G* of Π:
     - Partition *G* into subgroups such that 2 states *s* and *t* of *G* are in the same subgroup iff:
       - $\forall\ a \in \Sigma$, states *s* and *t* have transitions on *a* to states in the same subgroup of Π
     - Call the new partition Π_{new}. At worse, each state will be in a subgroup by itself
  3. If $\Pi_{new} \neq \Pi$ then go back to step 2 with $\Pi := \Pi_{new}$; otherwise, proceed at step 4
  4. Each group in the final Π becomes a state in the minimized DFA
     - The states of a group *G* of Π cannot be distinguished and are merged into one state
     - A transition from group G_1 to G_2 is marked with input symbol *a* when:
       - All states of G_1 make transition to states in G_2 on input symbol *a*

Example on DFA Minimization

- Consider the DFA for **(a|b)*abb** obtained using subset construction algorithm
- Initial partition Π consists of 2 groups = {{*A*, *B*, *C*, *D*},{*E*}}
- {*A*, *B*, *C*}-succ under *b* ∈ {*A*, *B*, *C*, *D*}, while *D*-succ under *b* is *E*
- Therefore, Π_{new} = {{*A*, *B*, *C*},{*D*},{*E*}}
- {*A*, *C*}-succ under *b* is *C* while *B*-succ under *b* is *D*
- Therefore, Π_{new} = {{*A*, *C*},{*B*},{*D*},{*E*}}
- {*A*, *C*}-succ under *a* is *B*, and {*A*, *C*}-succ under *b* is *C*
- {*A*, *C*} does not require further partitioning; states *A* and *C* can be merged
- Therefore, final Π = {{*A*, *C*},{*B*},{*D*},{*E*}}



DFA for **(a|b)*abb**

Minimized DFA for **(a|b)*abb**