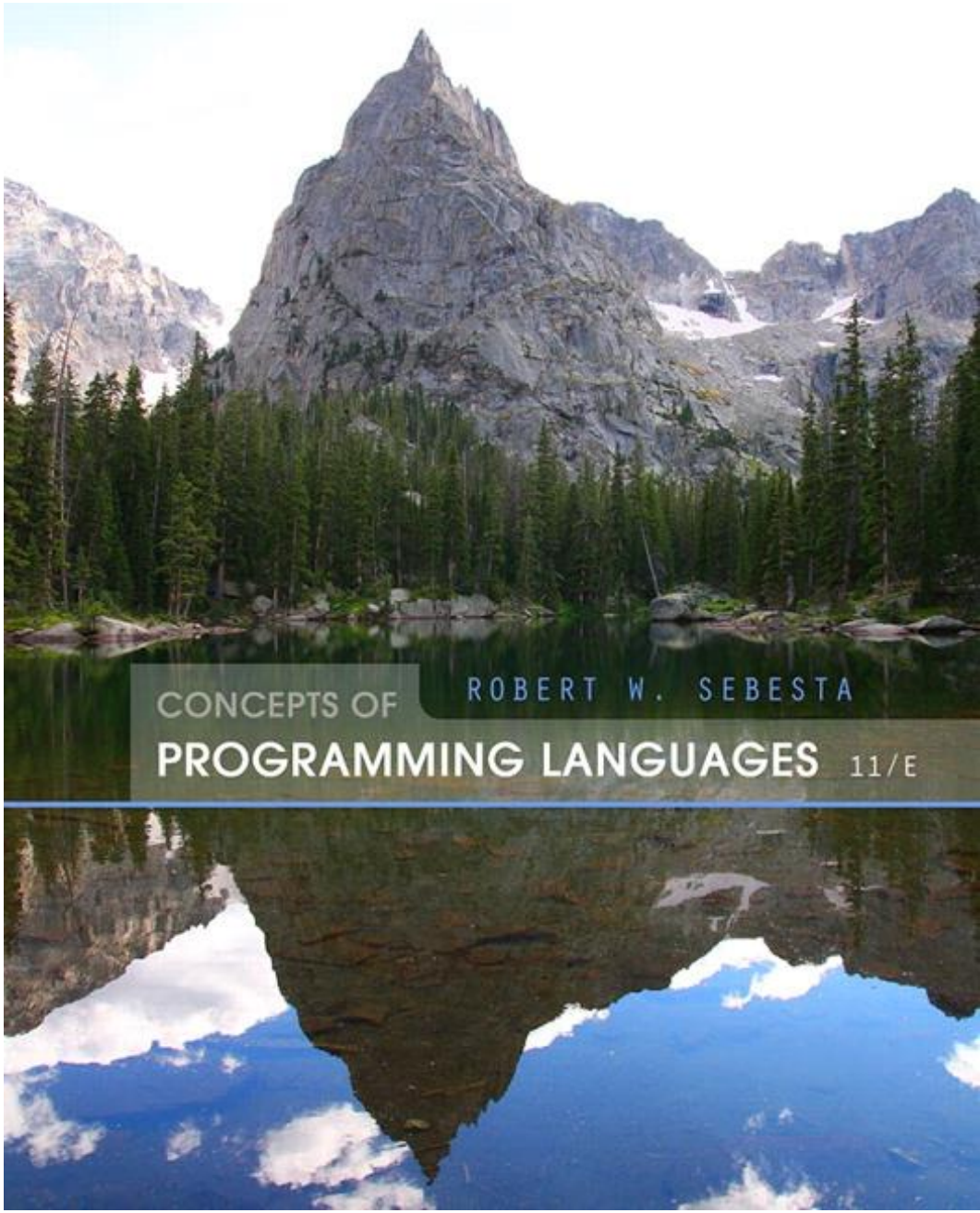
Chapter 4

Lexical and Syntax Analysis



ISBN 0-321-49362-1

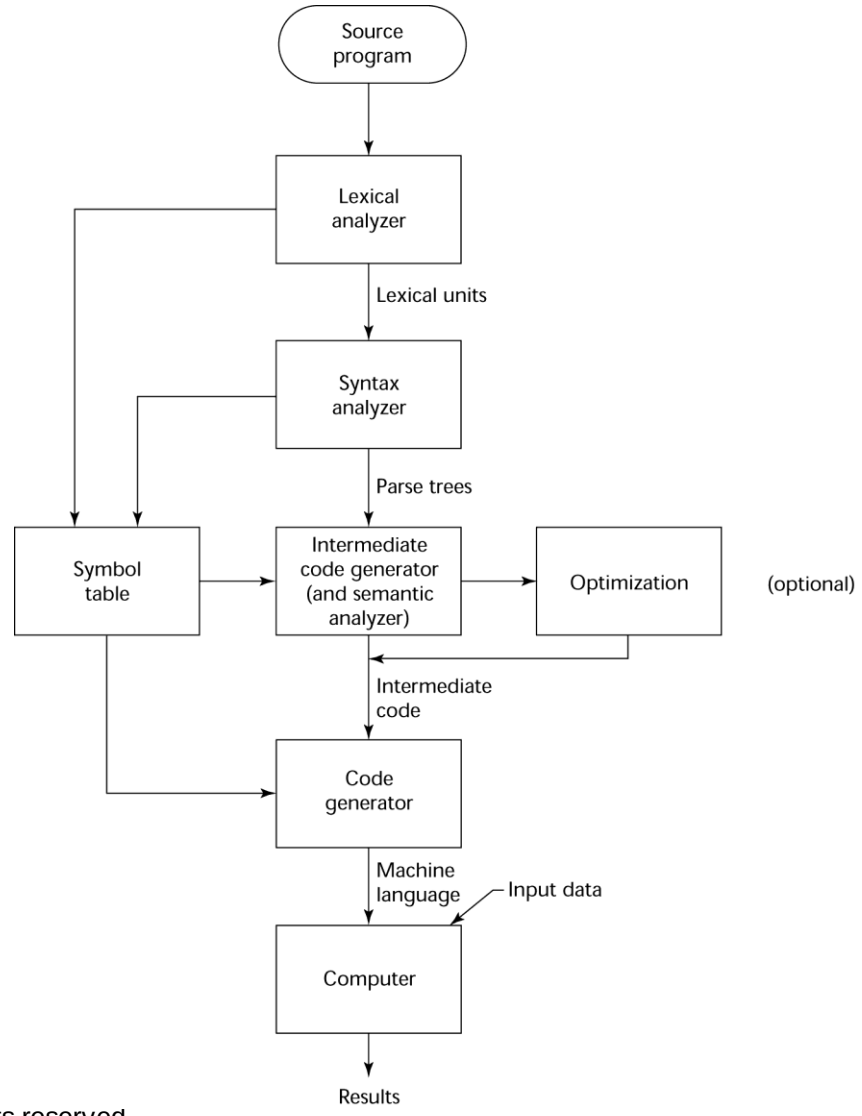
Learning Objectives

- Describe the differences between compilers and interpreters and outline the phases and merits of each.
- Recognize the underlying formal models such as finite state automata and their connection to language definition through regular expressions and grammars.
- Gain hands-on experience with building a parser, both manually and using some language tools.
- Demonstrate the different forms of binding, scope, and lifetime in the imperative model.
- Evaluate languages with regard to typing, e.g. static vs. dynamic typing.
- Describe the importance of types and type-checking in providing abstraction and safety.

Chapter 4 Topics

- Introduction
- Lexical Analysis
  - Regular Expressions
  - Nondeterministic Finite Automaton (NFA)
  - Deterministic Finite Automaton (DFA)
- Syntax Analysis
  - Recursive-Descent Parsing
  - Bottom-Up Parsing

The Compilation Process



Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Advantages of Using BNF to Describe Syntax

- Provides a clear and concise syntax description
- The parser can be based directly on the BNF
- Parsers based on BNF are easy to maintain

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* – less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* – separation allows optimization of the lexical analyzer
- *Portability* – parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- A "front-end" for the parser
- A lexical analyzer is a **pattern matcher** for character strings
- Identifies substrings of the source program that belong together – *lexemes*
  - Lexemes match a **character pattern**, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token may be `Identifier`

Lexical Analysis

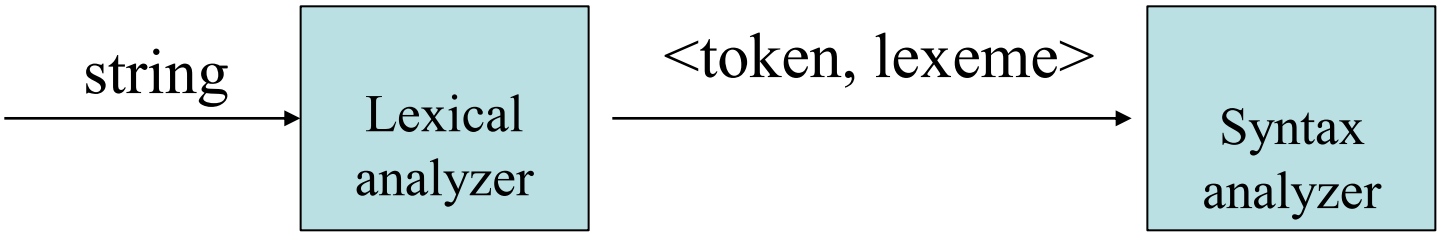
- Token (or token class)
  - In English
    - Noun, verb, adjectives,…
  - In a programming language
    - Identifier, Keyword, Number, ‘(’, ‘)’, ‘+’,….
    - Each token class corresponds to some set of strings that could appear in a program

Lexical Analysis

- Identifier – strings of letters or digits, starting with a letter
  - B1, Foo, A33a
- Integer – a non-empty string of digits
  - 1, 5, 003, 00
- Keyword:
  - “if”, “else”, “class”
- Operator: +, -, ==, <=
- Whitespace:
  - A non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- The goal of lexical analysis
  - Tokenize – classify program substrings according to role (token class)
  - Communicate <token, lexeme> pairs to the syntax analyzer



  - Example:
    - \tif(a == b)\n\t\tc=0;\n\telse\n\t\tc=1;

Regular Expression

- Used to specify the lexical structure (token classes) of a programming language
  - What set of strings is in a token class--use regular languages
- Can be defined by regular expression

Regular Expression

- A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings
- Each regular expression denotes a set of strings
- For each token class, we need to determine its regular expression

Definition of Regular Expression

R is a regular expression if it is:

1. a for some *a* in the alphabet Σ, standing for the language {a}
2. ϵ, standing for the language {ϵ}, empty string
3. Ø, standing for the empty language
4. R_1+R_2 where R_1 and R_2 are regular expressions, and + signifies union (sometimes | is used)
5. R_1R_2 where R_1 and R_2 are regular expressions and this signifies concatenation
6. R* where R is a regular expression and signifies closure
7. (R) where R is a regular expression, then a parenthesized R is also a regular expression

This definition may seem circular, but 1-3 form the basis
Precedence: Parentheses have the highest precedence, followed by *, concatenation, and then union.

Definition of Regular Expression

- Over some alphabet $\sum$
  - The family of characters
- Two basic expressions
  - Single character, ‘c’ = {“c”}
  - Epsilon, ϵ = {“”}
- Three compound expressions
  - Union: A + B, is also a regular expression
  - Concatenation: AB , is also a regular expression
  - Iteration (Kleene closure): A*, is also a regular expression

Compound Regular Expressions

- A and B are two regular expressions, L(A) = a + b, L(B) = c + d
- Union: L(A +B) = L(A)UL(B) = {a, b, c, d}
- Concatenation: L(AB) = L(A) concatenated with L(B) = {ab, ac, bc, bd}
  - Cross product
- Kleene closure: L(A*) = ϵ U L(A) U L(AA) U L(AAA) U…..= {ϵ, a, b, aa, ab, ba, bb, ……}
- Precedence
  - Kleene closure > concatenation > union

Operands in a Regular Expression

- **Characters** from the alphabet over which the regular expression is defined
- **Variables** whose values are any pattern defined by a regular expression
- **Epsilon** which denotes the empty string containing no characters
- **Null** which denotes the empty set of strings

A Regular Expression Example

- Regular expressions over
- $\Sigma = \{0, 1\}$
  - 1^*
  - $(1+0)1$
  - $0^* + 1^*$
  - $(0+1)^*$

Lexical Specification

- Keywords
  - ‘if’ + ‘else’ + ‘then’ + …
- Integer–a non–empty string of digits
  - digit = ‘0’+‘1’+‘2’+‘3’+‘4’+‘5’+‘6’+‘7+‘8’+‘9’
  - digits = digit digit* = digit$^+$
- Identifier–strings of letters or digits, starting with a letter
  - letter = ‘a’+‘b’+…+‘z’+‘A’+‘B’+…‘Z’=[a–zA–Z]
  - Identifier = letter(letter + digit)*
- Whitespace = (‘ ’ + ‘\n’ + ‘\t’)$^+$
- OpenPran = ‘(’, ClosePran = ‘)’,….

RegExpr Notions

- At least one: $A^+ == AA^*$
- Option: $A? == A + \epsilon$
- Wildcard matching any character except newline: '.'
- Range: 'a'+'b'+…+'z' = [a-z]
- Exclude Range: complement of (a-z):
  [^a-z] == ~(a-z)
- Start of a line: '^'
- End of a line: '$'

RegExp for Floating Point Number

Create a regular expression for floating point number with scientific notation, e.g., -2.52E-3

- digit = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7+'8'+'9'
- digits = digit$^+$
- sign = ('+' + '-')?
- fraction = ('.' digits)?
- exponent = (('E' + 'e') sign digits)?
- floatNum = sign digits fraction exponent

A different way of defining it:

- [+-]?\d+(\.\d+)?([eE][+-]?\d+)?

Specify Lexical Spec using RegExp

- How much input is used?
  - Maximal much. When faced with a choice of two different prefixes of the input, either of which would be a valid token, always take the longer one
  - e.g., '=' and '=='
- Which token class is used?
  - Priority ordering. The rule listed first has higher priority, e.g., keywords are listed before identifier
- What if no rule matches?
  - Error = [all strings not in the lexical spec]
  - Put it last in priority

Exercises: Use RegExp to define

- The set of strings over {0,1} that end in 3 consecutive 1's
- The set of strings over {0,1} that have at least one 1
- The set of strings over {0,1} that have at most one 1
- The set of strings over {a..z,A..Z} that contain the word PLS
- The set of strings over {a..z,A..Z} that contains at least 2 Ms

Topic Map

Finite Automata

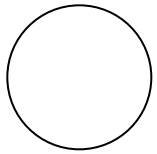
- Regular expressions can be used to describe lexical specification
- Finite automata are the implementation model
- A finite automaton consists of
  - An input alphabet $\sum$
  - A finite set of states S
  - A start state N
  - A set of accepting states F
  - A set of transitions state $->^{input}$ state

Finite Automata

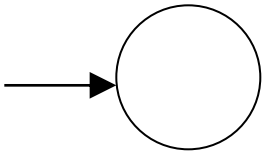
- Start from the start state N
- Transition
  - $s_1 ->^a s_2$: in state s_1 on input a go to state s_2
- If an automaton ends in an accepting state when it gets to the end of the input => **accept** the input string
- Otherwise => **reject**
  - Terminates in a state that is not an accepting state
  - Gets stuck

Finite Automata

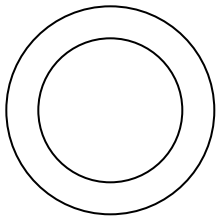
- A state



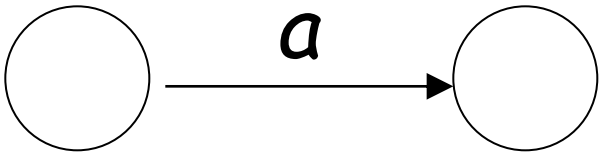
- Start state



- Accepting state



- Transition



Examples

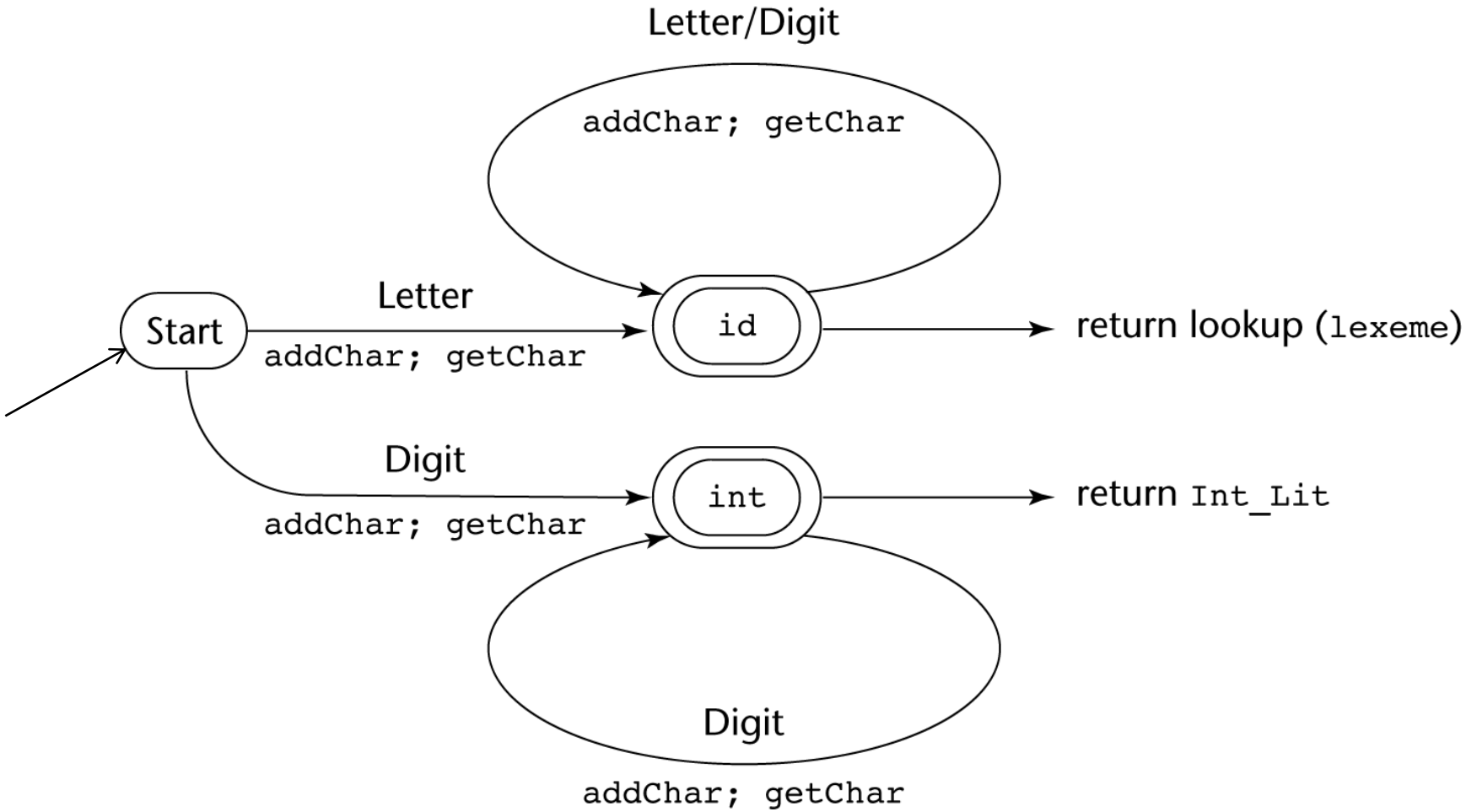
- Alphabet: {0, 1}, define a finite automaton that can recognize
  - only “0”
  - only “101”
  - “1” and “0”
  - any number of 1s followed by a single 0

- Regular expressions?

Lexical Specification

- Keywords
  - ‘if’ + ‘else’ + ‘then’ + …
- Integer–a non–empty string of digits
  - digit = ‘0’+‘1’+‘2’+‘3’+‘4’+‘5’+‘6’+‘7+‘8’+‘9’
  - digits = digit digit* = digit$^+$
- Identifier–strings of letters or digits, starting with a letter
  - letter = ‘a’+‘b’+…+‘z’+‘A’+‘B’+…‘Z’=[a–zA–Z]
  - Identifier = letter(letter + digit)*
- Whitespace = (‘ ’ + ‘\n’ + ‘\t’)$^+$
- OpenPran = ‘(’, ClosePran = ‘)’,….

Finite Automaton (or State Diagram)



State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language – such a diagram would be very large!

Lexical Analysis (continued)

- In many cases, transitions can be combined to simplify the state diagram
- Equivalent class of characters
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent – use a digit class

Lexical Analysis (continued)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (continued)

- Convenient utility subprograms:
  - **`getChar`** – gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** – puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme`**
  - **`lookup`** – determines whether the string in **`lexeme`** is a reserved word (returns a code)

Lexical Analyzer

Implementation:

→ SHOW `front.c` (pp. 172–177)

– Following is the output of the lexical analyzer of `front.c` when used on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

RegExp might be used in Program3

- Match a letter
  - [a-zA-Z] or [[:alpha:]]
- Match a digit
  - [0-9] or \d or [[:digit:]]
- Match an alphanumeric
  - \w or [[:alnum:]]
- Match white space
  - \s or [[:space:]]
- Match a string literal
  - /matchMe/

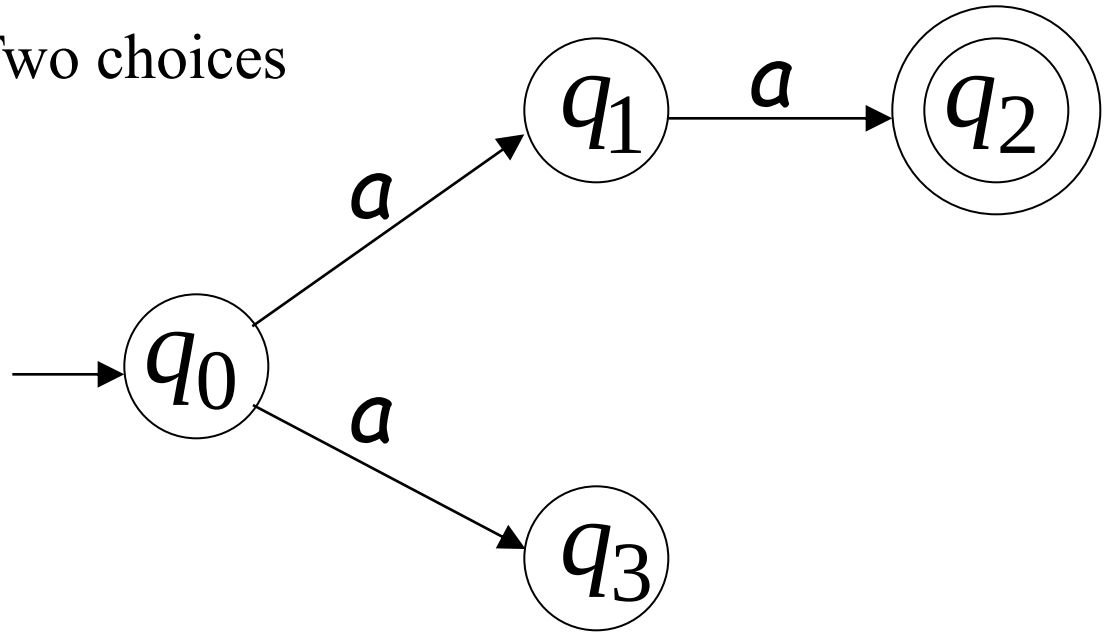
Two Types of Finite Automata

- Nondeterministic Finite Automaton (NFA)
  - From a state, there are two or more outgoing transitions labeled with the same character (duplicate transitions), or
  - With λ transition(s)

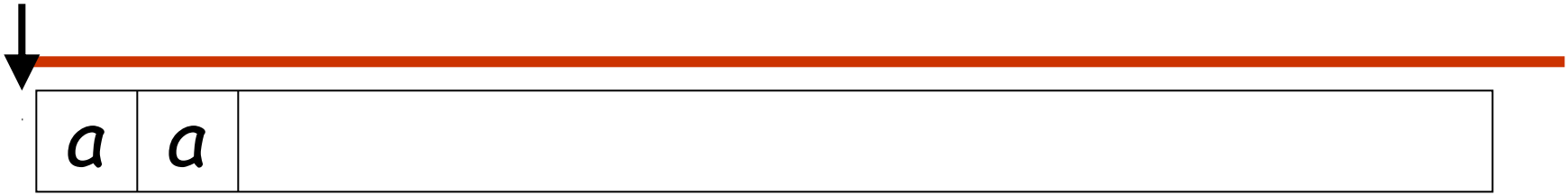
- Deterministic Finite Automaton (DFA)
  - For each input symbol, one can determine the state to which the machine will move

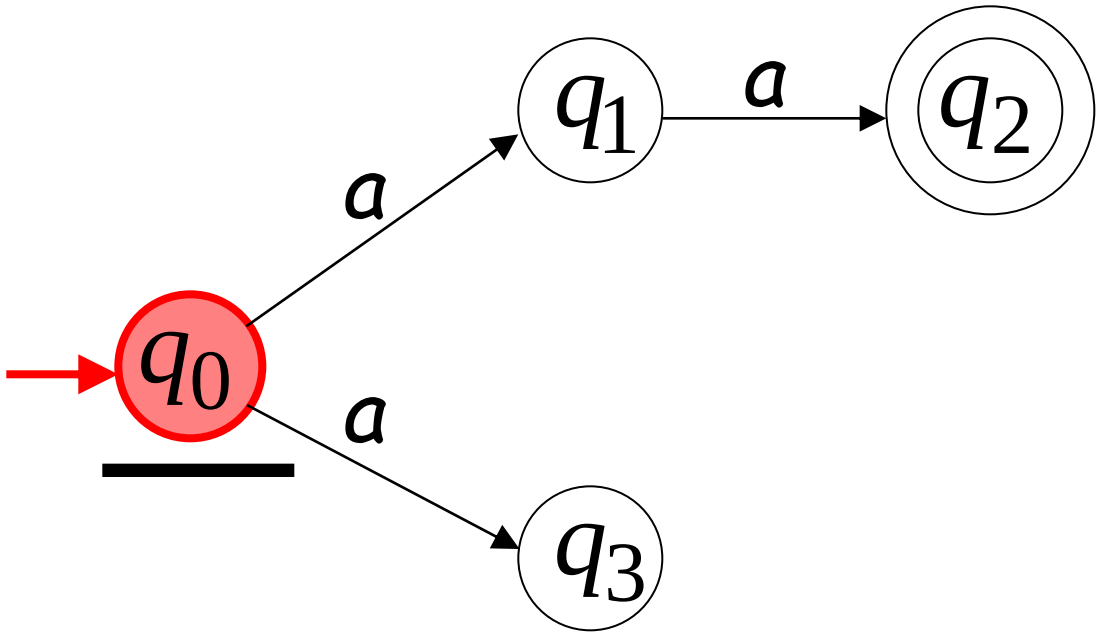
Nondeterministic Finite Automaton (NFA)

Alphabet = $\{a\}$

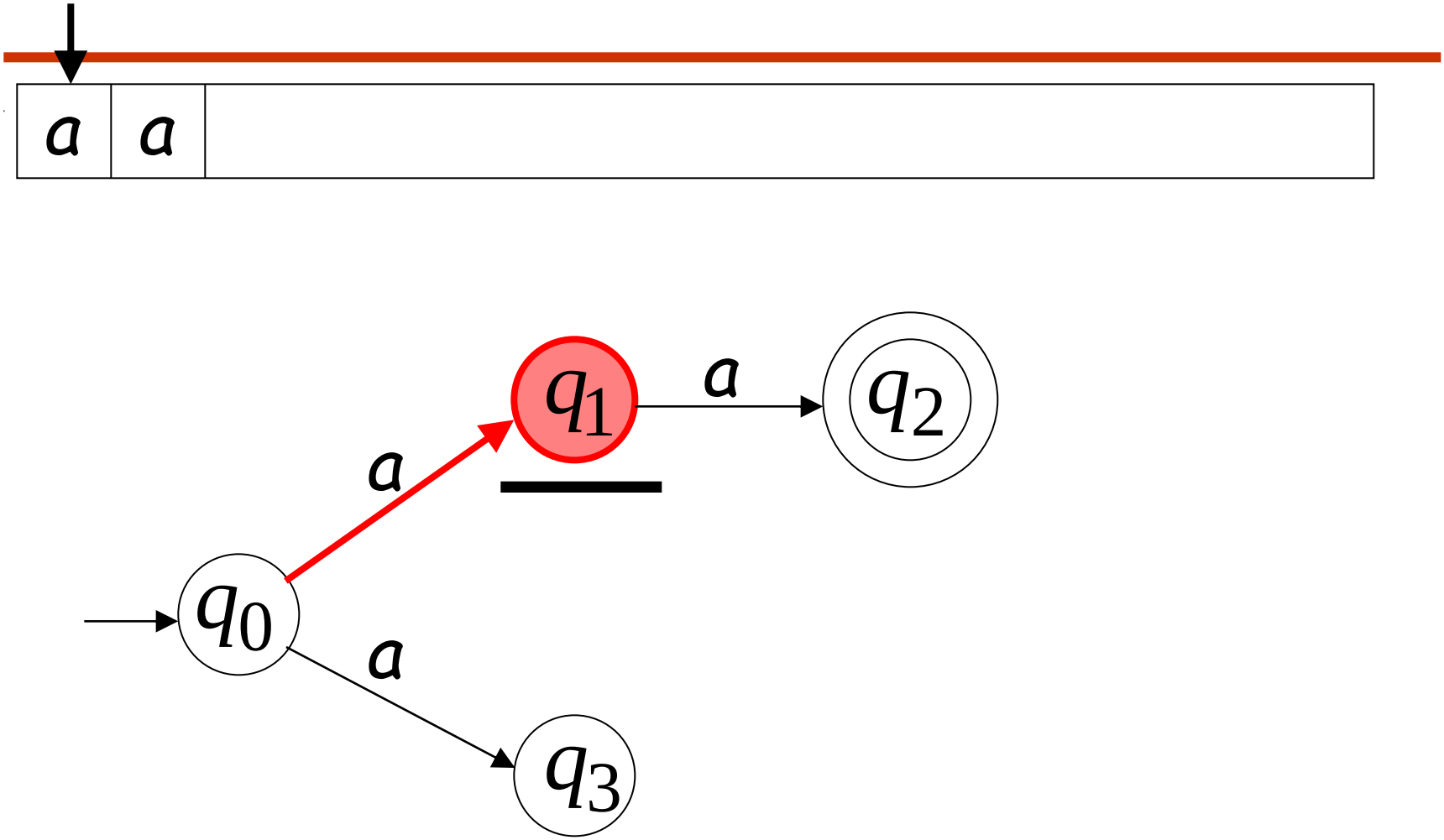


First Choice

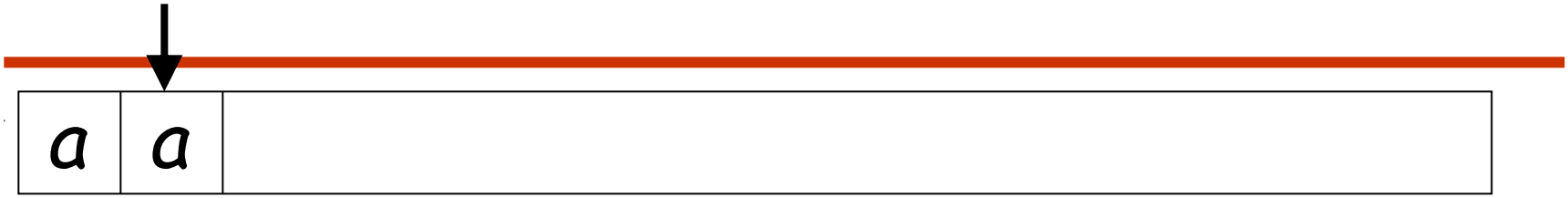


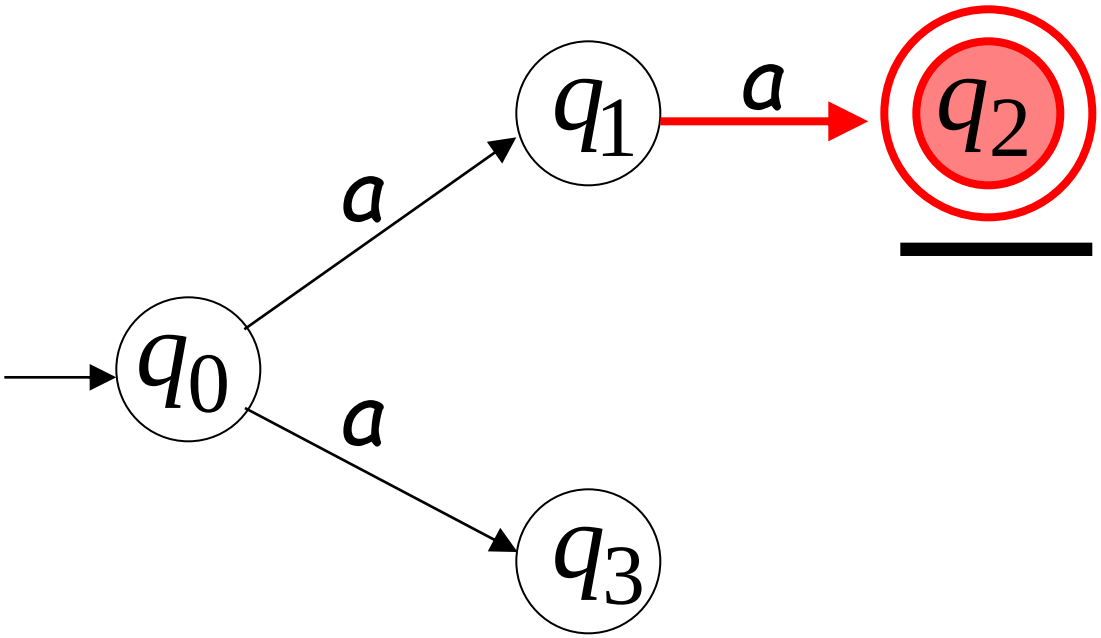


First Choice

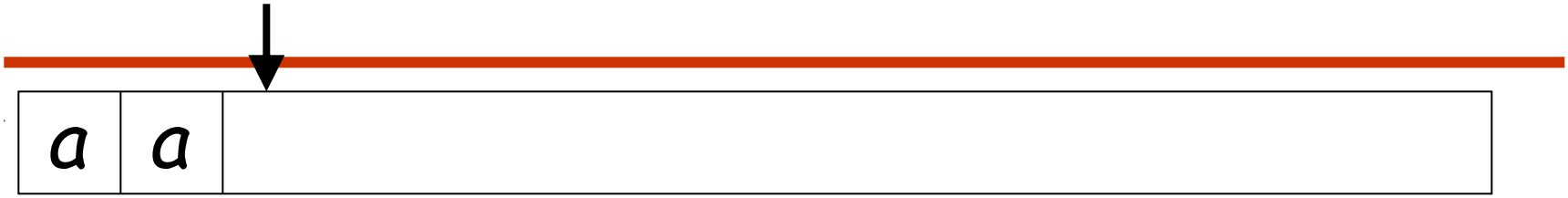


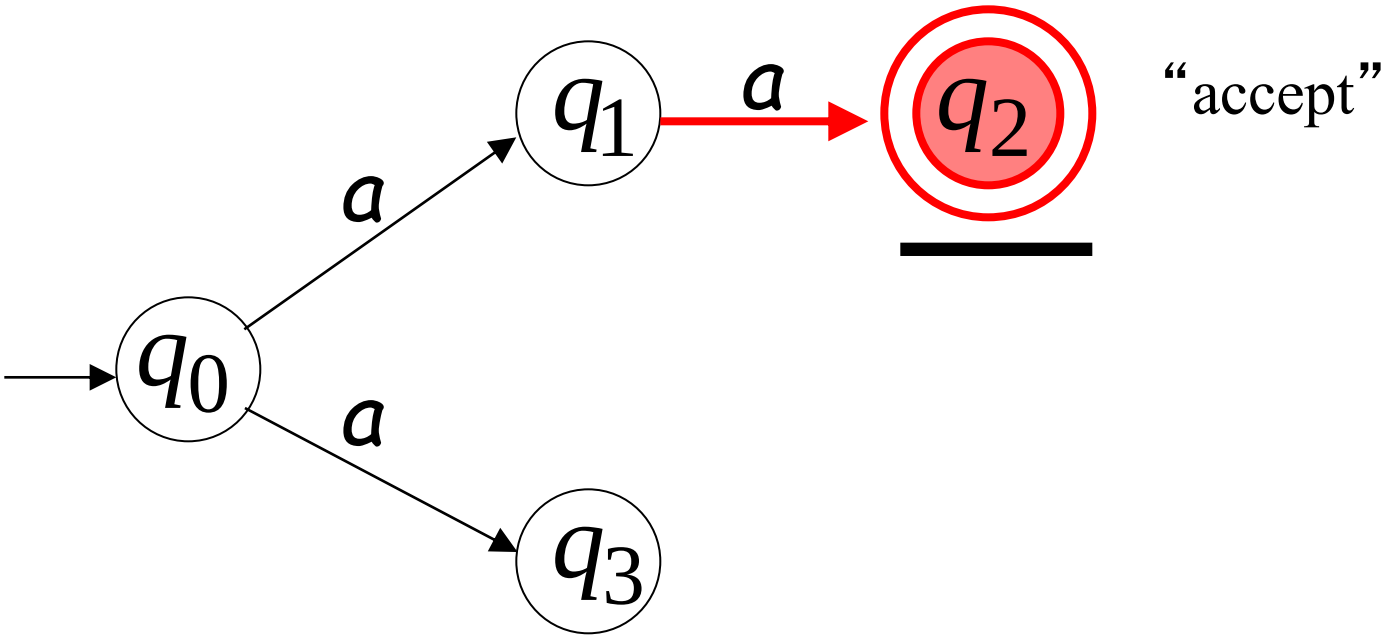
First Choice



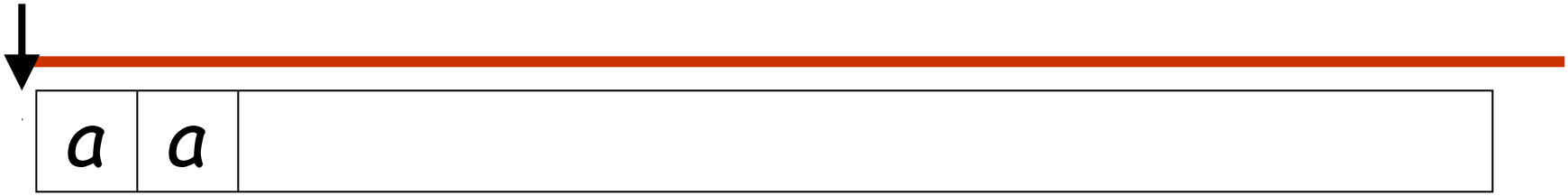


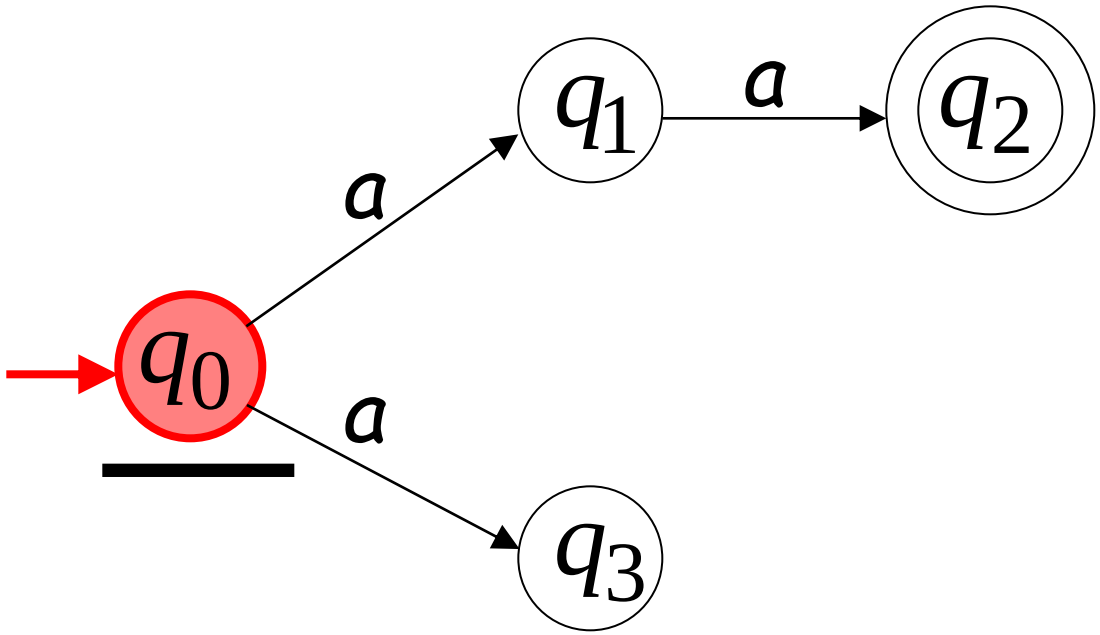
First Choice



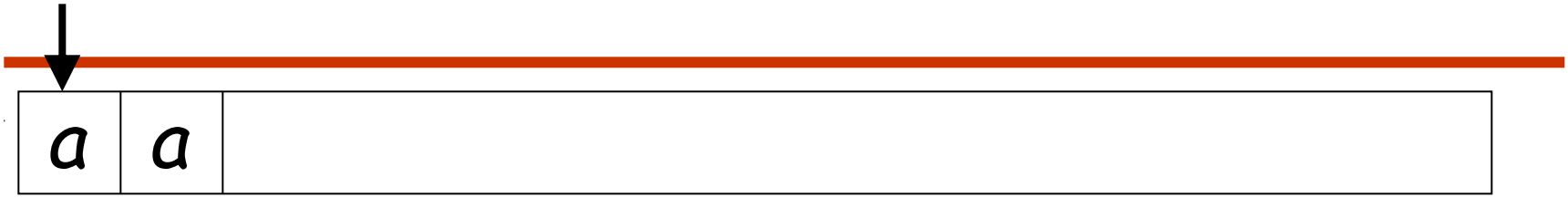


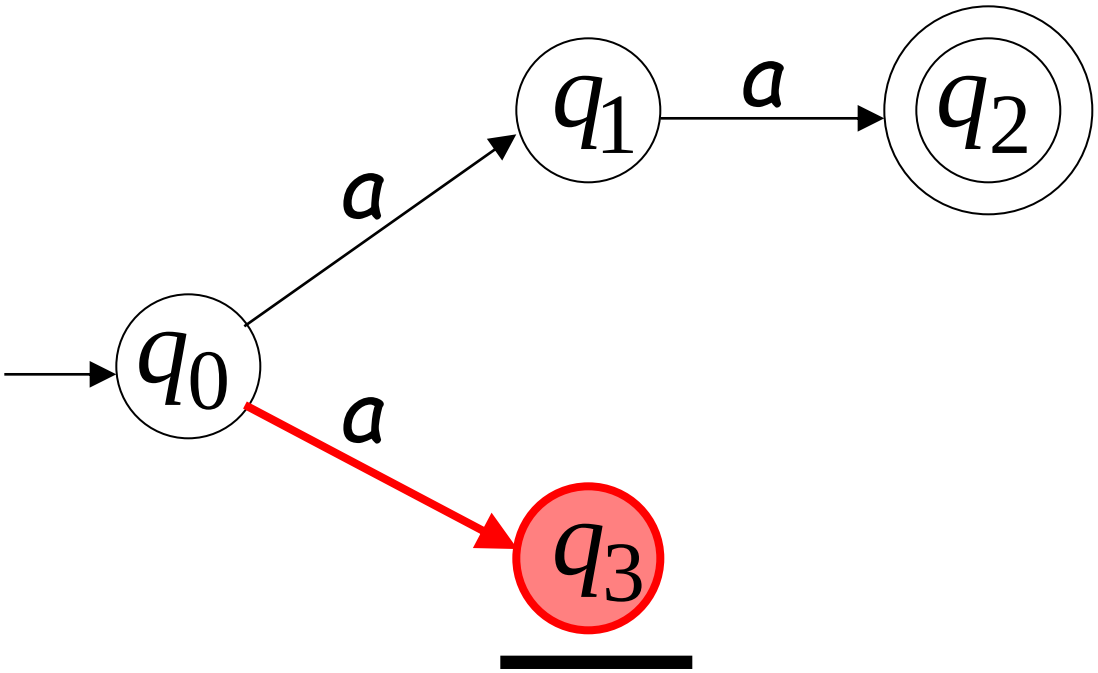
Second Choice



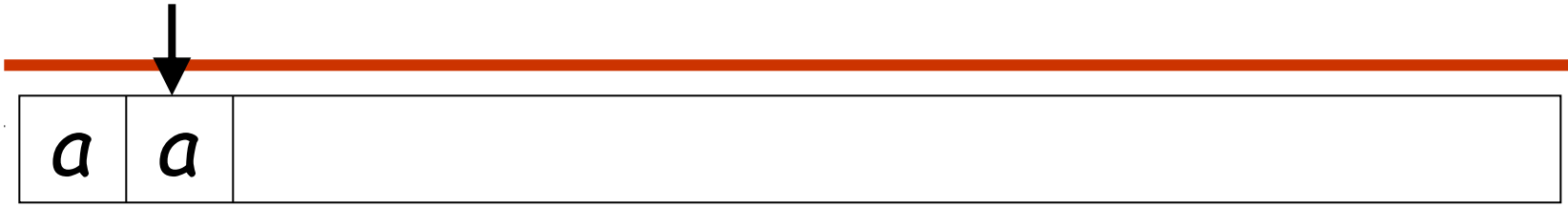


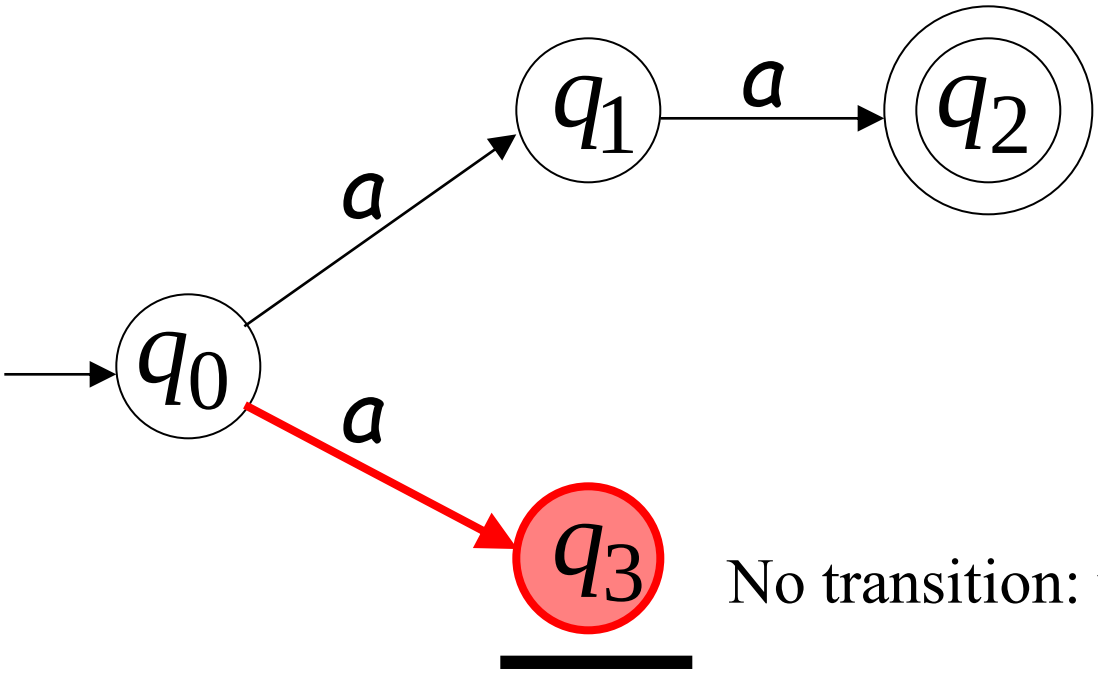
Second Choice





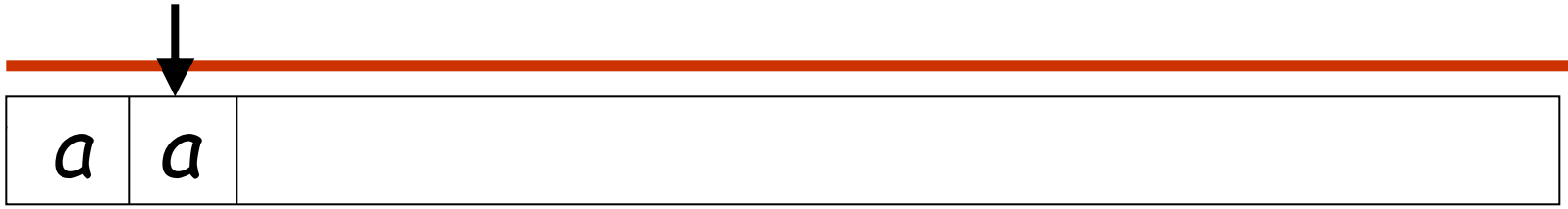
Second Choice

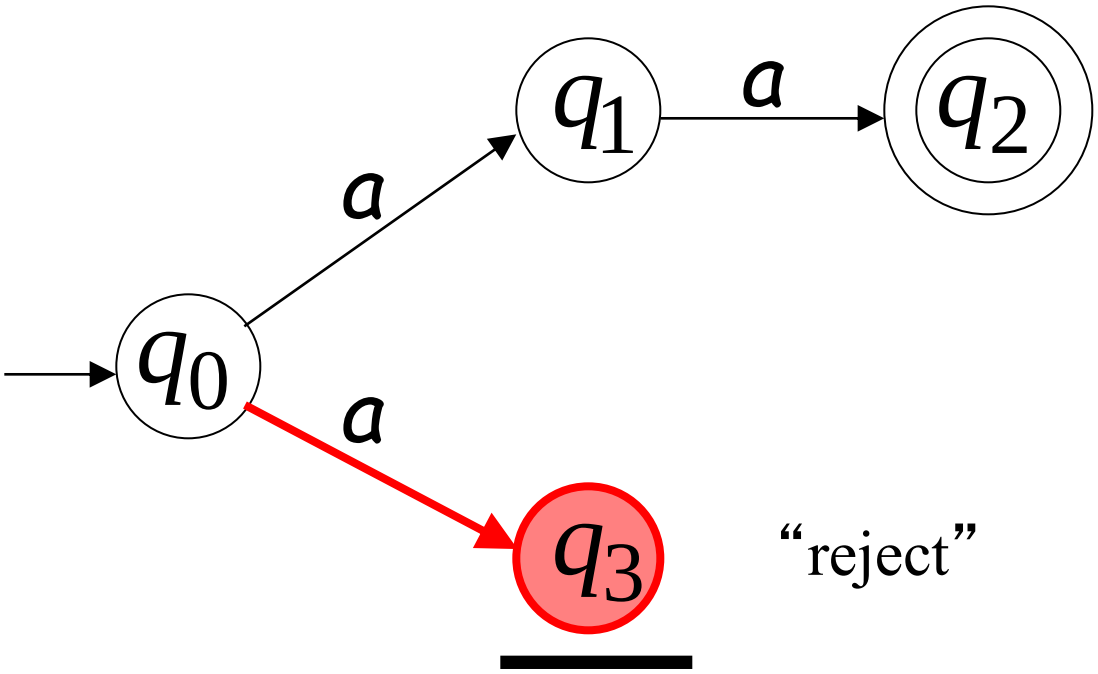




No transition: the FA hangs

Second Choice



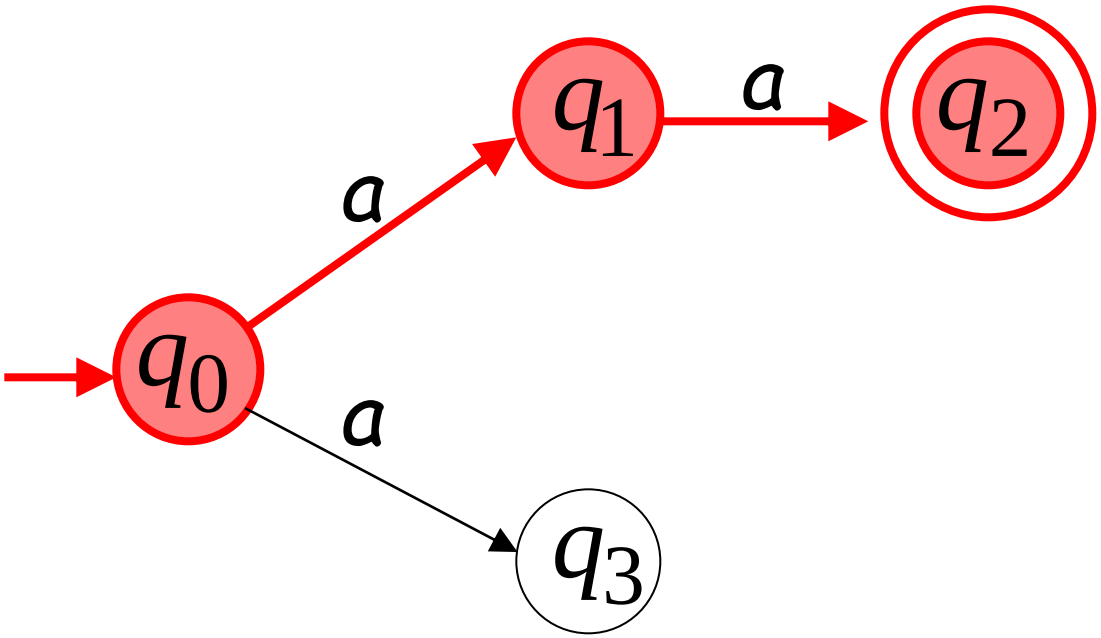


Observation

- An NFA accepts a string if there is a computation of the NFA that accepts the string
- Language of a NFA $=$ the set of accepted strings

Example

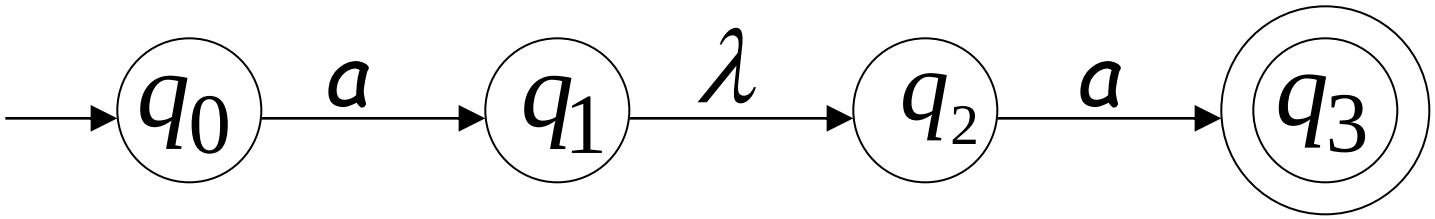
aa is accepted by the NFA:

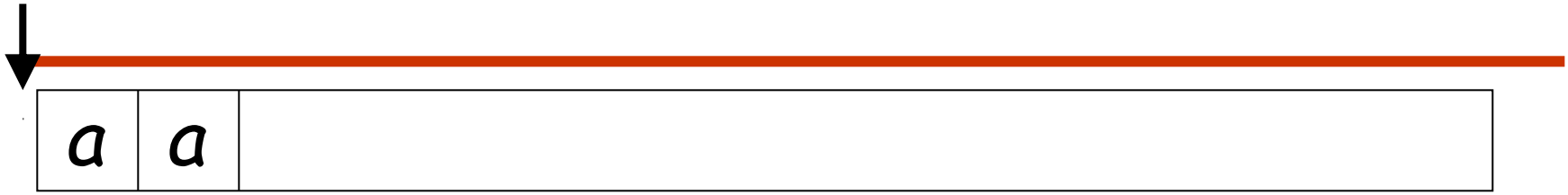


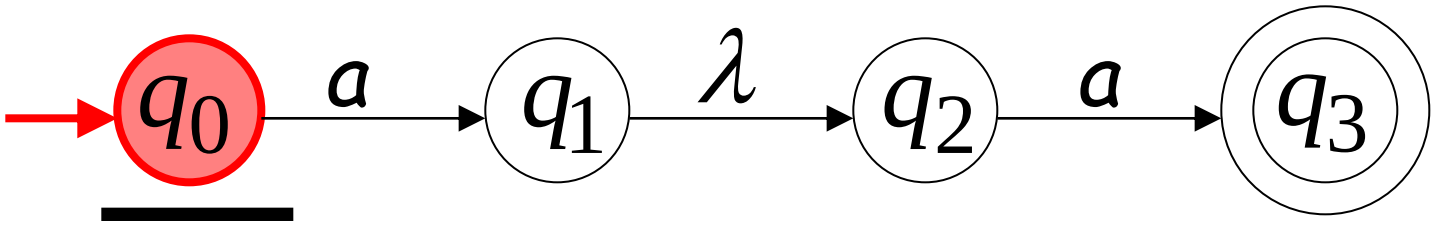
Question: does the NFA accept aaa ?

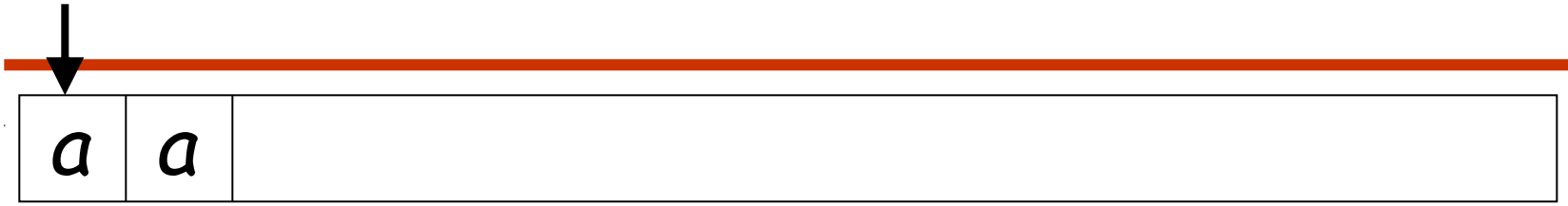
Lambda Transition

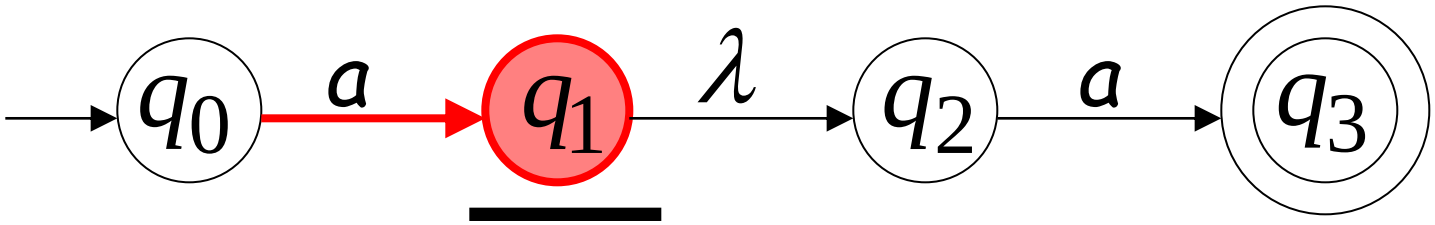
$\lambda \equiv \varepsilon$ both denote an empty string



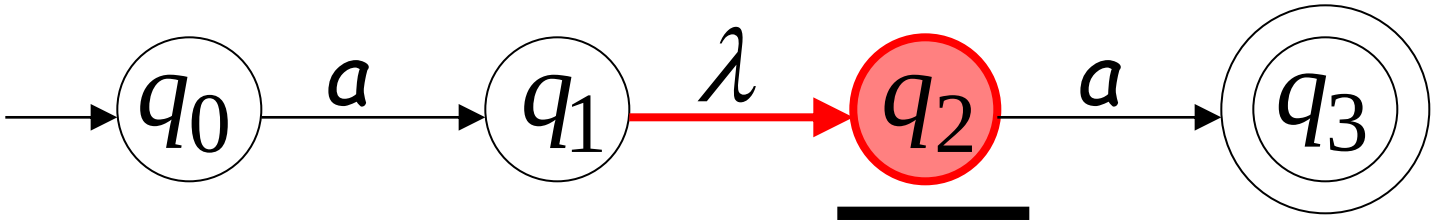

a
a

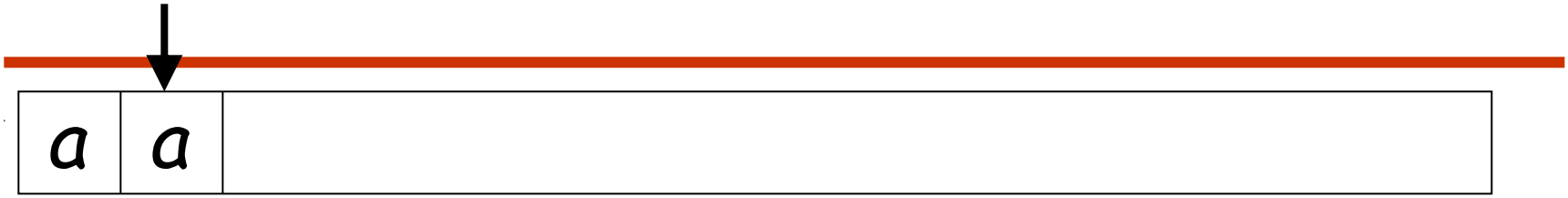

q_0
a
q_1
λ
q_2
a
q_3

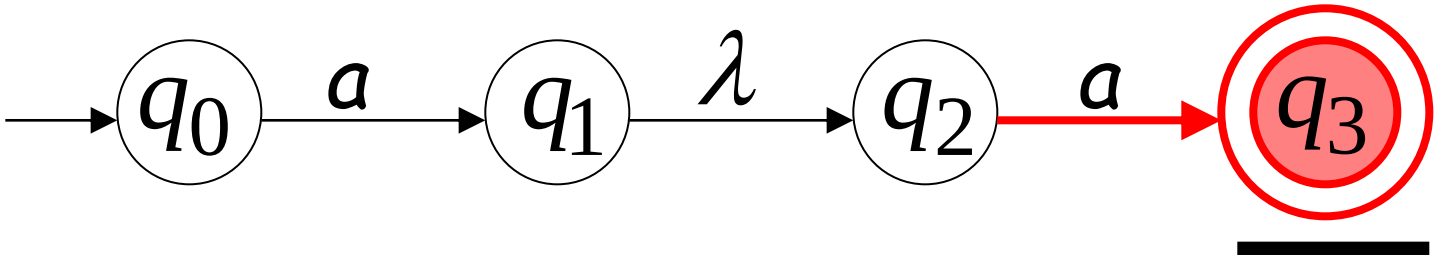

a
a

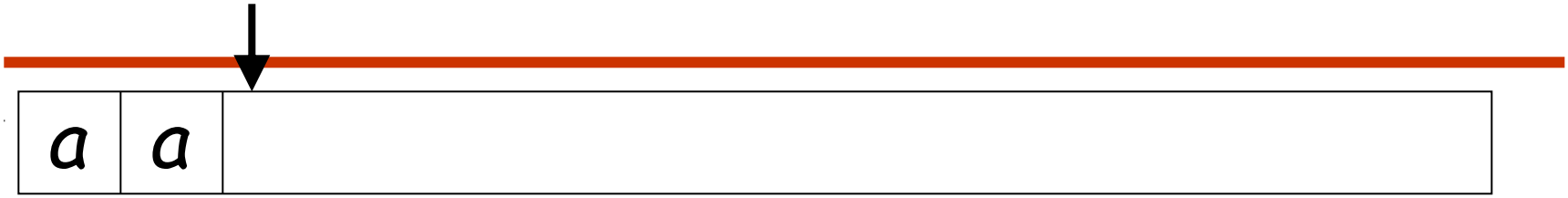

q_0
a
q_1
λ
q_2
a
q_3

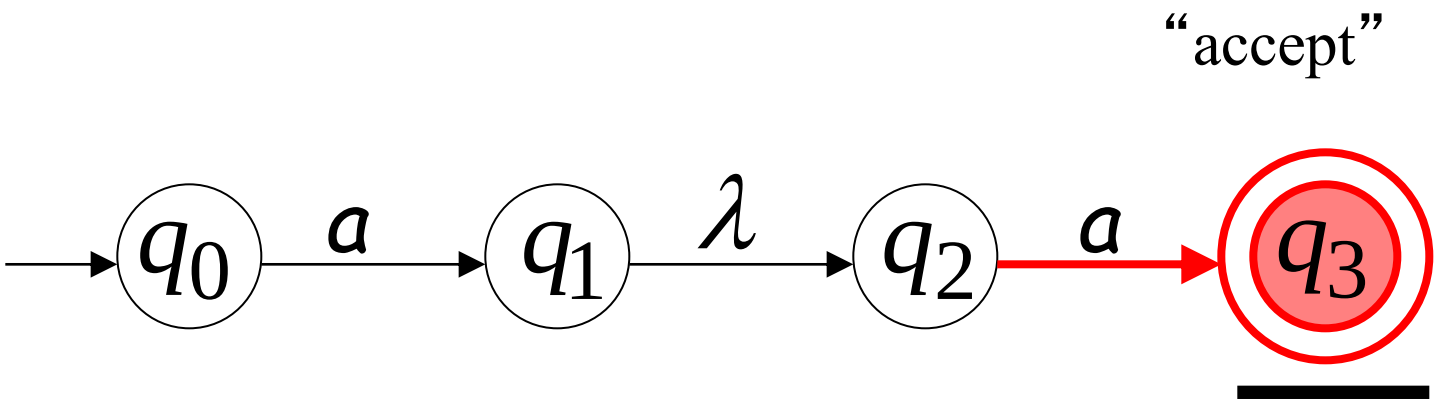
(read head doesn't move)




a
a

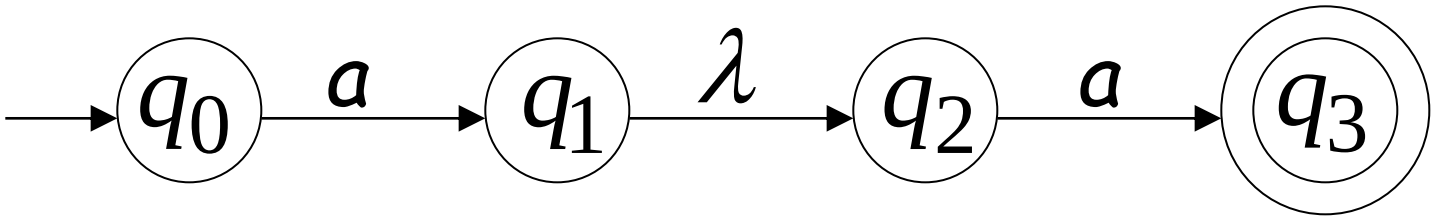

q_0
a
q_1
λ
q_2
a
q_3



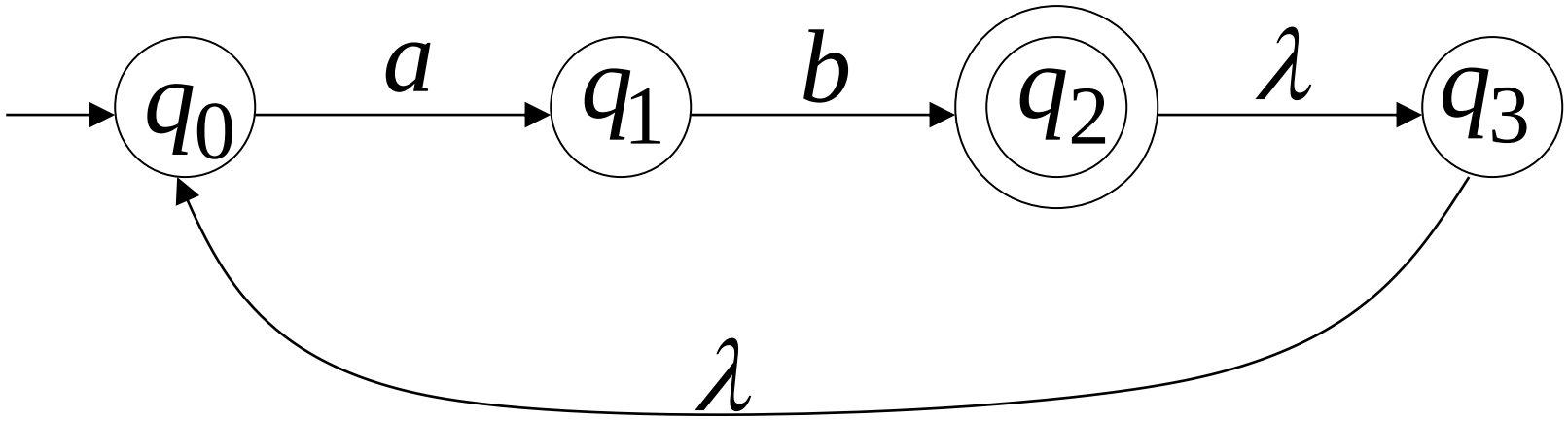


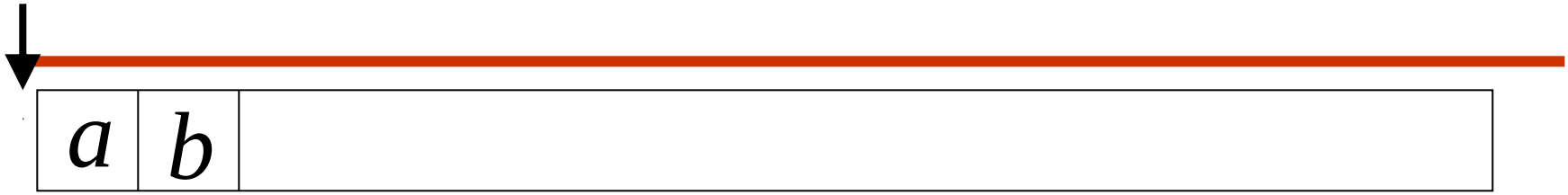
String aa is accepted

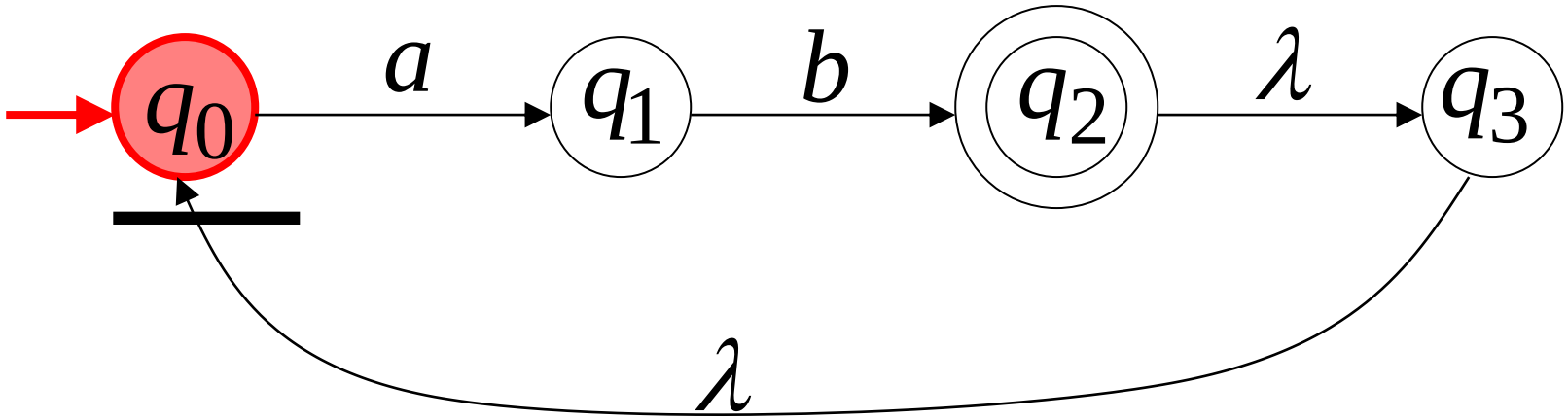
Language accepted: $L = \{aa\}$

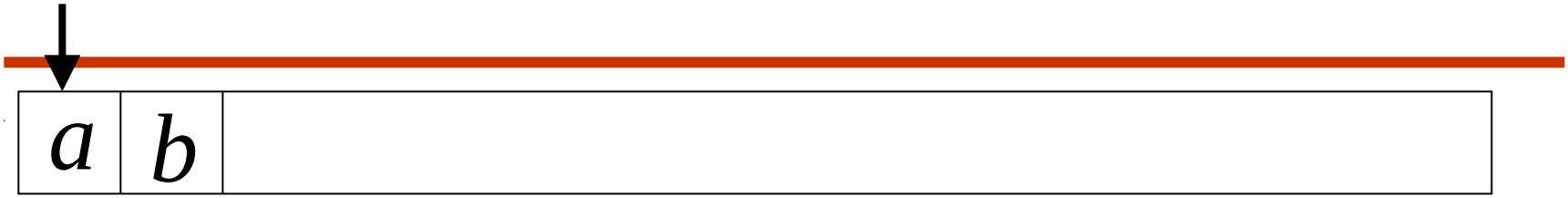


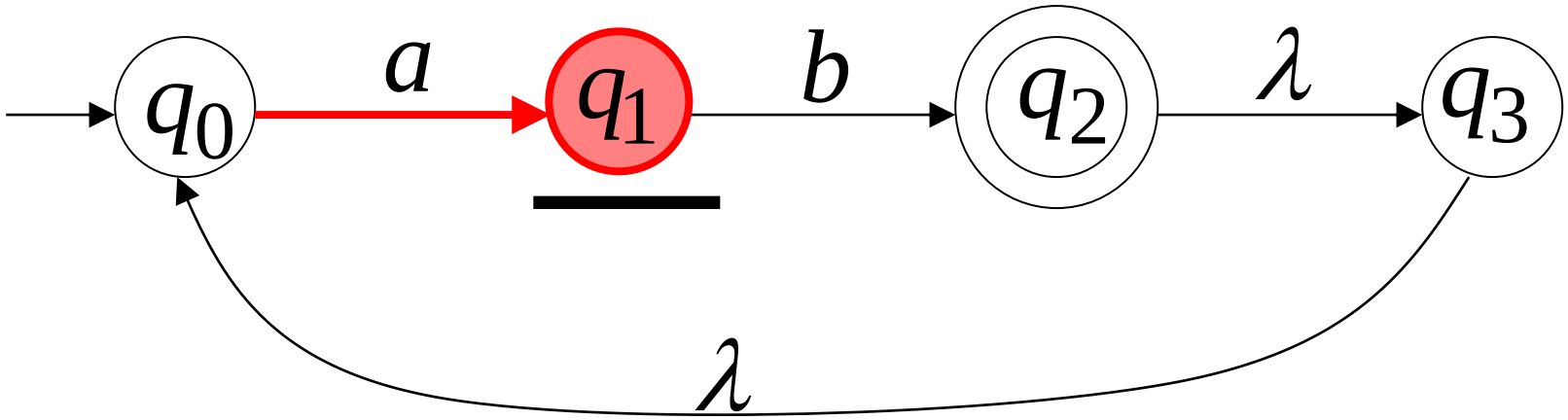
Another NFA Example

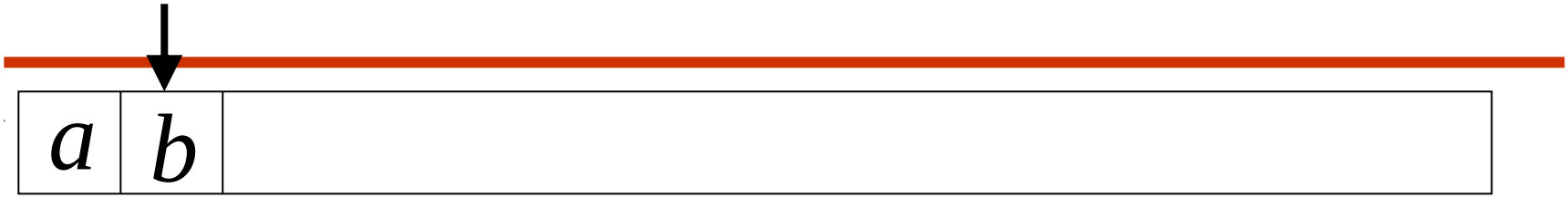


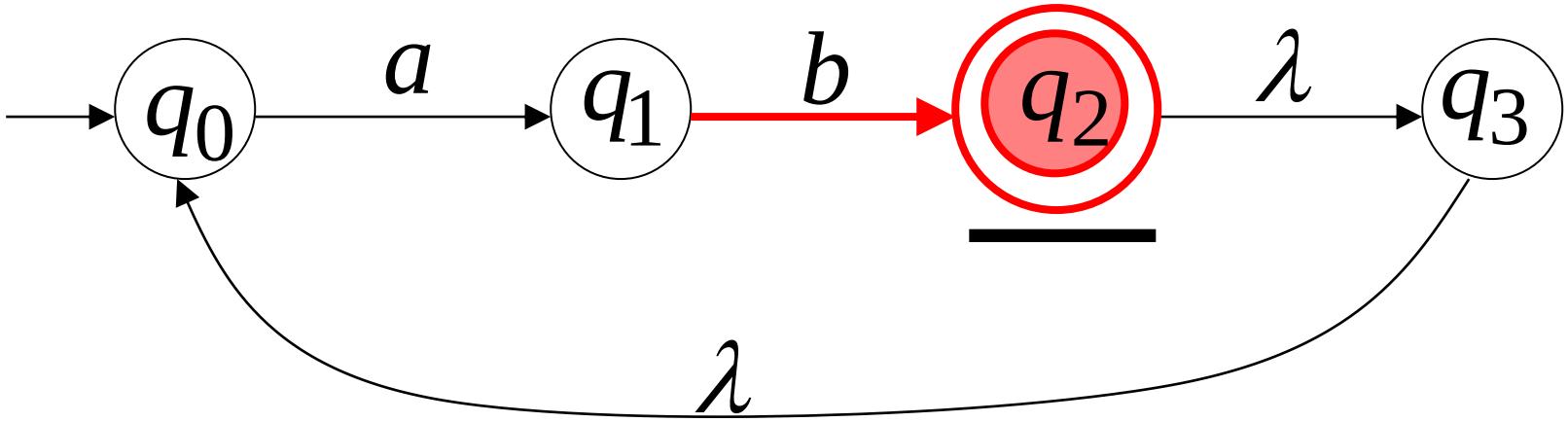

a
b

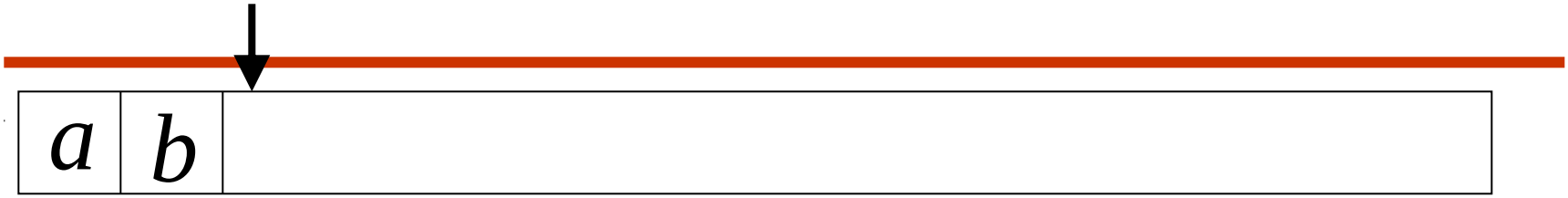

q_0
a
q_1
b
q_2
λ
q_3
λ

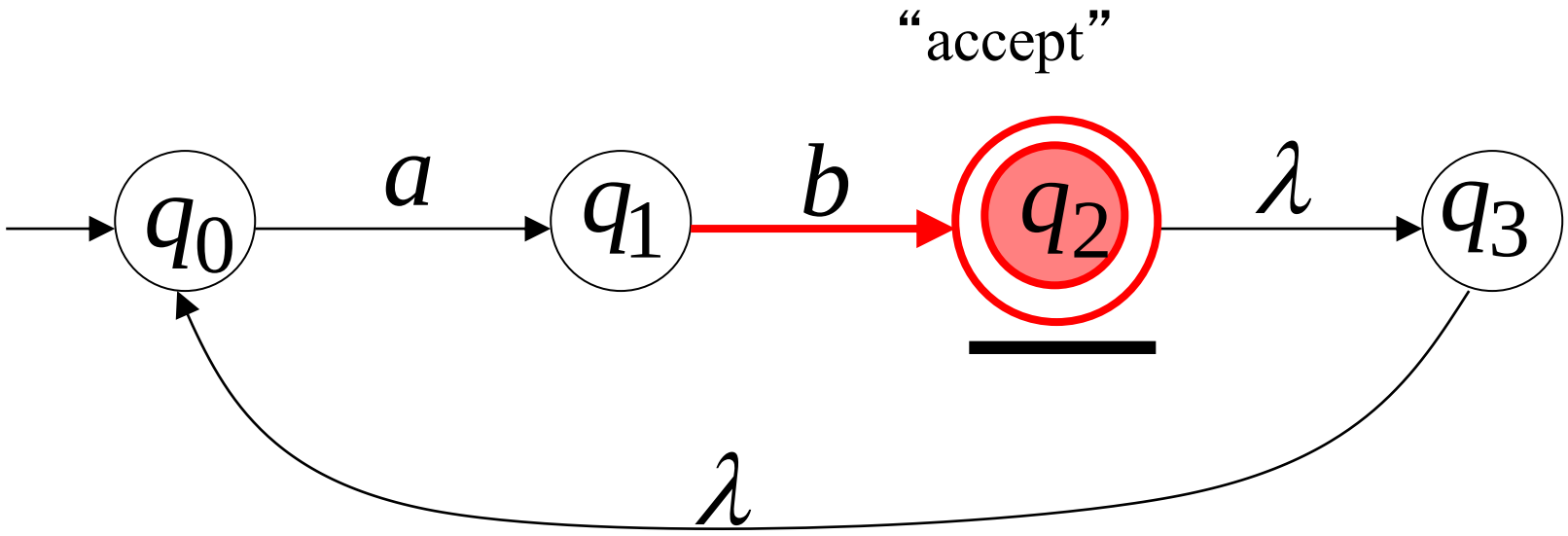

a
b


q_0
a
q_1
b
q_2
λ
q_3
λ

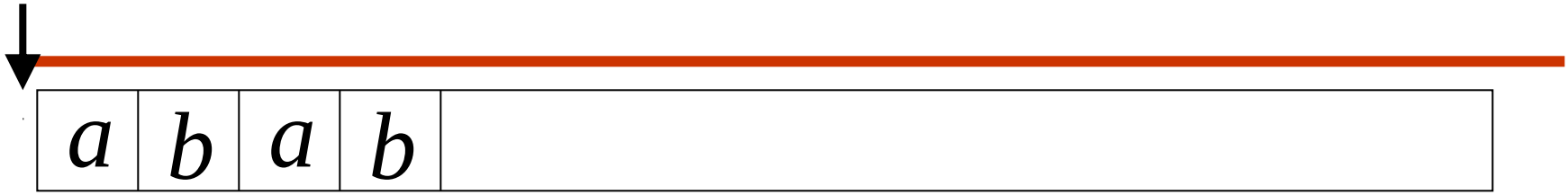

a
b

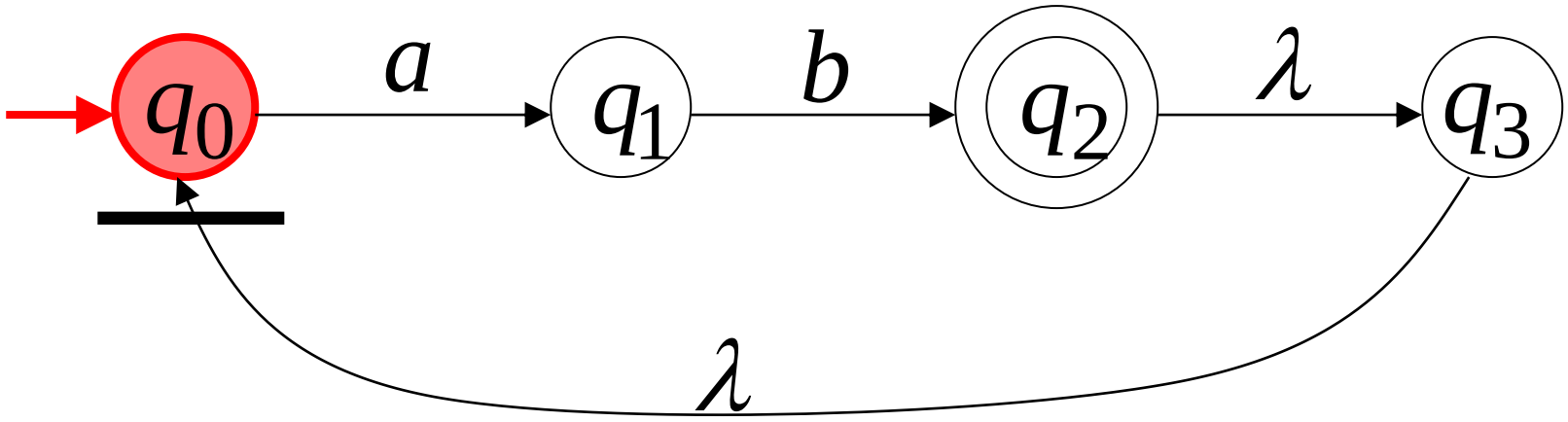

q_0
a
q_1
b
q_2
λ
q_3
λ

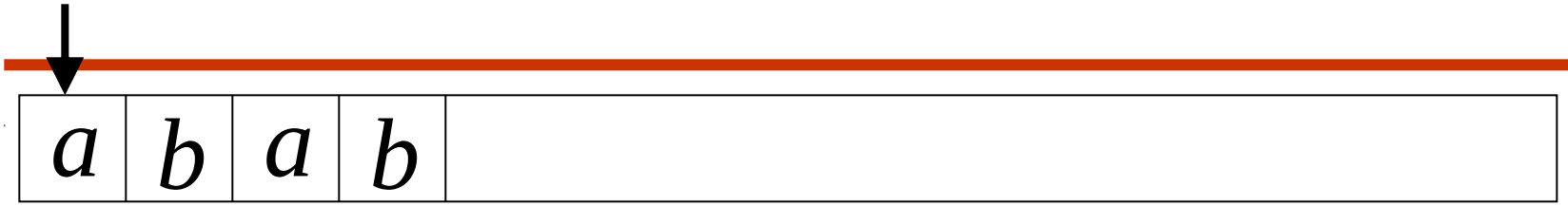

a
b

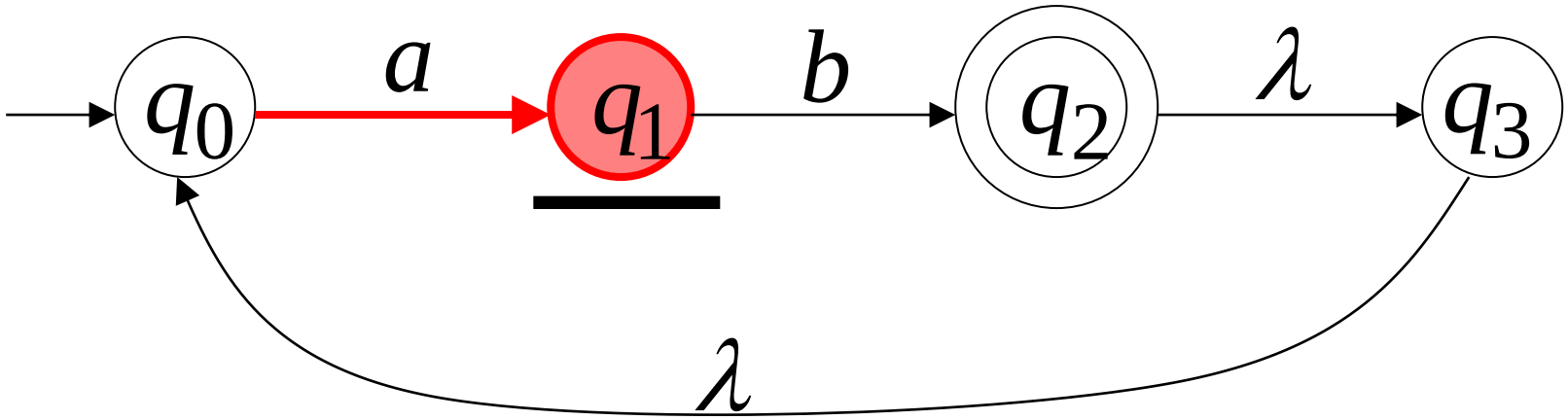

"accept"
q_0
a
q_1
b
q_2
λ
q_3
λ

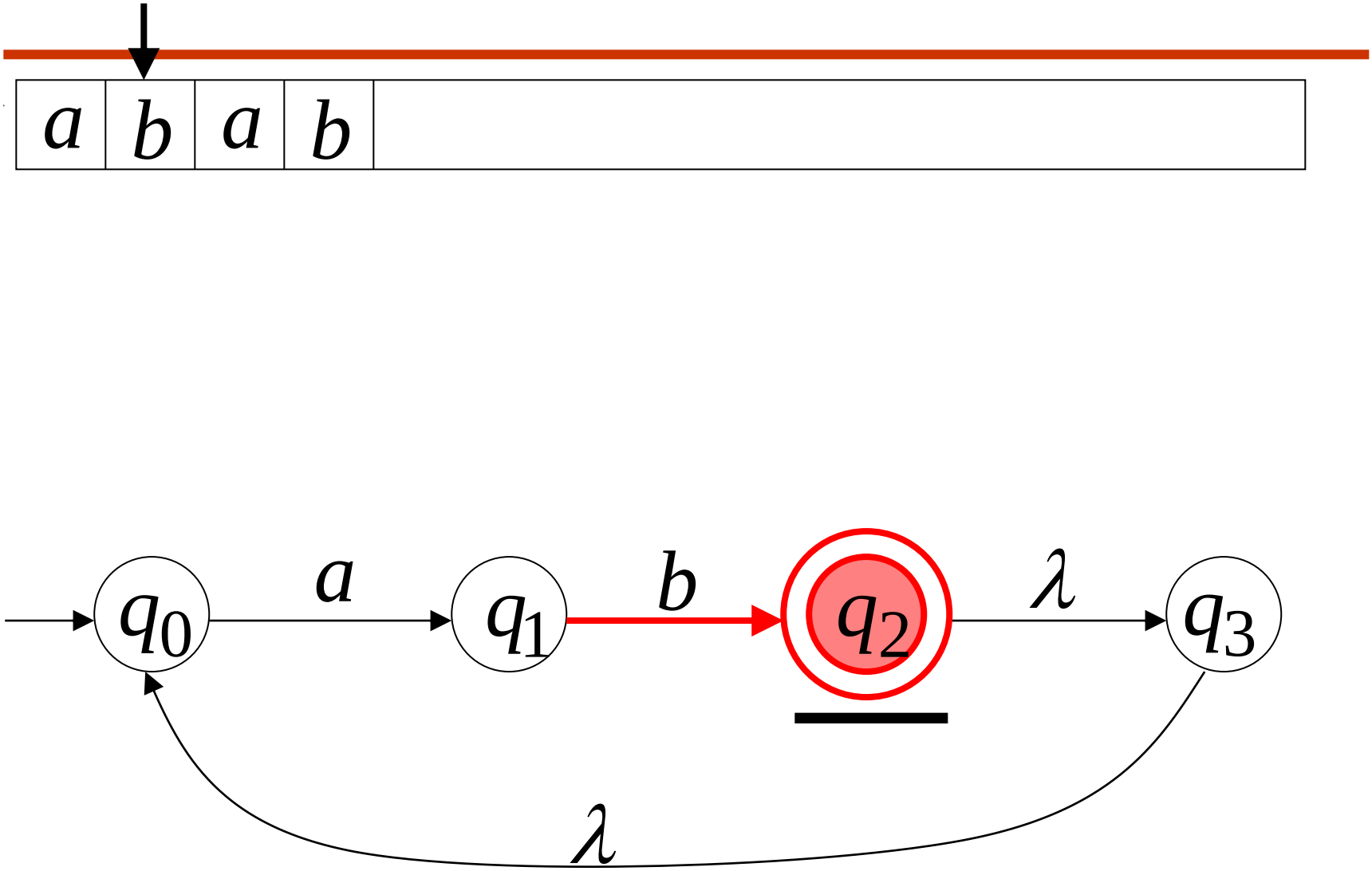
Another String

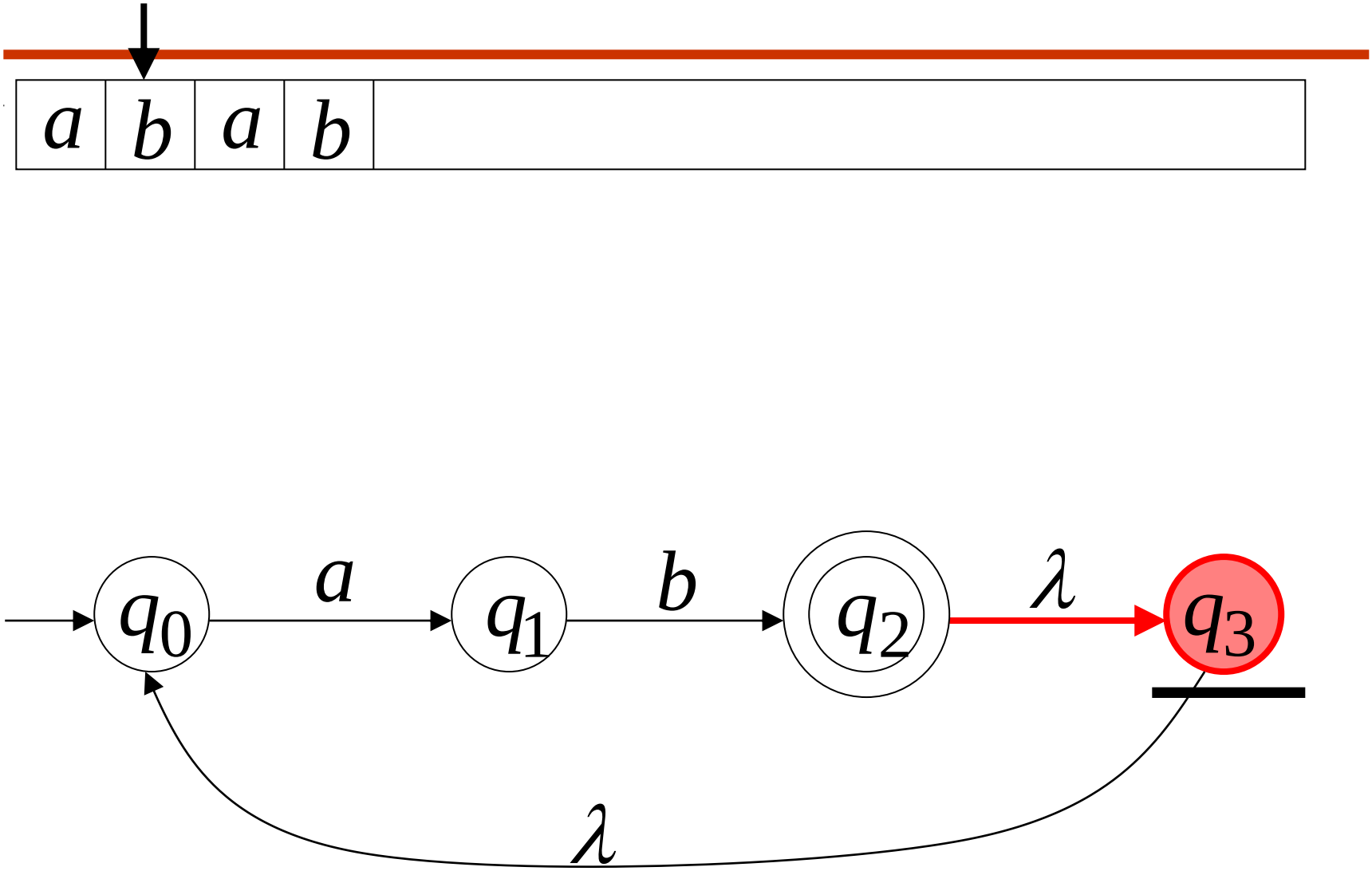


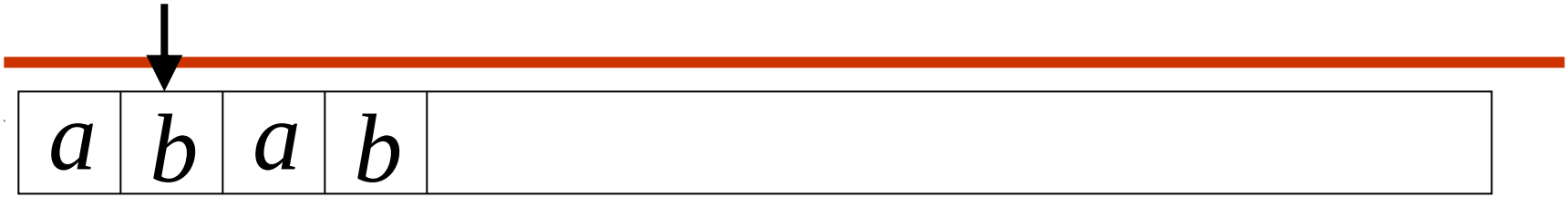


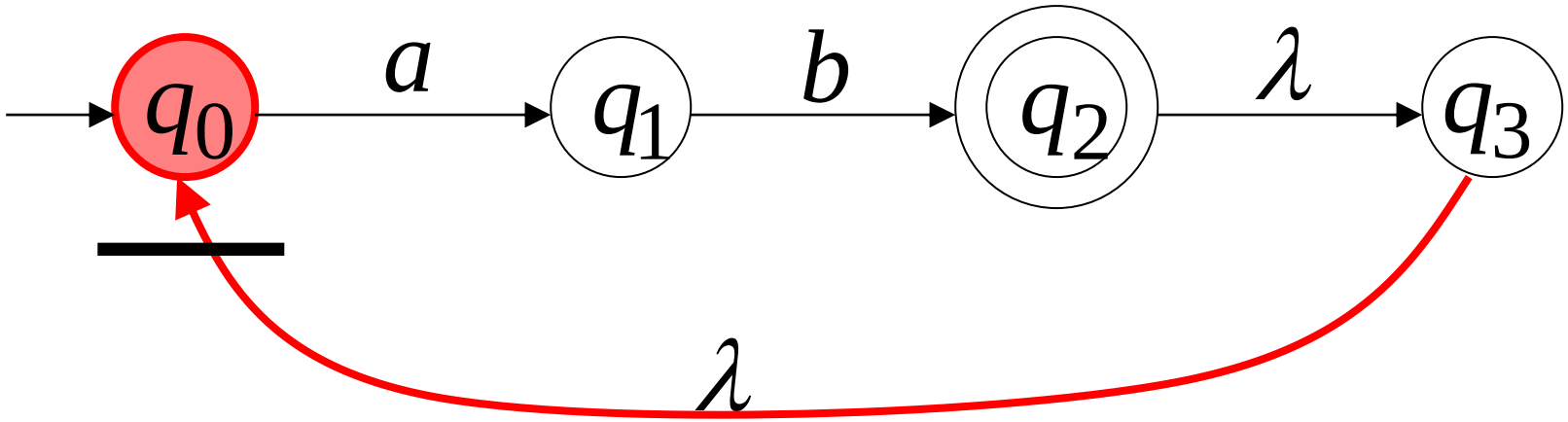

a
b
a
b

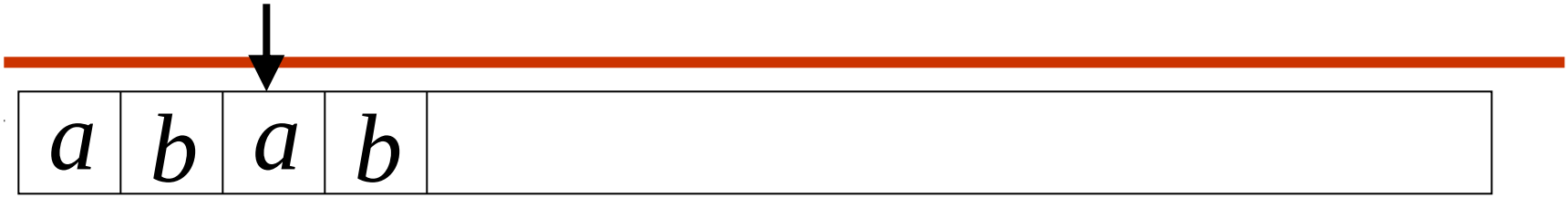

q_0
a
q_1
b
q_2
λ
q_3
λ

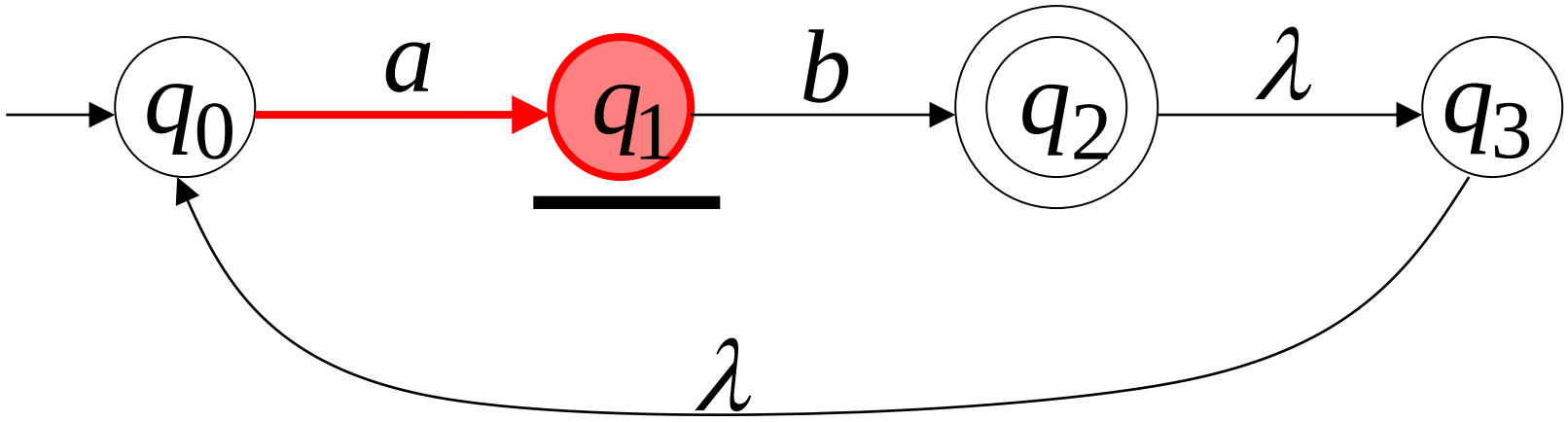

a
b
a
b
q_0
a
q_1
b
q_2
λ
q_3
λ

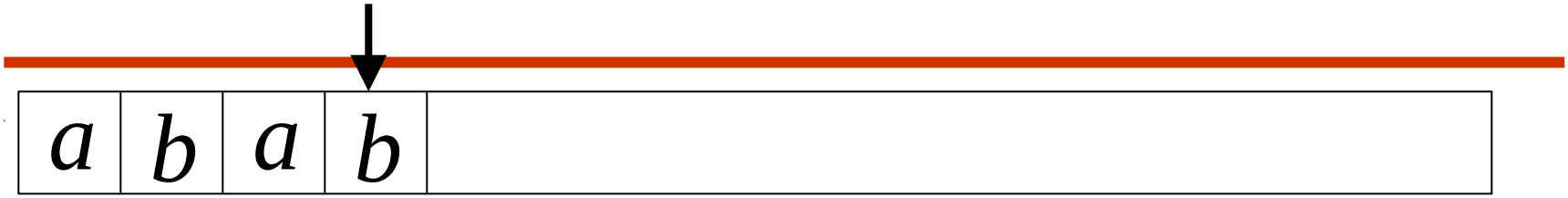

a
b
a
b
q_0
a
q_1
b
q_2
λ
q_3
λ

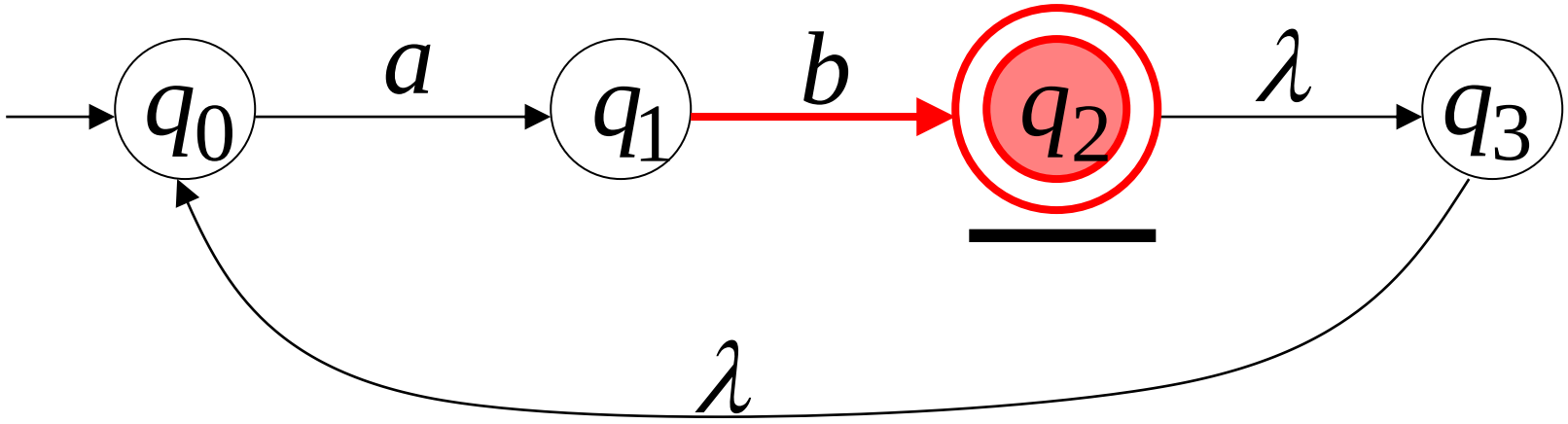

a
b
a
b

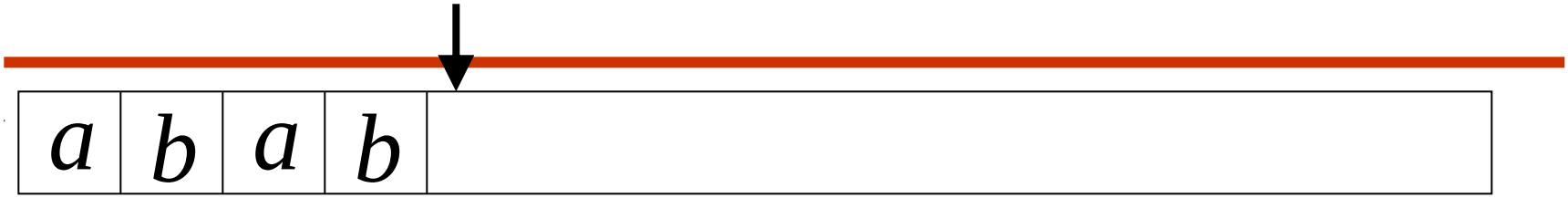

q_0
a
q_1
b
q_2
λ
q_3
λ

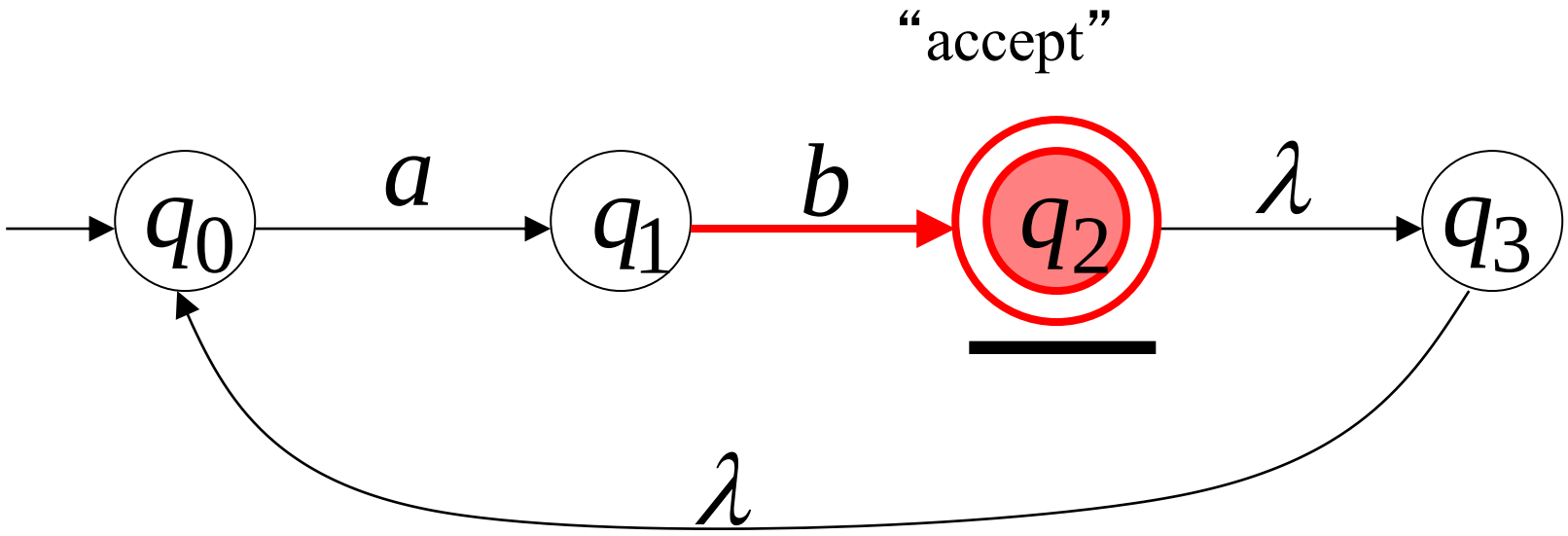

a
b
a
b


q_0
a
q_1
b
q_2
λ
q_3
λ


a
b
a
b


q_0
a
q_1
b
q_2
λ
q_3
λ

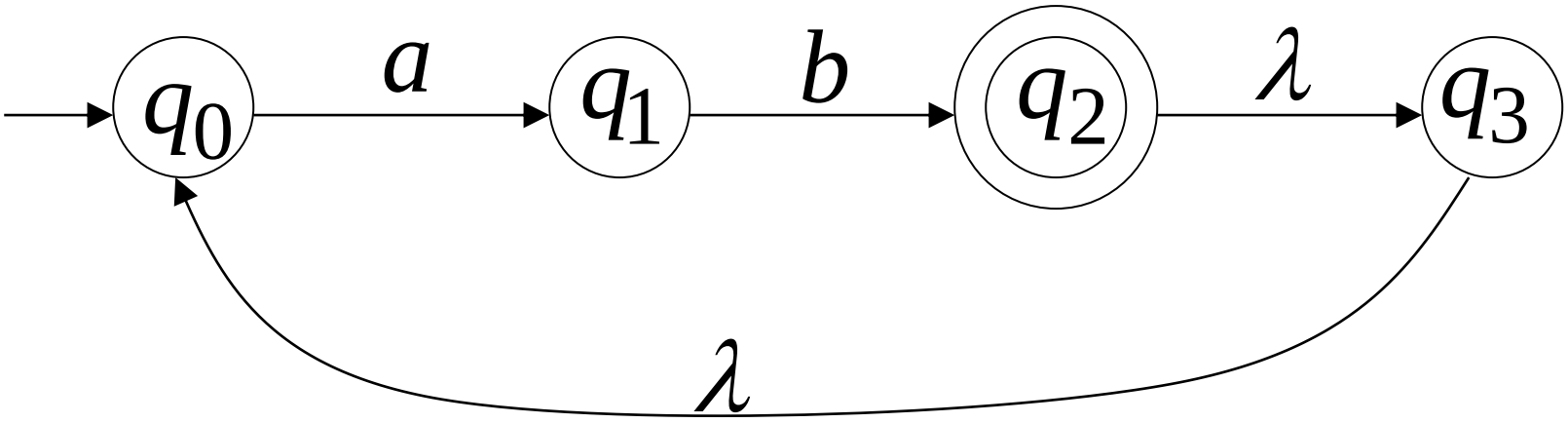

a
b
a
b


"accept"
q_0
q_1
q_2
q_3
a
b
λ
λ

Language Accepted

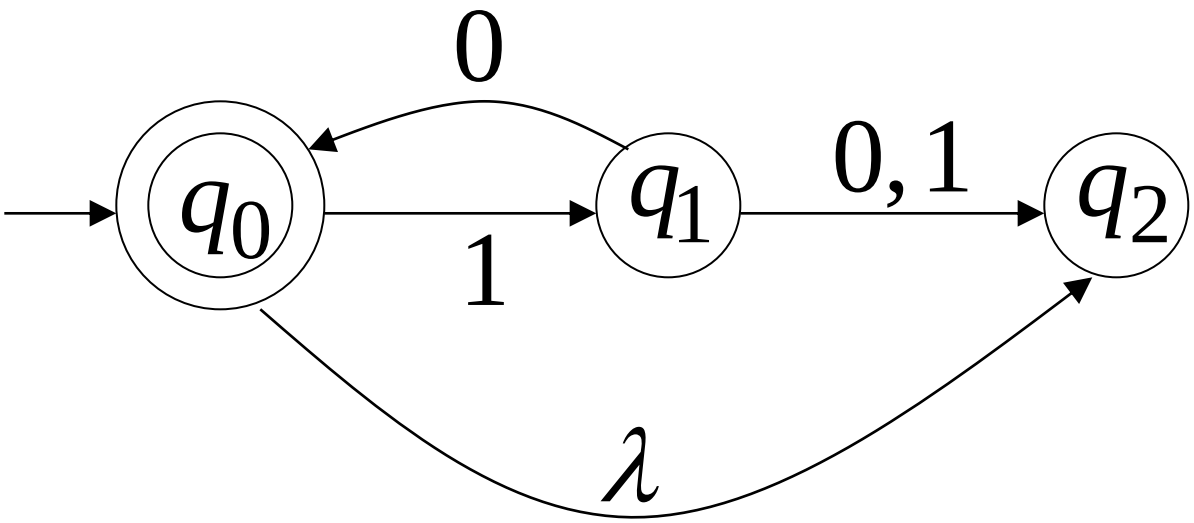
$$L = \{ab,\ abab,\ ababab,\ ...\}$$

$$= \{ab\}^{+}$$



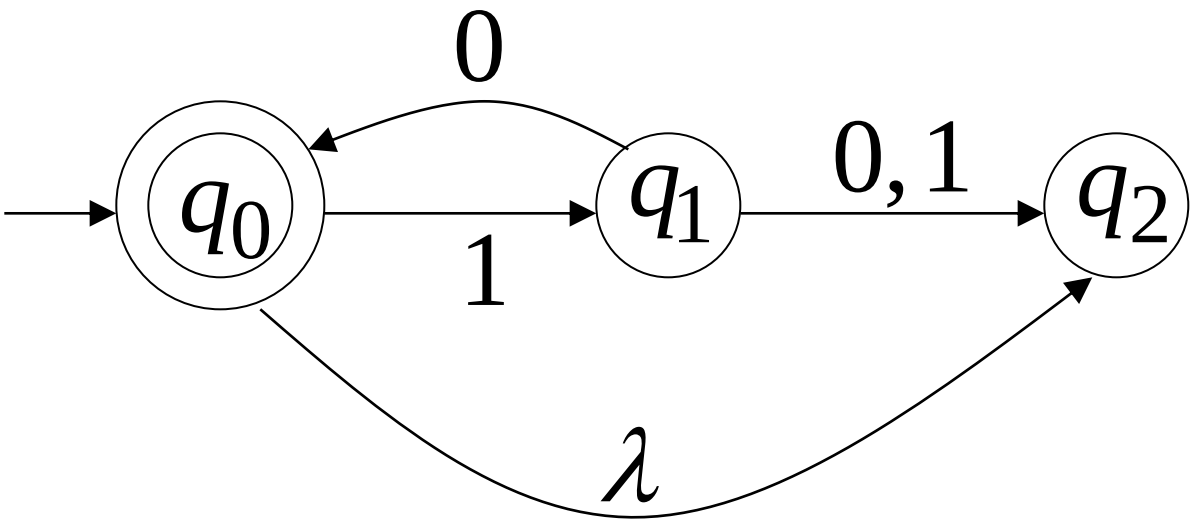
Exercise

1. Identify the features that make it an NFA
2. Determine the set of strings the NFA can recognize



Language accepted

$$L = \{\lambda,\ 10,\ 1010,\ 101010,\ \ldots\}$$
$$= \{10\}^*$$



Formal Definition of NFAs

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

Q: Set of states, e.g., $\{q_0, q_1, q_2\}$

Σ: Input alphabet, e.g., $\{a, b\}$

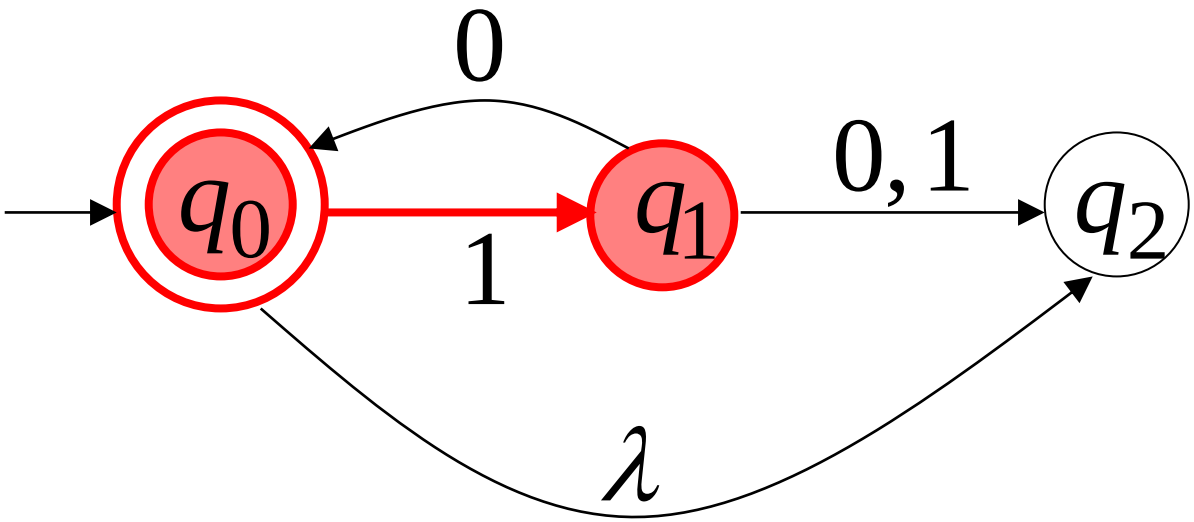
δ: Transition function

q_0: Initial state

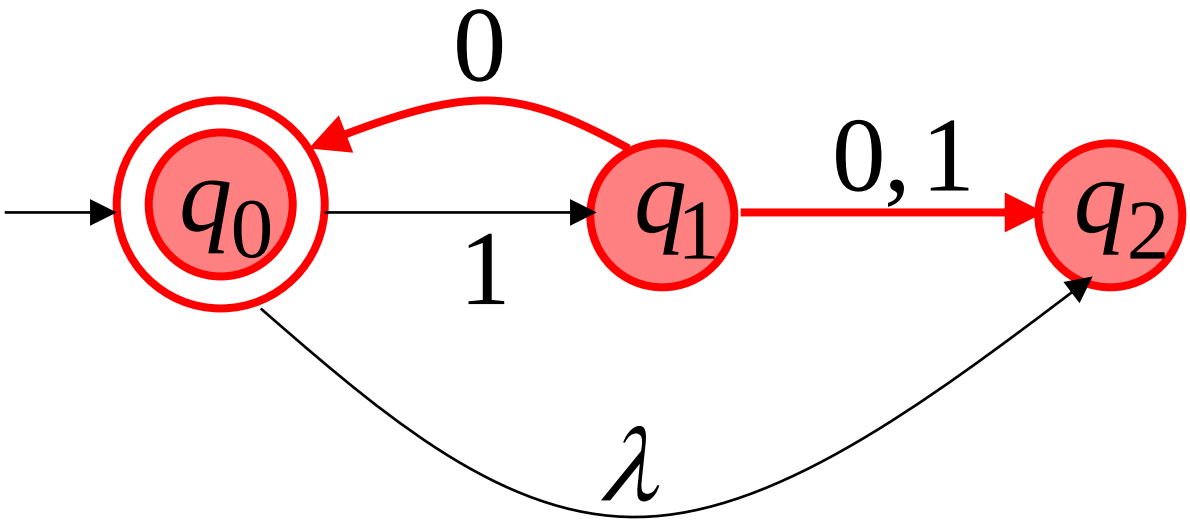
F: Final states

Transition Function δ

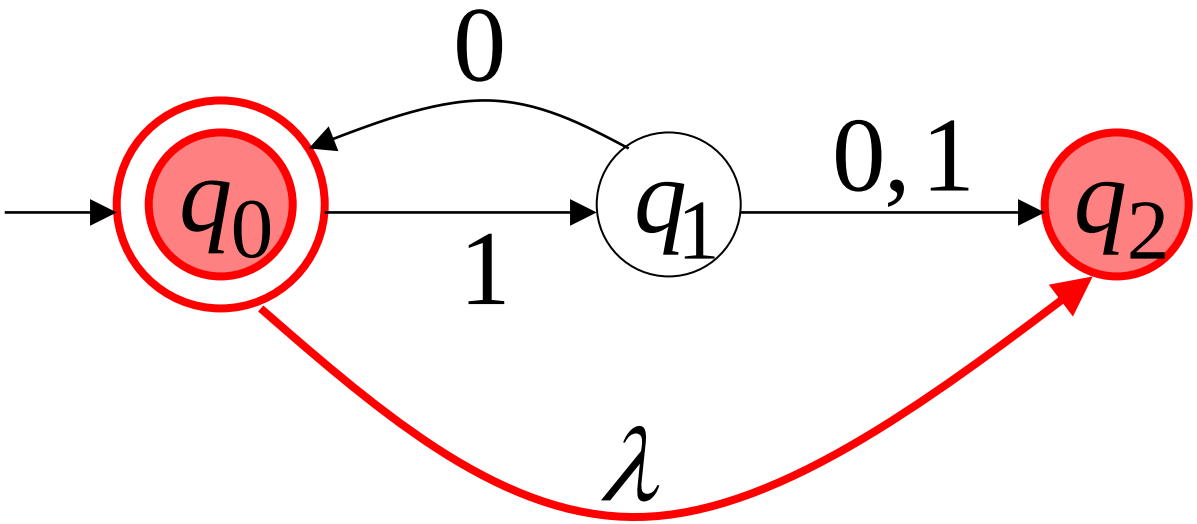
$$\delta(q_0, 1) = \{q_1\}$$



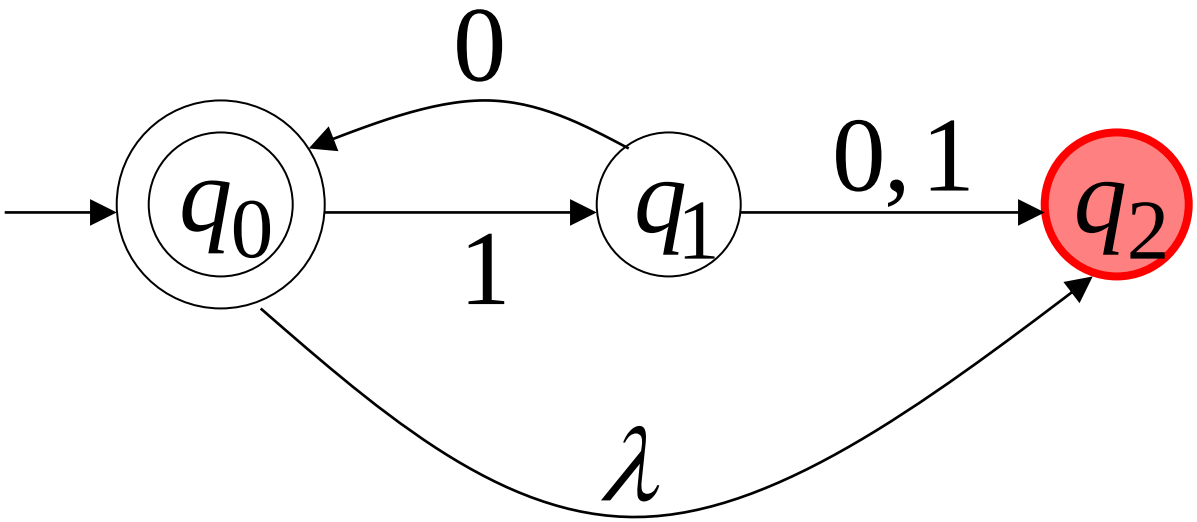
$$\delta(q_1,0)=\{q_0,q_2\}$$



$$\delta(q_0, \lambda) = \{q_0, q_2\}$$



$$\delta(q_2,1)=\varnothing$$



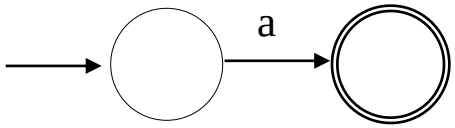
Converting RE to NFA

- Converting a RE to a λ-NFA
  - Inductive construction
  - For each kind of RE, define an equivalent NFA
  - Start with a simple basis, use that to build more complex parts of the NFA

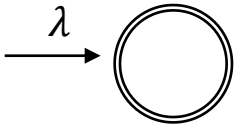
RE to λ–NFA

- Basis:

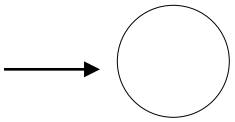
R = a



R = ε

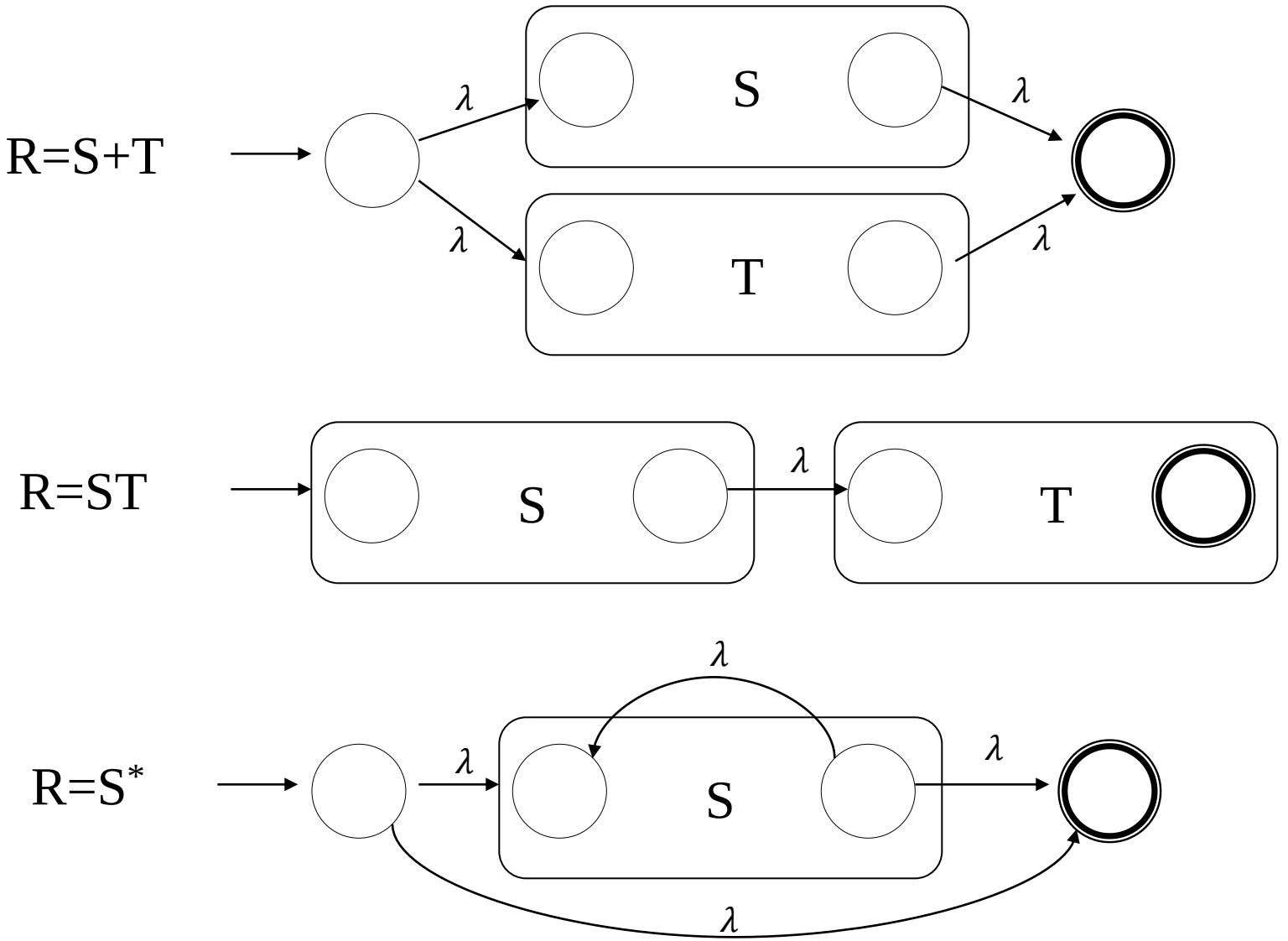


R = Ø



Next slide: More complex RE's

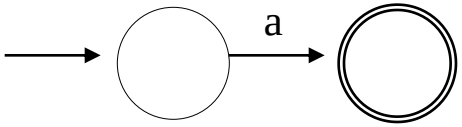
Compound RE to λ–NFA



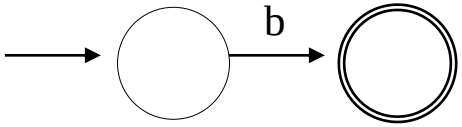
RE to λ-NFA Example

- Convert R= (ab+a)* to an NFA
  - We proceed in stages, starting from simple elements and working our way up

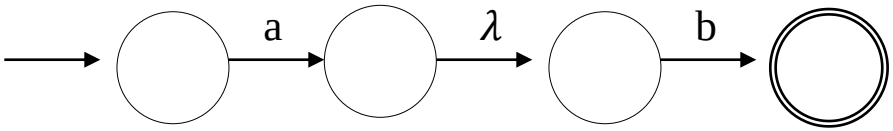
a



b

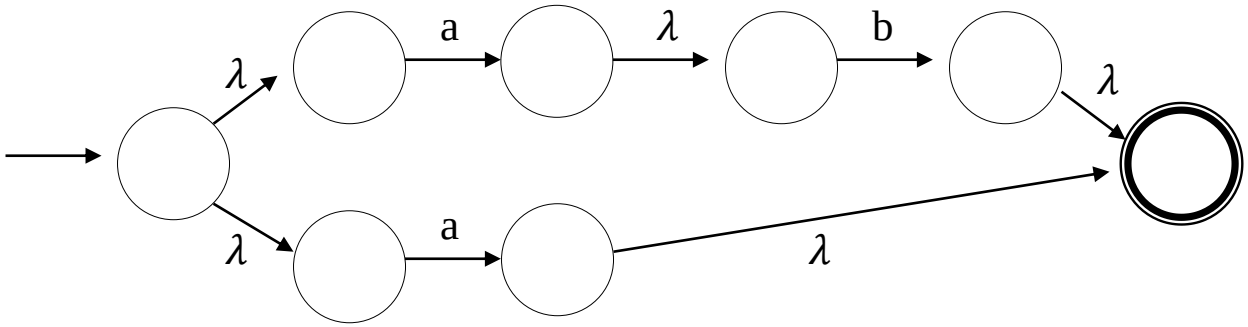


ab

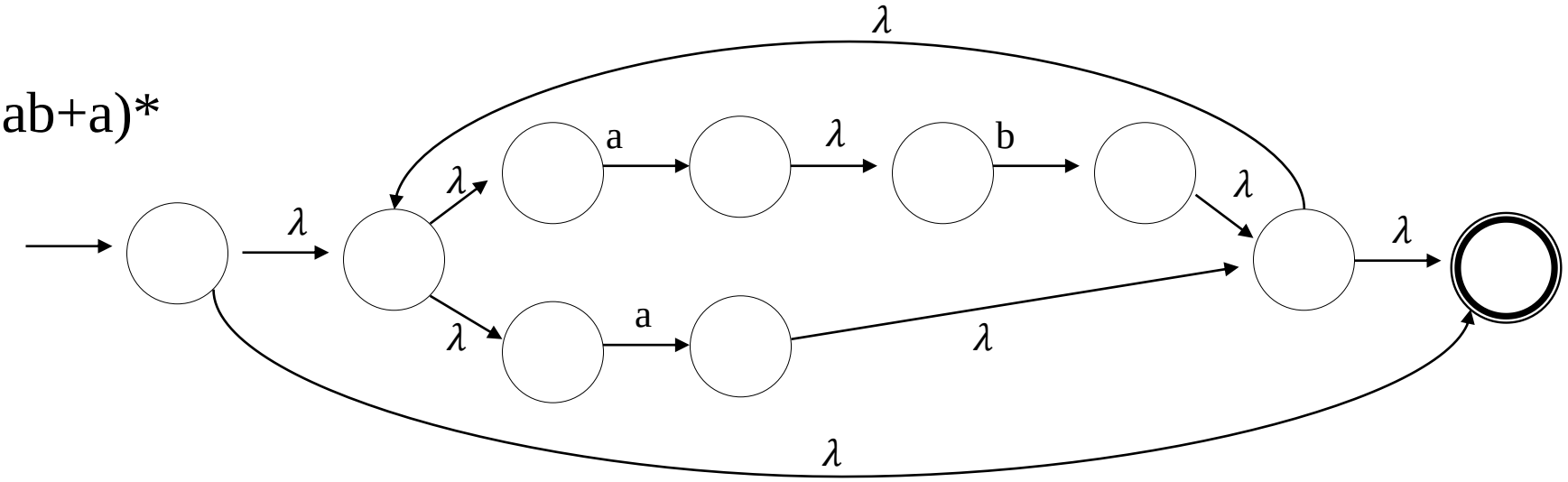


RE to ϵ-NFA Example (2)

ab+a



(ab+a)*



In-class Exercise

Convert the regular expression (1+0)*1 to a NFA that accepts the same language

Topic Map

Two Types of Finite Automata

- Nondeterministic Finite Automaton (NFA)
- Deterministic Finite Automaton (DFA)
  - For each input symbol, one can determine the state to which the machine will move

DFA vs NFA

- Deterministic vs nondeterministic
  - For every nondeterministic automaton, there is an equivalent deterministic automaton
  - Finite automata are equivalent iff they both accept the same language

$$L(M_1) = L(M_2)$$

DFA vs NFA

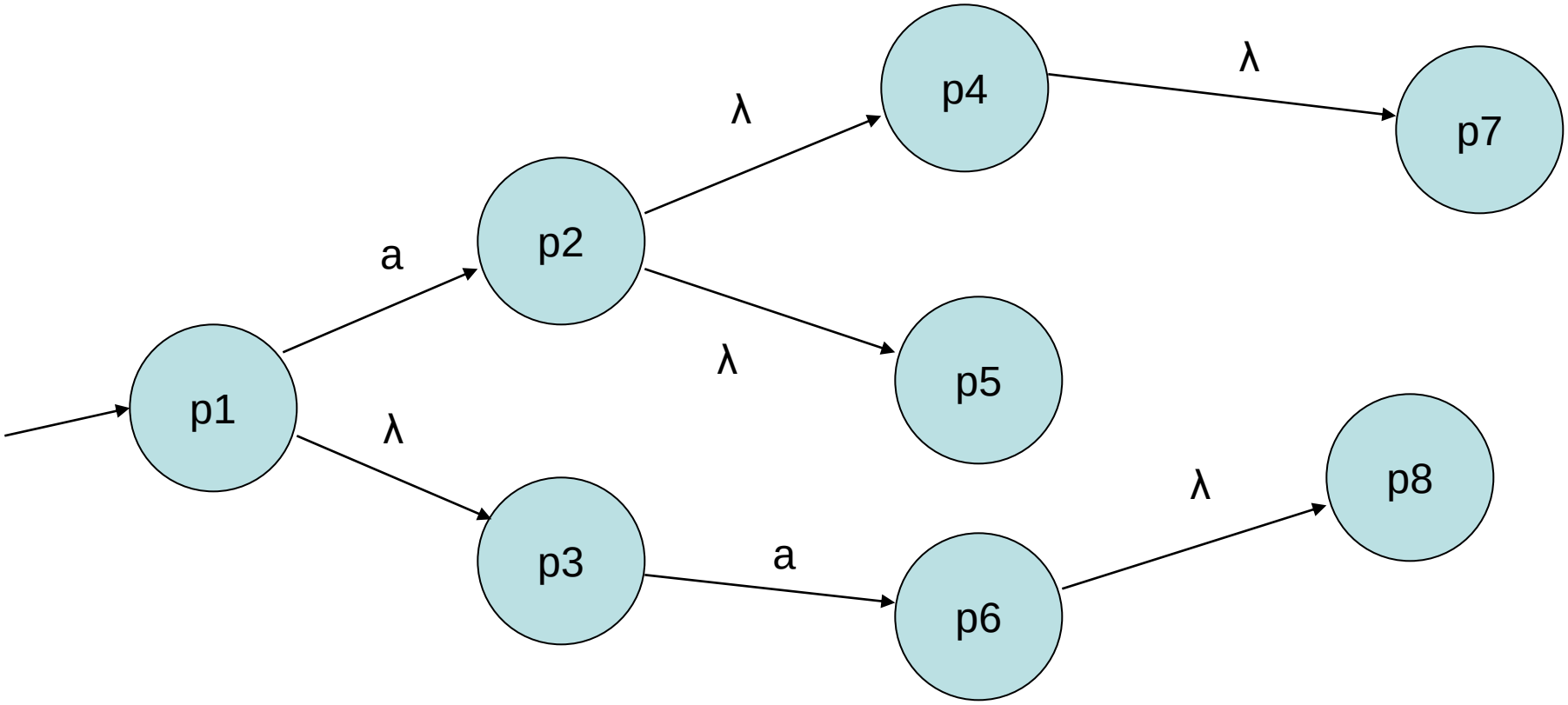
- Deterministic vs nondeterministic
  - In DFA, label resultant state as a set of states drawn from NFA
    - {q1, q2, q3,…}
  - For a set of |Q| states, there are exactly 2^Q subsets
    - Finite number of states

Removing Nondeterminism

- By simulating all moves of an NFA-λ (an NFA with λ transitions) *in parallel* using a DFA

- *λ-closure* of a state is the set of states reachable using only the λ-transitions

NFA–λ



λ – Closure

Selected λ closures

p_1: {p_1, p_3}

p_2: {p_2, p_4, p_5, p_7}

$$\delta(p1, a) = \{p_2, p_4, p_5, p_7, p_6, p_8\}$$

Equivalence Construction

- Goal: given an NFA–λ M_1, construct a DFA M_2 such that $\mathcal{L}(M_1) = \mathcal{L}(M_2)$
- Observe that
  - **A node of the DFA = a set of nodes of NFA–λ**
  - **Transition of the DFA = transition among a set of nodes of NFA–λ**

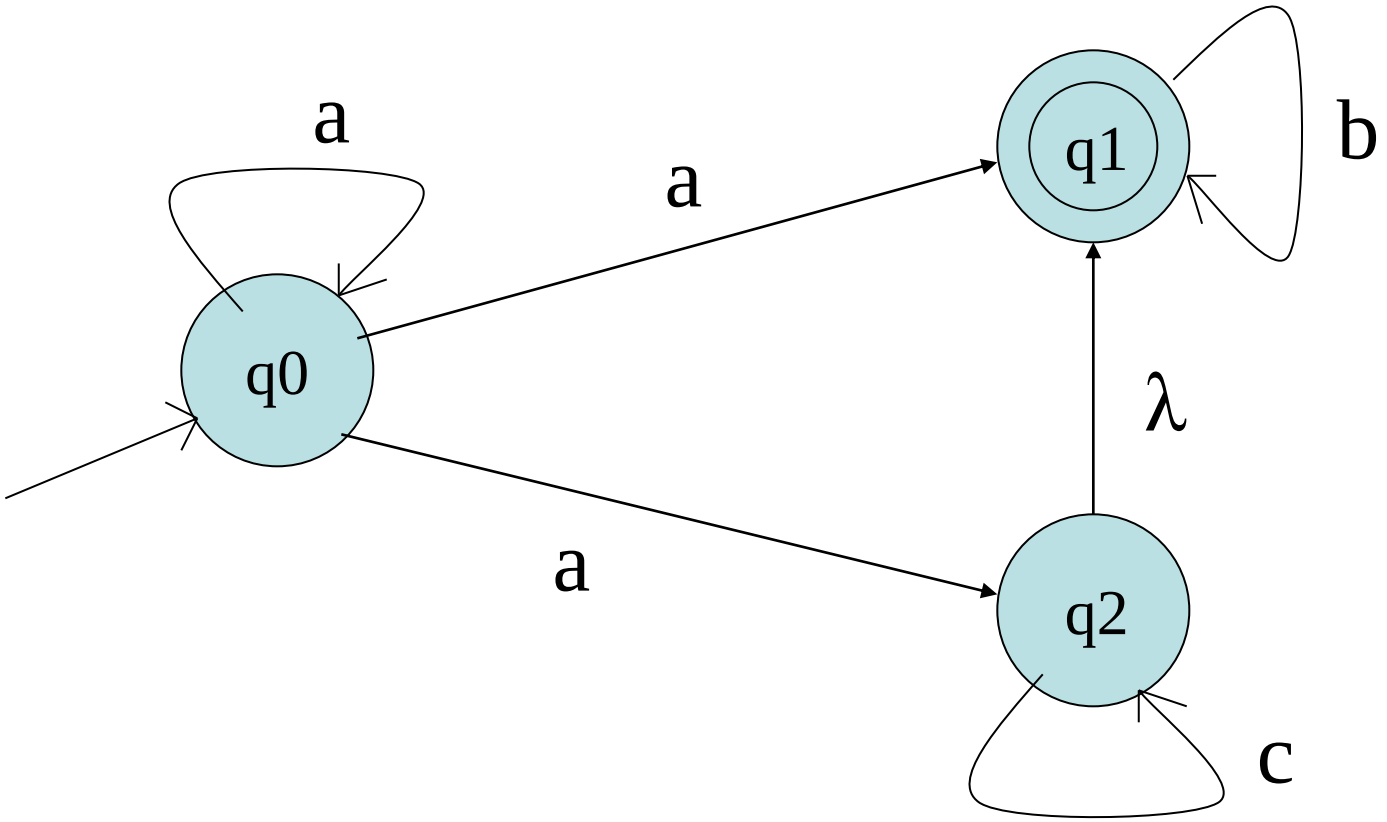
Special States to Identify

Start state of DFA = λ - $closure(\{q_0\})$

Final/Accepting state of DFA = All subsets of states of NFA-λ that contain an accepting state of the NFA-λ

Dead state of DFA = ϕ

Example



Example

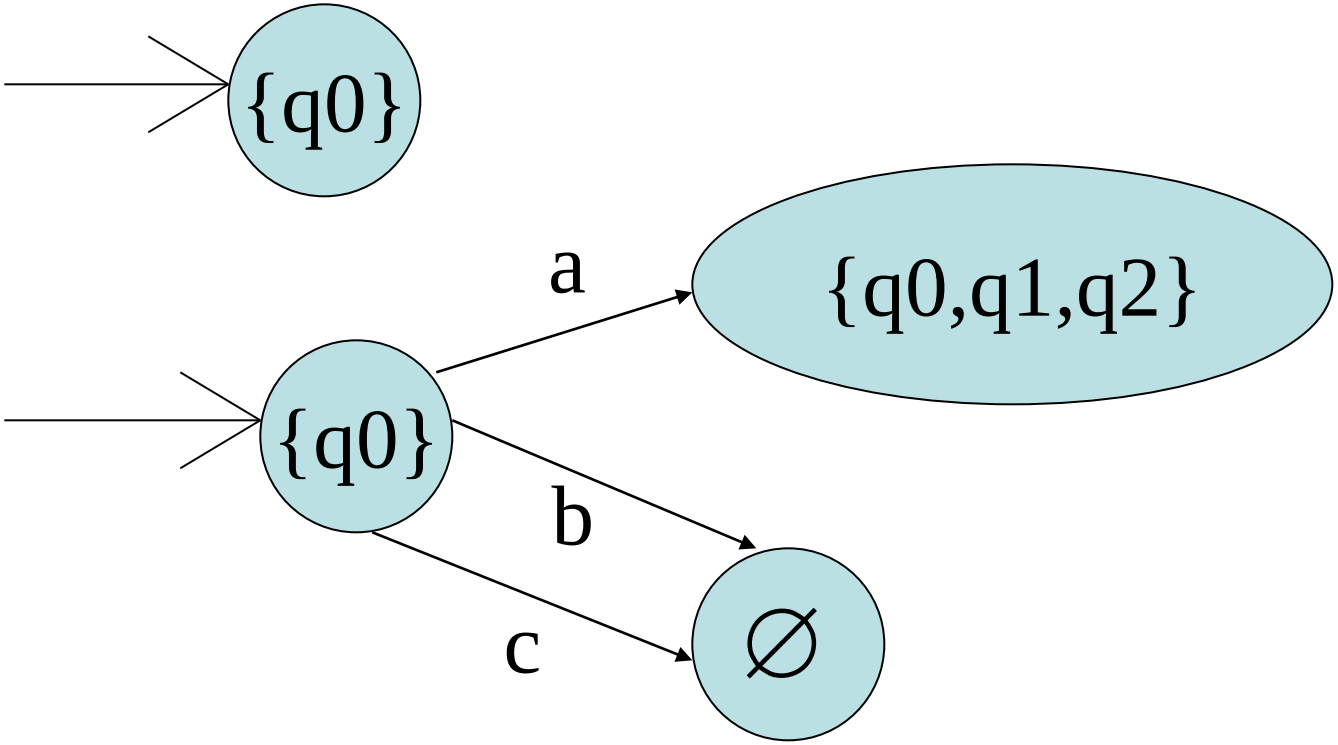
- Identify λ-closures
  - q_0: $\{q_0\}$
  - q_1: $\{q_1\}$
  - q_2: $\{q_1, q_2\}$

Example

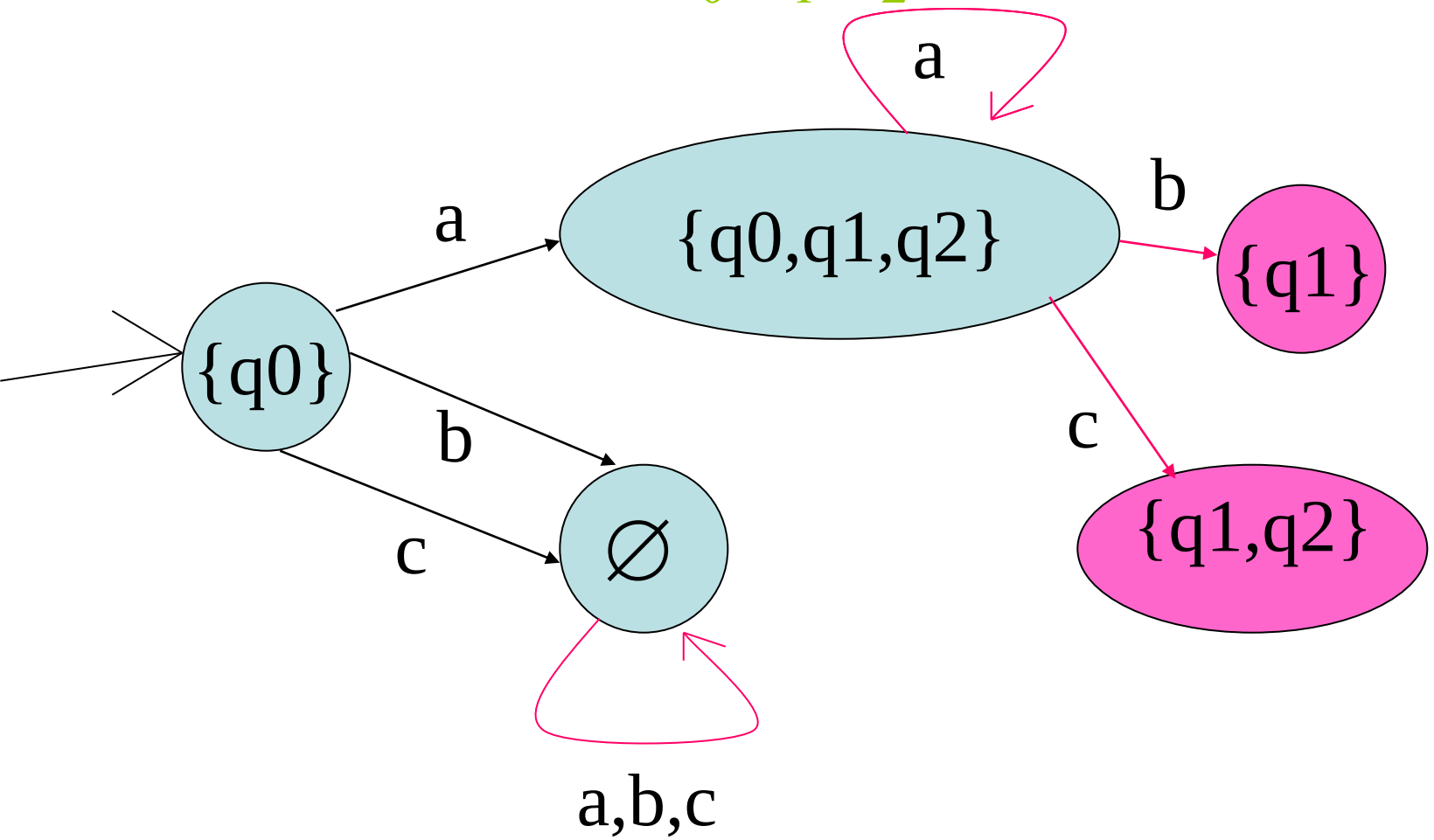
Identify transitions

- Start with λ-closure of start state
- $\{q_0\}$: Where can you go on each input?
  - a: {q0,q1,q2}
    - So, {q0,q1,q2} is a state in the DFA
  - b, c: Nowhere, so {Φ} is in the DFA
- Next, do the same for $\{q_0, q_1, q_2\}$ and {Φ}
  - Find destinations from any node in the set for each of the three alphabet symbols

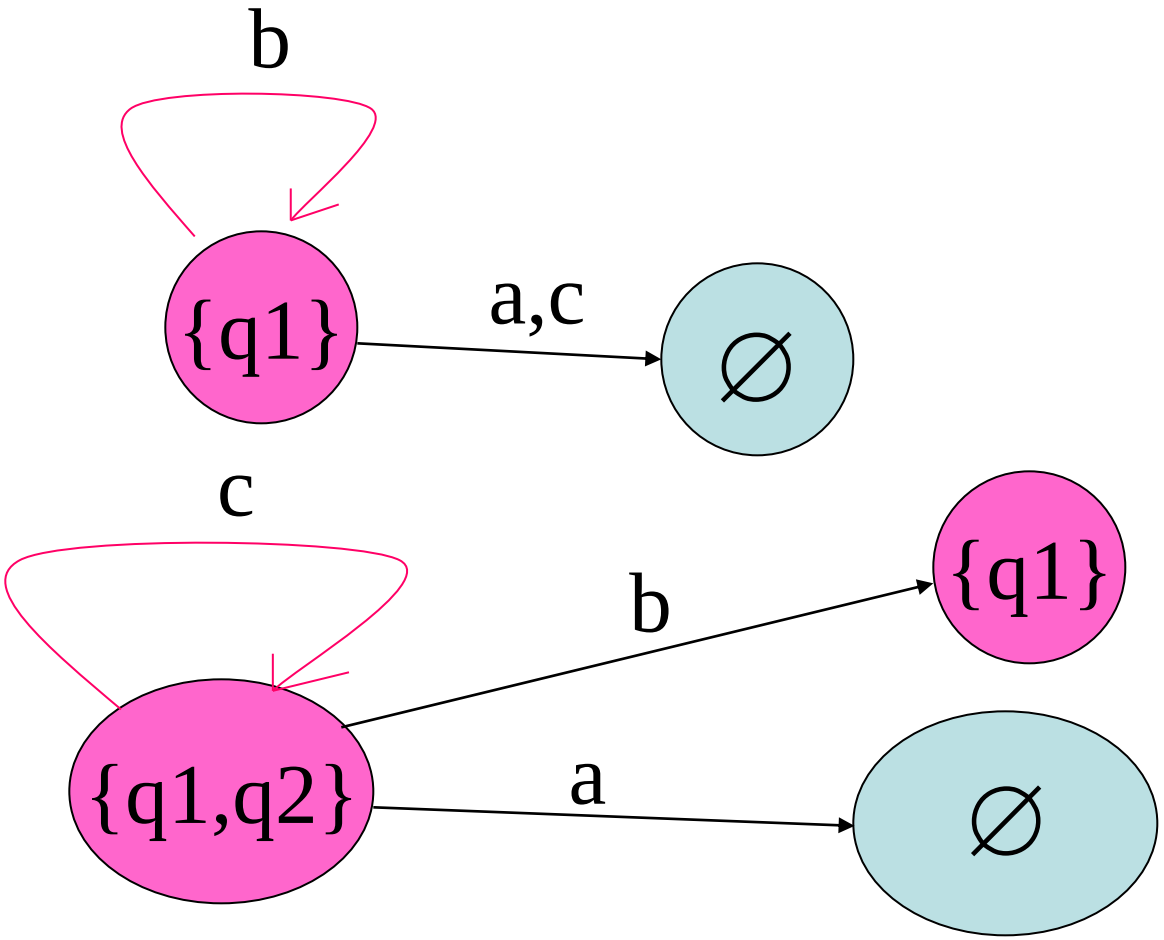
All steps from $\{q_0\}$



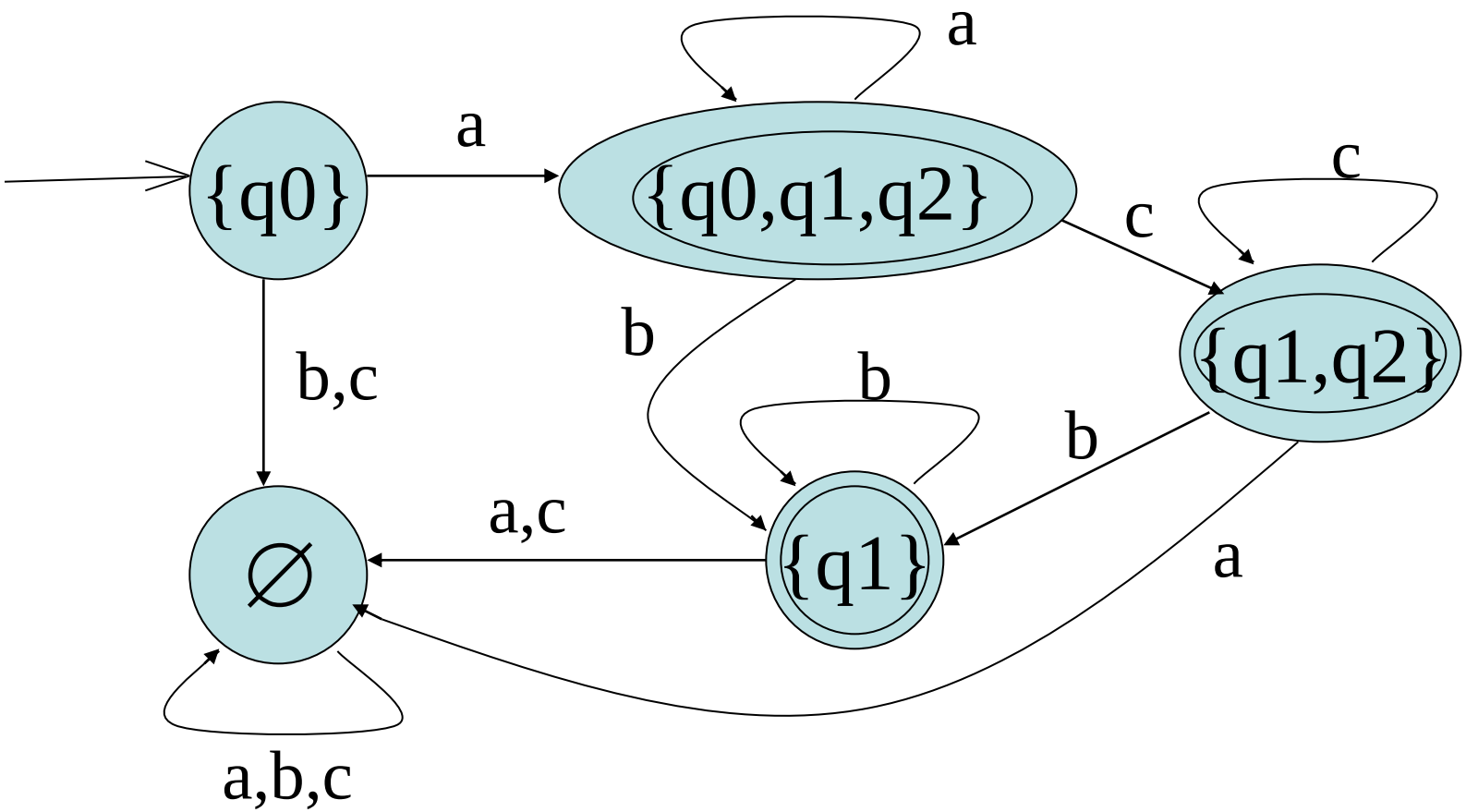
All steps from $\{q_0,q_1,q_2\}$



All steps from $\{q_1\}$ and $\{q_1,q_2\}$

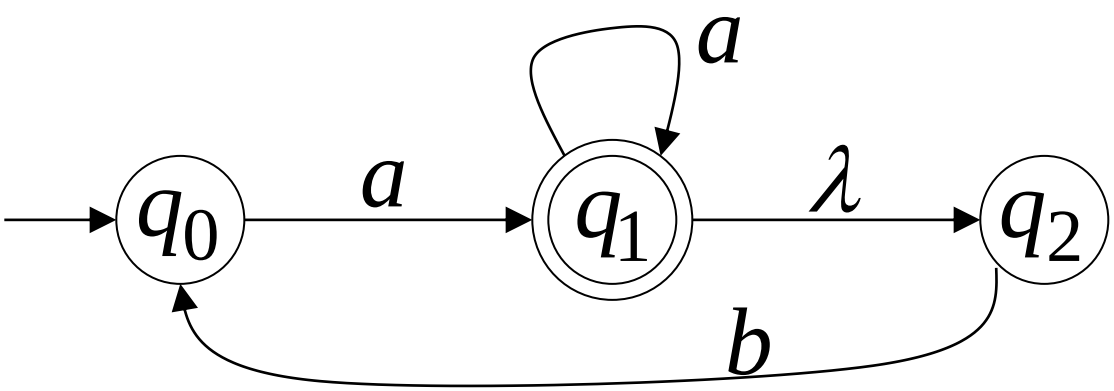


Equivalent DFA



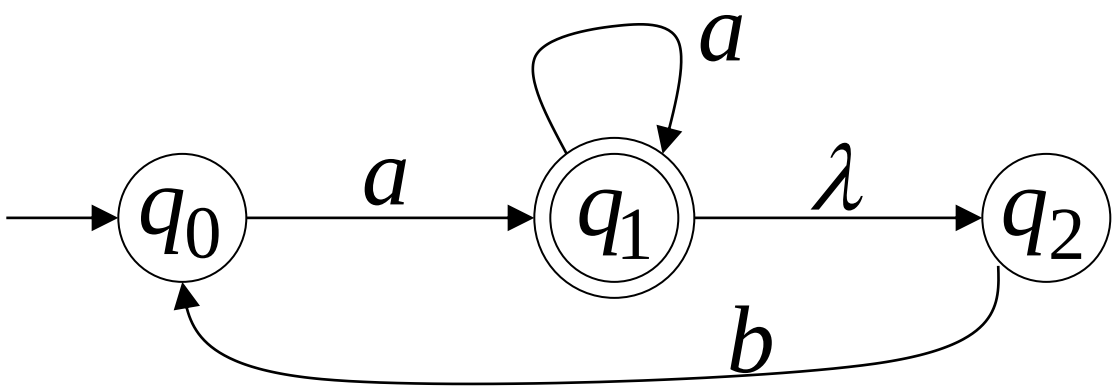
Exercise: Convert this NFA

NFA

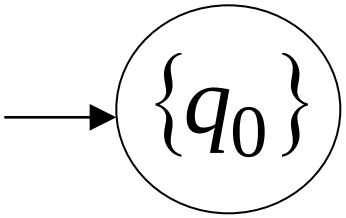


NFA to DFA

NFA

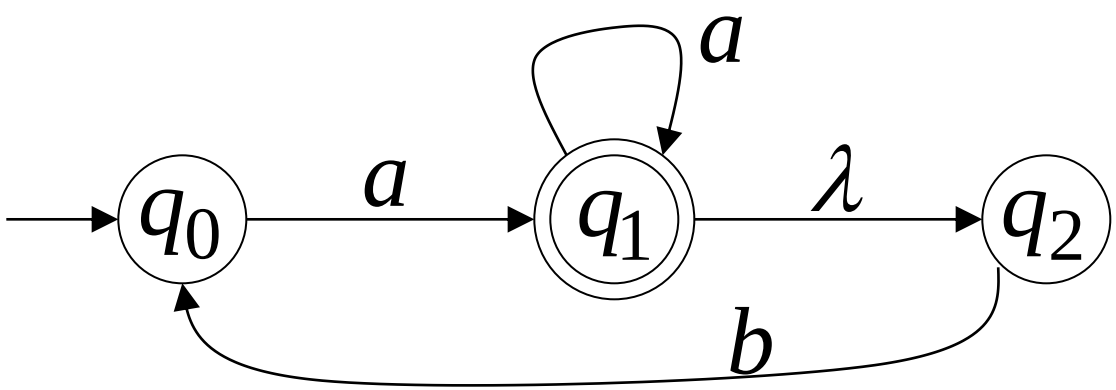


DFA

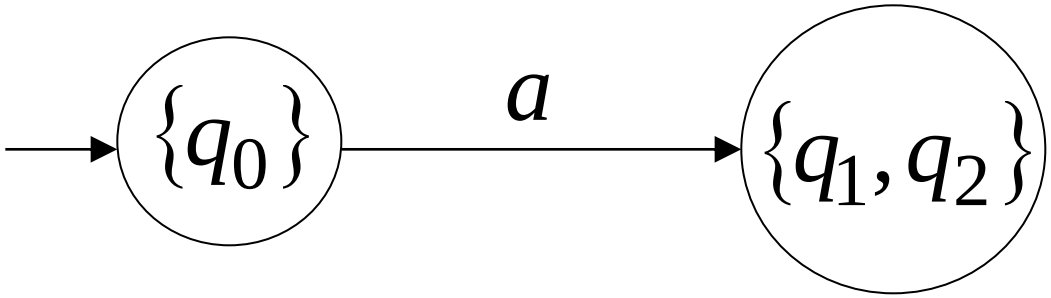


NFA to DFA

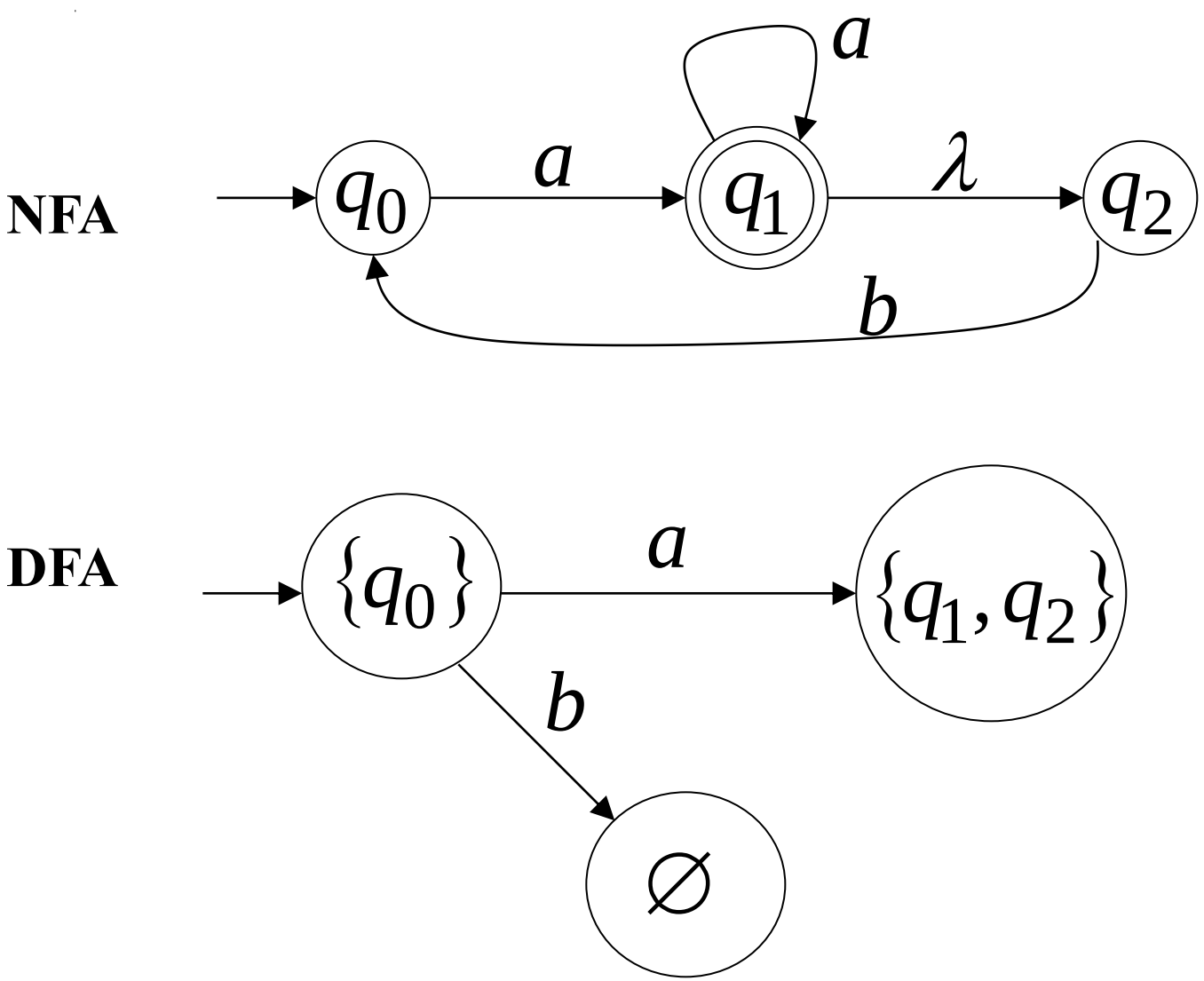
NFA



DFA

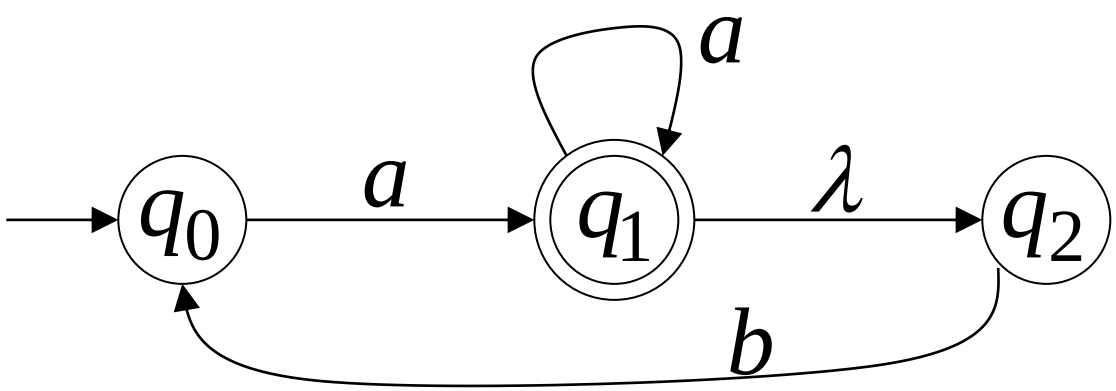


NFA to DFA

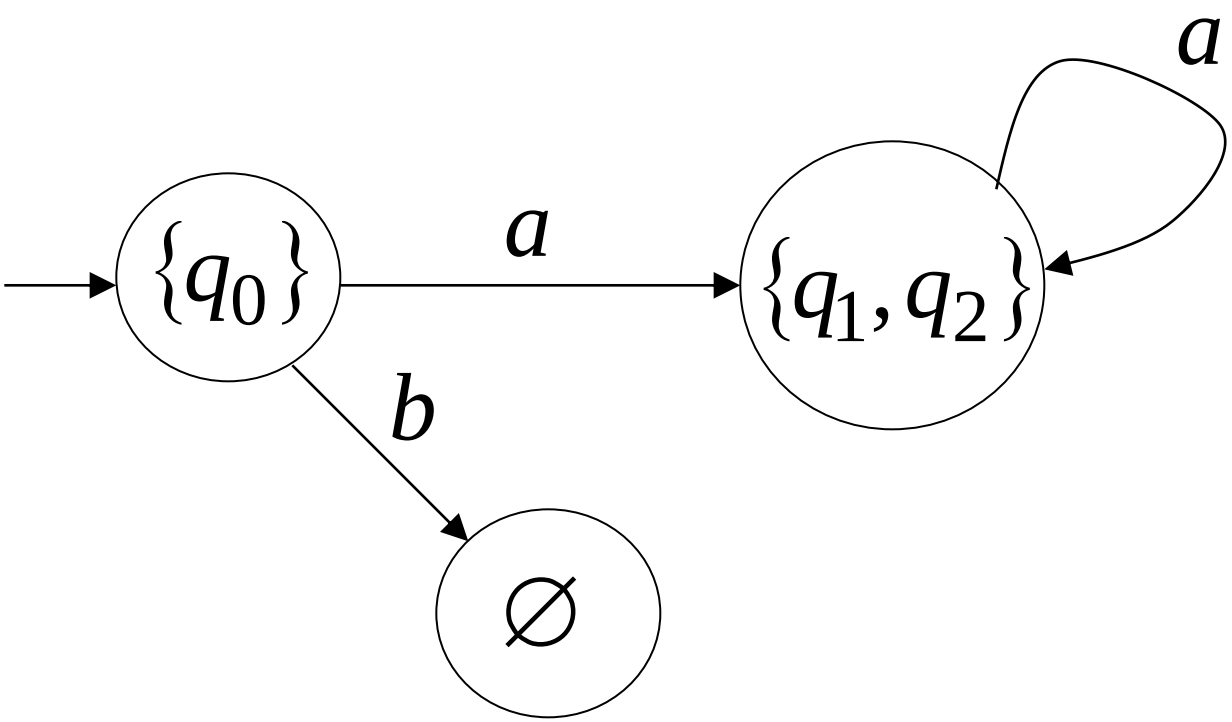


NFA to DFA

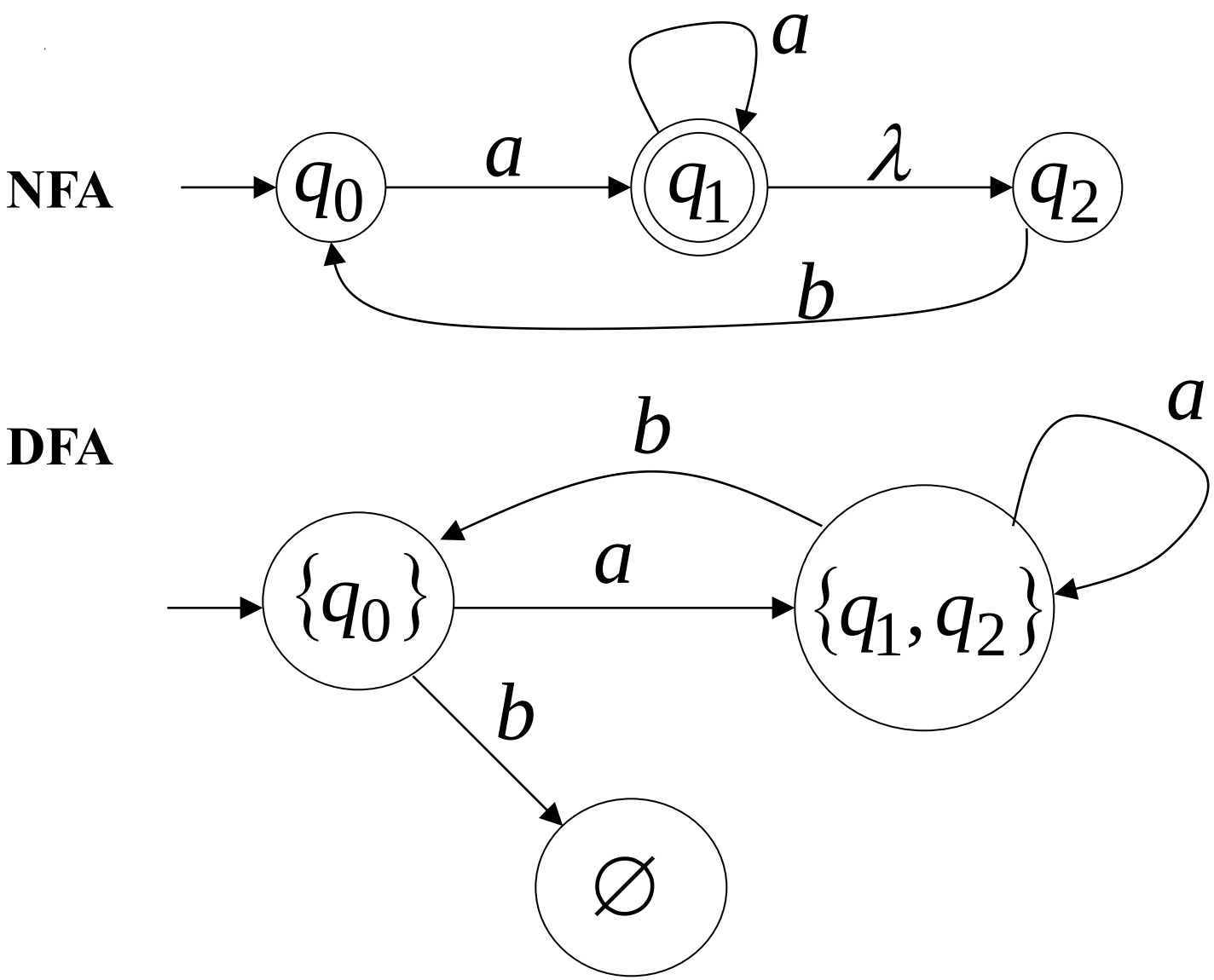
NFA



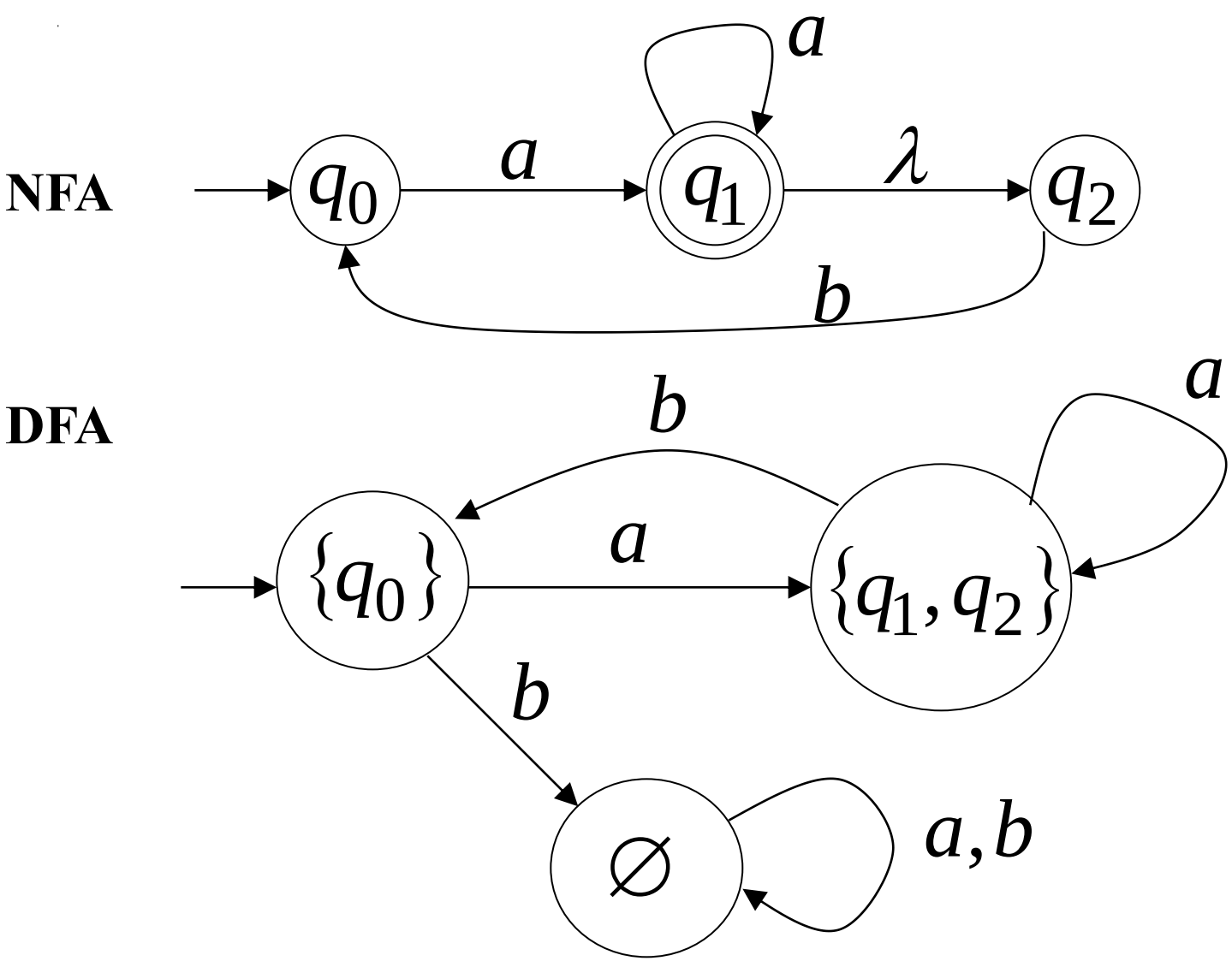
DFA



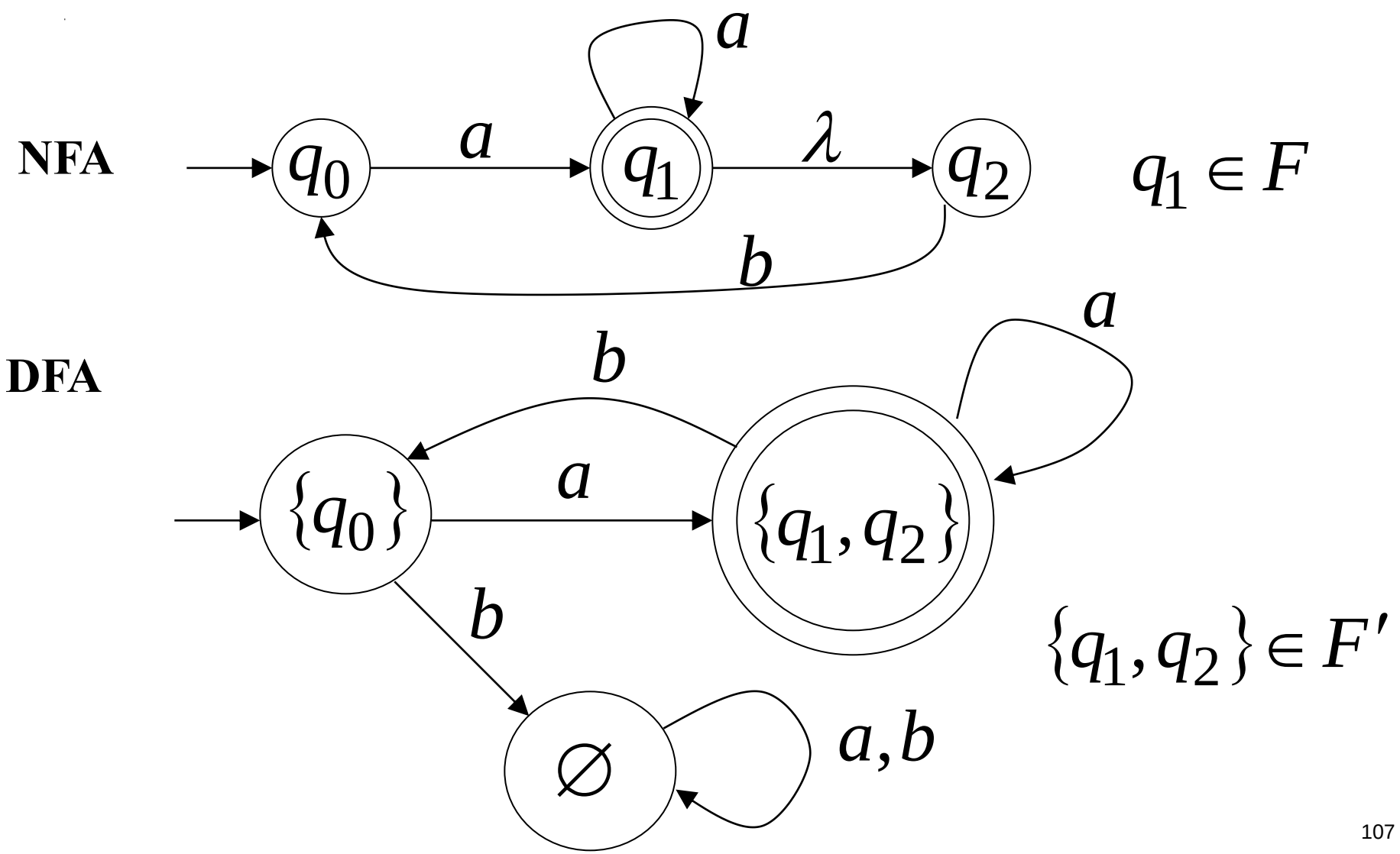
NFA to DFA



NFA to DFA



NFA to DFA


NFA
q_0
q_1
q_2
a
a
λ
b
$q_1 \in F$
DFA
$\{q_0\}$
$\{q_1, q_2\}$
$\emptyset$
a
b
a
b
a, b
$\{q_1, q_2\} \in F'$

Topic Map

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description
  - Design a finite automaton that describes the tokens and write a program that implements the state automaton
  - Design a finite automaton that describes the tokens and hand-construct a table-driven implementation of the state diagram

Implementing Finite Automata

- Hand-construct a transition table for DFA

- A DFA can be implemented by a 2D table
  - One dimension is **states**
  - The other dimension is **input symbols**
  - For every transition $S_i -^{a}-> S_k$ define T[i, a] = k

Implementing DFA

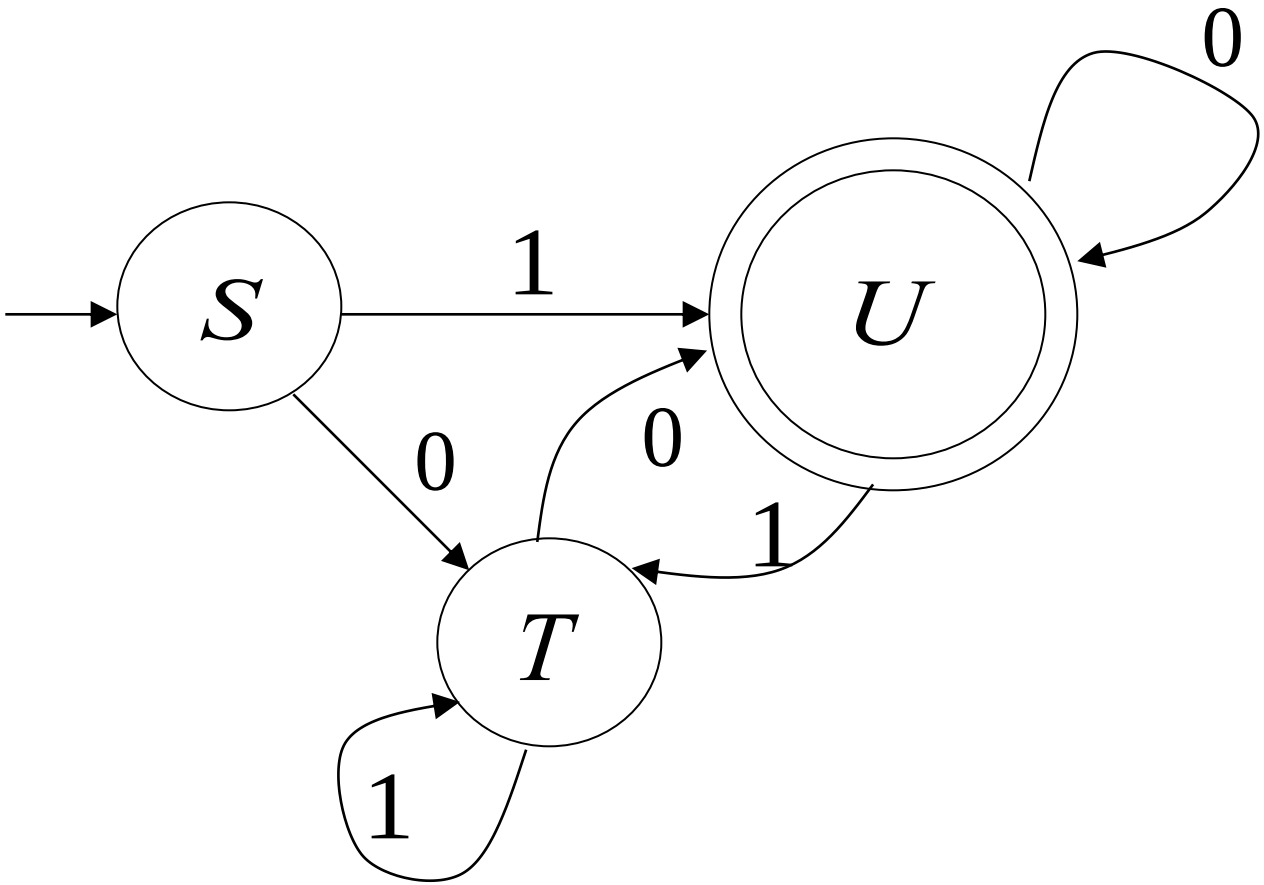


Table Driven Implementation

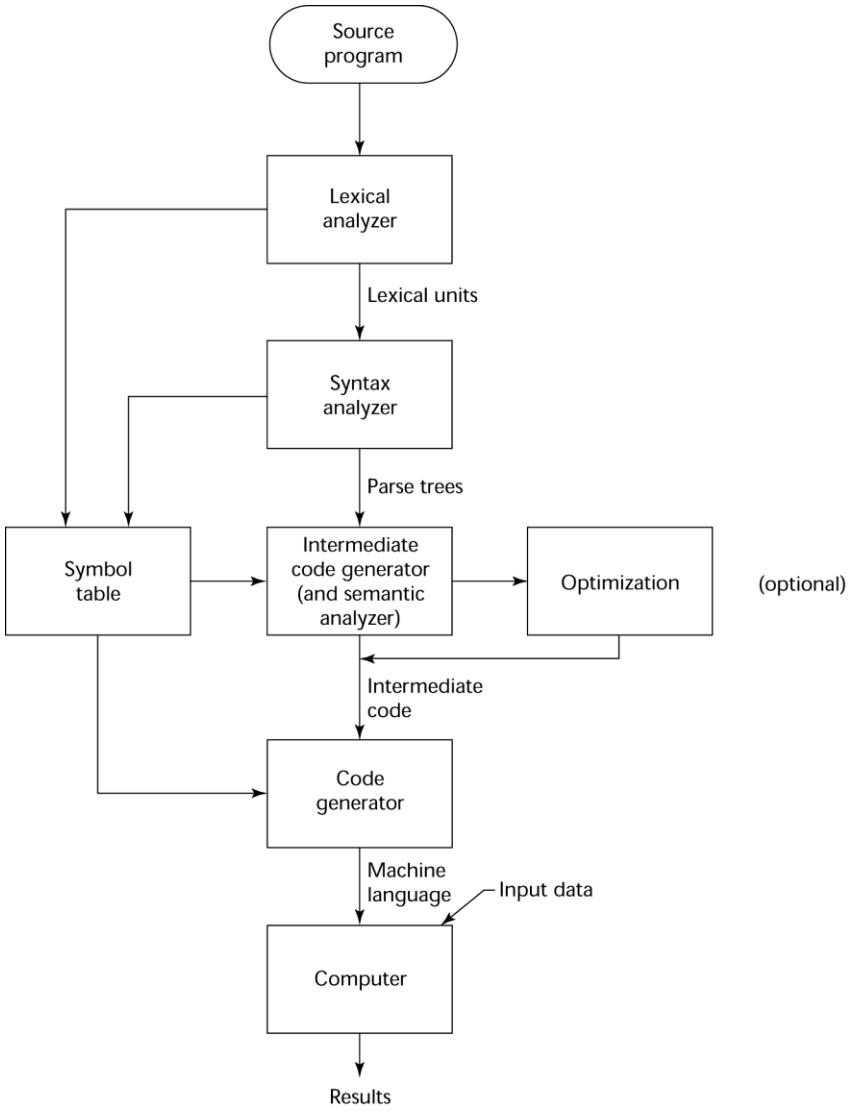
The transition table of the DFA

	0	1
S	T	U
T	U	T
U	U	T

Table Driven Implementation

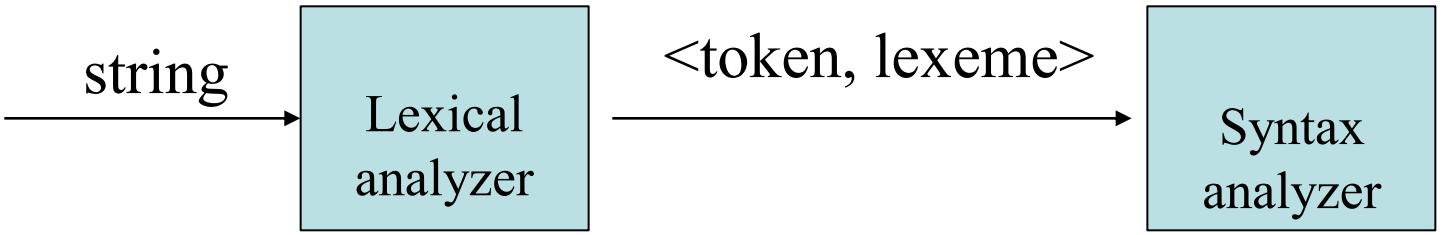
```
i=0;//input index
state = 0;//current state
while(input[i])//input array
{//table name is A
    state = A[state, input[i++]];
}
```

Revisit: The Compilation Process



Lexer -> Parser

- The goal of lexical analysis
  - Classify program substrings according to role (token class)
  - Communicate <token, lexeme> pairs to the syntax analyzer



The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, usually an Abstract Syntax Tree (**AST**), for the program

Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

Context Free Grammar and Parser

- Consider the grammar
  - <E> -> <T> | <T>+<E>
  - <T> -> int | int*<T> | (<T>)
- Given an input program 1*2 + 3*4
  - The output of the Lexer: int_1, '*', int_2, '+', int_3, '*' , int_4
  - Create a leftmost derivation and a rightmost derivation, corresponding to two strategies of building a parser
- Build a parse tree -> AST

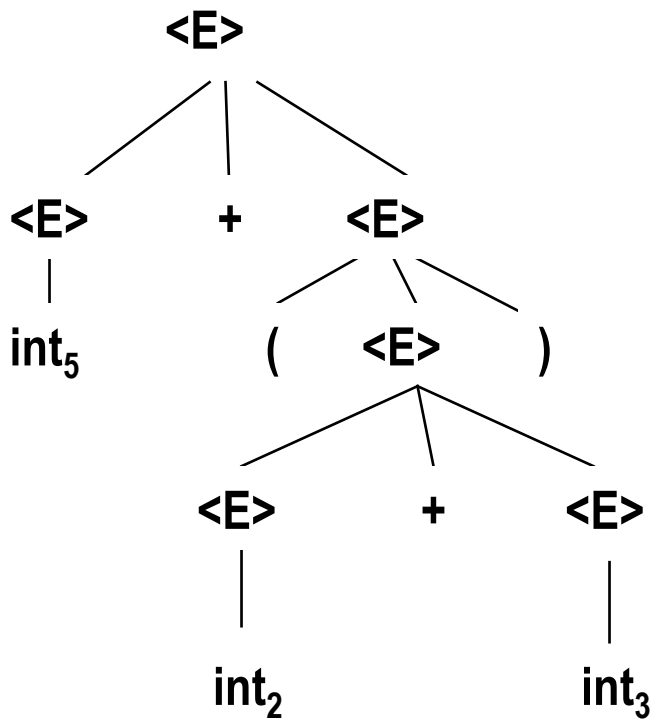
Abstract Syntax Tree

- The core data structure used in compilers
- Like parse trees but ignore some details
- A structural representation of a program

Abstract Syntax Trees

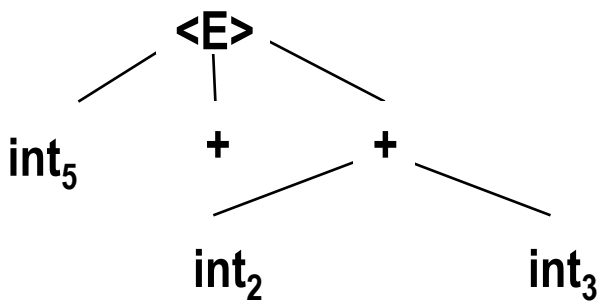
- Consider the grammar
  - <E> -> int | (<E>) | <E> + <E>
- And the string
  - 5 + (2 + 3)
- After lexical analysis, we get a list of tokens: int_5, '+', '(' , int_2, '+' , int_3, ')'
- During parsing we build a parse tree…

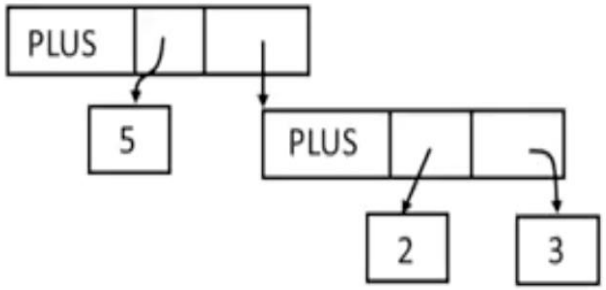
The Parse Tree



- A parse tree
  - A faithful representation of the program
- Traces the operation of the parser
- Captures nesting structure
- But too much information
  - Parentheses
  - Single-successor nodes

Abstract Syntax Tree





- Also captures the nesting structure
- But abstracts from the concrete syntax
  - More compact and easier to use
- Keeps just adequate information to faithfully represent a program
- An important data structure in a compiler

The Parsing Problem (continued)

- Two categories of parsers
  - *Top down* – produce the parser tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* – produce the parser tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input

The Parsing Problem (continued)

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms (LL algorithms)
  - Recursive descent – a coded implementation
  - Parsing table– table driven implementation

The Parsing Problem (continued)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem (continued)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time (O(n), where n is the length of the input)

Recursive-Descent Parsing

- Consider the BNF grammar
  - <E> -> <T> | <T> + <E>
  - <T> -> int | int * <T> | (<E>)
- Token stream is: (int_5)
- Start with top-level non-terminal E
- Always try the rules for a non-terminal in order
- When one rule fails, backtrack to try alternative rules

Recursive-Descent Parsing

- First try <E>-> <T> and <T> ->int, check if int matches the next token in the input that is a '('. it fails. So backtrack to T rules and choose int*T. Unfortunately, it fails again.
- Backtrack again to T rules and choose (E). The leftmost terminal '(' matches the next token '(', so proceed by advancing input pointer.

Recursive-Descent Algorithm

- Let TOKEN be the type of tokens
  - Special tokens INT, LP, RP, PLUS, TIMES
- Let the global pointer ***next*** point to the next input token

Recursive-Descent Algorithm

- Define bool functions that check for a match of
  - A given token terminal:

```
bool term(TOKEN tok) {return *next++ == tok;}
```

  - The nth rule of non-terminal S:

bool S_n() {…}

  - Try all rules of S:

```
bool S() {…}
```

Functions of Non-Terminal E

- For rule <E> -> <T>
  - bool E1() {return T();}
- For rule <E> -> <T> + <E>
  - bool E2() {return T() && term(PLUS) && E();}
- Try all rules for E
  - bool E() {

    TOKEN *save = next;

    return (next = save, E1()) || (next = save, E2());

    }

Functions of Non-terminal T

- For rule <T> -> int
  - bool T1() {return term(INT);}
- For rule <T> -> int * <T>
  - bool T2() {return term(INT) && term(TIMES) && T();}
- For rule <T> ->(<E>)
  - bool T3() {return term(LP) && E() && term(RP);}
- Try all rules for T

```
bool T() {
    TOKEN *save = next;
    return (next = save, T1())
        || (next = save, T2())
        || (next = save, T3());}
```

Recursive-Descent Algorithm

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- Easy to implement by hand
- Try example input (int)

Recursive-Descent Algorithm

```
bool term(TOKEN tok) {return *next++ ==tok;}
bool E1() {return T();}
bool E2() {return T() && term(PLUS) && E();}
bool E() { TOKEN *save = next;
          return (next = save, E1()) || (next = save, E2());}
bool T1() {return term(INT);}
bool T2() {return term(INT) && term(TIMES) && T();}
bool T3() {return term(LP) && E() && term(RP);}
bool T() {TOKEN *save = next;
         return (next = save, T1())
                || (next = save, T2())
                || (next = save, T3());}
```

Limitations

- If a production for non-terminal X succeeds
  - Cannot backtrack to try a different production for X later
  - Try the algorithm on input $int_5 + int_5$
- General recursive-descent algorithms support such "full" backtracking
  - Can implement any grammar

Limitations

- The presented recursive descent algorithm is not general
  - But is easy to implement by hand
- Sufficient for grammars where for any non-terminal at most one production can succeed
- The example grammar can be rewritten to work with the presented algorithm
  - By left factoring

Exercise

- Write a recursive decent parser for the following grammar

```
<assign> -> <var> = <expr>
<expr> -> <var> + <var>
<var> -> A | B | C
```

Recursive-Descent Parsing

- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of non-terminals
- Create a function for each nonterminal in its associated grammar
- For example, the grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int | ( <expr> )
```

Recursive-Descent Parsing

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, call `lex` to get the next input token, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing function

Recursive-Descent Parsing

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */
  term();
/* As long as the next token is + or -, call
   lex to get the next token and parse the next term */
  while (nextToken == ADD_OP || nextToken == SUB_OP){
    lex();
    term();
  }
}
```

Recursive-Descent Parsing

- This particular function does not detect errors
- Convention: every parsing routine leaves the next token in **`nextToken`**

Recursive-Descent Parsing

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>}
*/
void term() {
/* Parse the first factor */
  factor();

/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
} /* End of function term */
```

Recursive-Descent Parsing

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it uses to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing

```
/* Function factor, Parses strings in the language generated by the rule:
   <factor> -> id  | int | (<expr>)  */

 void factor() {
 /* Determine which RHS */
   if (nextToken == ID_CODE || nextToken == INT_CODE)
 /* For the RHS id or integer, just call lex */
     lex();
 /* If the RHS is (<expr>) - call lex to pass over the left parenthesis,
   call expr, and check for the right parenthesis */
  else if (nextToken == LP_CODE) {
     lex();
     expr();
     if (nextToken == RP_CODE)
        lex();
     else
        error();
    }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
}
```

Recursive-Descent Parsing

- Trace of the lexical and syntax analyzers on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Limitations in Recursive-Descent

- Consider the grammar
  - <E> -> <T> | <T> + <E>
  - <T> -> int | int * <T> | (<E>)
- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to left-factor the grammar

Pairwise Disjointness

- One characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS based on only one token of lookahead
  - Def: FIRST(α) = {a | α =>* aβ }

    (If α =>* ε, ε is in FIRST(α))

Pairwise Disjointness

- Pairwise Disjointness Test:
  - For each nonterminal A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

    $$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$

- Examples:

  1. $A \rightarrow a \mid bB \mid cAb$ yes
  2. $X \rightarrow ab \mid Yd$ no

     $Y \rightarrow a \mid b \mid c$

More examples

1. $A \rightarrow aB \mid bAb \mid Bb$

   $B \rightarrow cB \mid d$

2. $A \rightarrow aB \mid BAb$

   $B \rightarrow aB \mid b$

Left Factoring

- The idea is to eliminate the common prefixes of multiple productions for one non-terminal
- Factor out the common prefix and introduce a new non-terminal, e.g.,
  - **Replace <E> -> <T> | <T> + <E> with**
  - **<E> -> <T> <X>**
  - **<X> -> +<E> | ε**
- Left-factor <T> -> int | int * <T> | (<E>)

Left Factoring

- <E> -> <T> | <T> + <E>
- <T> -> int | int * <T> | (<E>)

- ➔
  - **<E> -> <T> <X>**
  - **<X> -> +<E> | ε**
  - **<T> -> int <Y> | (<E>)**
  - **<Y> -> *<T> | ε**

Try (int), int* int, and int + int*int

Exercise

Perform left-factoring for:

<variable>→identifier | identifier ‘[’<expression>‘]’

Modified grammar:

<variable> → identifier <new>

<new> → ε | ‘[’<expression>‘]’

Left Recursion

- Consider a production S -> Sα
  - bool S_1() {return S() && term(a)}
  - bool S () {return S_1 ()}
- S() goes into an infinite loop
- A left-recursive grammar has a non-terminal S where
  - S ->$^+$Sα for some α
- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser

Left Recursion

- Consider the left-recursive grammar
  - $S \rightarrow S\alpha|\beta$
  - $S \rightarrow S\alpha \rightarrow S\alpha\alpha \rightarrow S\alpha\alpha\alpha \ldots \alpha \rightarrow \beta\alpha\alpha\alpha$
- S generates all strings with a β and followed by any number of α's in the order of right to left
- The very last thing it produces is the first thing that appears in the input
- However, the recursive descent parsing needs to match the first part of the input

Rewrite Left Recursion

- Replace left recursion by right recursion
  - $S \rightarrow \beta S'$
  - $S' \rightarrow \alpha S' \mid \varepsilon$
- In a more general form
  1. $S \rightarrow S\alpha_1 | \ldots | S\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$, where none of the β's begins with S
  2. Replace the original S-rules with

     $S \rightarrow \beta_1 S' \mid \beta_2 S' \mid \ldots \mid \beta_n S'$

     $S' \rightarrow \alpha_1 S' \mid \alpha_2 S' \mid \ldots \mid \alpha_m S' \mid \varepsilon$

Predictive Parsing

- Predictive Parser is a recursive descent parser that does not require backtracking
- Can "predict" which production to use by looking at the next few tokens, **lookahead**
- Accepts LL(k) grammars
  - The first L: left-to-right scan
  - The second L: leftmost derivation
  - k: the number of tokens to look ahead, in practice k is almost always equal to 1

Recursive Descent vs Predictive Parsing

- In recursive descent
  - At each step, there may be many choices of production to use
  - Backtracking used to **undo bad choices**
- In LL(1)
  - At each step, at most one choice of production
  - Given a sentential form $\dot{\omega}A\beta$ and the next input token t, for the leftmost non-terminal A, there is only a production $A->\alpha$ that can be used
  - Any other production is guaranteed to be incorrect

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (continued)

- Intuition about handles:
  - Def: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$
  - Def: β is a *phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - Def: β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

Bottom-up Parsing (continued)

- Intuition about handles (continued):
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

Bottom-up Parsing (continued)

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (continued)

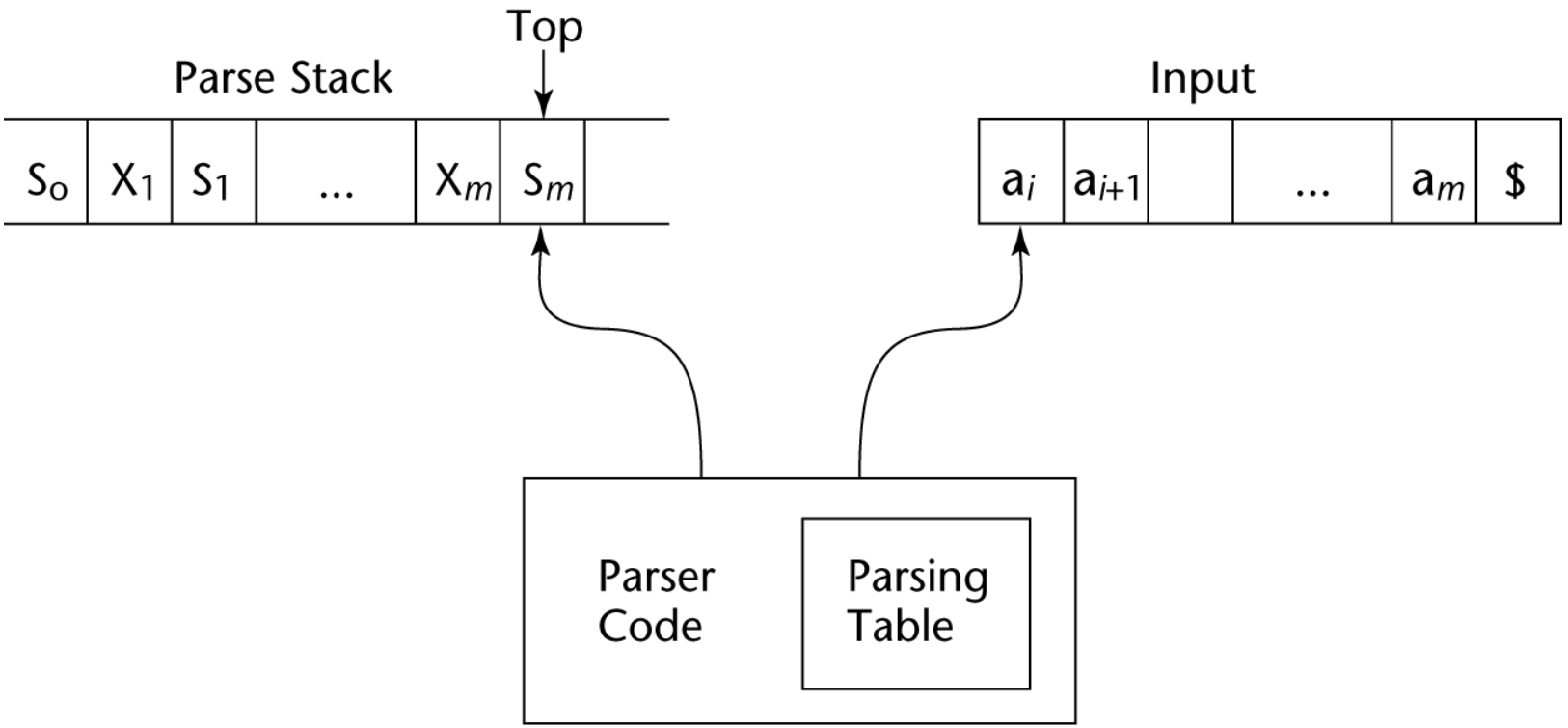
- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (continued)

- An LR configuration stores the state of an LR parser

$(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$

Structure of An LR Parser



Bottom-up Parsing (continued)

- Initial configuration: (S_0, $a_1 \ldots a_n$\$)
- Parser actions:
  - For a Shift, the next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the Action table
  - For a Reduce, remove the handle from the stack, along with its state symbols. Push the LHS of the rule. Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table

Bottom-up Parsing (continued)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Bottom-up Parsing (continued)

- Parser actions (continued):
  - For an Accept, the parse is complete and no errors were found.
  - For an Error, the parser calls an error-handling routine.

Grammar

1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (continued)

- A parser table can be generated from a given grammar with a tool, e.g., **`yacc`** or **`bison`**

Bottom-up Parsing (continued)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
- A recursive-descent parser is an LL parser
  - Detects syntax errors
  - Produces a parse tree
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach