Finite State Machines

- An automaton is a computation that determines whether a given string belongs to a specified language
- A finite state machine (FSM) is an automaton that recognize regular languages (regular expressions)

Finite State Machines(cont'd)

- In particular ,finite automata can be used to describe the process of recognizing patterns in input strings and so can be used to construct scanners.

Finite State Machines(cont'd

- Formal basis for lexical analysis is the finite state automaton (FSA)
  - REs generate regular sets
  - FSAs recognize regular sets
- FSA – informal definition:
  - A finite set of states
  - Transitions between states
  - An initial state (start)
  - A set of final states (accepting states)

Finite Automata and Lexical Analysis

- The tokens of a language are specified using regular expressions.
- A scanner is a big DFA, essentially the “aggregate” of the automata for the individual tokens.

Two Kinds of FSA

- Non-deterministic finite automata (NFA)
  - There may be multiple possible transitions or some transitions that do not require an input (ε)
- Deterministic finite automata (DFA)
  - The transition from each state is uniquely determined by the current input character
    - For each state, at most 1 edge labeled ‘a’ leaving state
  - No ε transitions

NFA

- Non-deterministic finite automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have (ε) moves

Epsilon Moves

- ε -moves machine can move from state A to state B without consuming input

NFA vs DFA

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement – table driven.
- For a given language, the NFA can be simpler than the DFA.
- DFA can be exponentially larger than NFA.

NFA vs DFA

- NFAs are the key to automating
  - RE → DFA Construction
- NFA Construction (Thompson's construction)
  - Build an NFA for each RE term.
  - Combine NFAs with ε -moves.

RE → Finite Automata

- Can we build a finite automation for every regular expression?
- **Yes**, - build FA inductively based on the definitions of Regular Expression

- DFAs are easier to implement – table driven.

RE → NFA Construction

- Subset construction
  - NFA → DFA
- Build the simulation
- Minimum number of states in DFA (Hopcroft's algorithm)

RE → NFA Construction

- Key idea
  - NFA pattern for each symbol and each operator
  - Join them with ε – moves in precedence order

NFA → DFA Construction

- The algorithm is called subset construction.
- In the transition table of an NFA, each entry is a set of states.
- In DFA, each entry is a single state.
- The general idea behind NFA-to-DFA construction is that each DFA state corresponds to a set of NFA states.

NFA → DFA Construction

- The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.
- We will use the following operations:
  - ε-closure(T):
    - Set of NFA states reachable from some NFA state s in T on ε-transitions alone.
  - move(T,a):
    - Set of NFA states to which there is a transition on input a from some NFA state s in set of states T.

NFA → DFA Construction

- Before it sees the first input symbol, NFA can be in any of the state in the set ε-closure(s0), where s0 is the start state of the NFA.
- ε-transitions The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.
- We will use the following operations:
  - ε-closure(T):
    - Set of NFA states reachable from some NFA state s in T on ε-transitions alone.
  - move(T,a):

Implementing the Scanner

- Three methods
  - Hand-coded approach:
    - Draw DFSM, then implement with loop and case statement
  - Hybrid approach :
    - Define tokens using regular expressions, convert to NFSM, apply algorithm to obtain minimal DSFM
    - Hand-code resulting DFSM
  - Automated approach:
    - Use regular grammar as input to lexical scanner generator (e.g. LEX)

Hand-coding

- Branch depending on first character:
  - If digit, scan numeric literal
  - If character, scan identifier or keyword
  - If operator, check next character (++, etc.)
- Return token found
- Write aggressive efficient code: goto's, global variables

NFAs & DFAs

- Non-Deterministic Finite Automata (NFAs) easily represent regular expression, but are somewhat less precise.

- Deterministic Finite Automata (DFAs) require more complexity to represent regular expressions, but offer more precision.

Non-Deterministic Finite Automata

- **An NFA is a mathematical model that consists of :**
  - **A set of states,S**
  - **A set of input symbols Σ (input symbol alphabet)**
  - **A transition function ,move, that maps state-symbol pairs to sets of states.**
    - ***move*(state, symbol) → set of states**
    - **A state s_0 that is distinguished as the start (or initial) state**
    - **A set of states F , distinguished as accepting (or final)state**

Representing NFAs

- **Transition Diagrams :**

  Number states (circles), arcs, final states, …

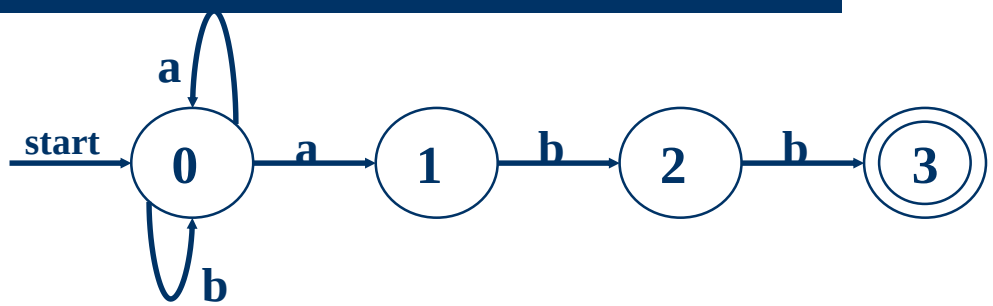
- **Transition Tables:**

  More suitable to representation within a computer

We'll see examples of both !

Example NFA

- S = { 0, 1, 2, 3 }
- s0 = 0
- F = { 3 }
- Σ = { a, b }

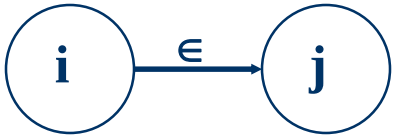


What Language is defined ? (a|b)*abb

What is the Transition Table ?

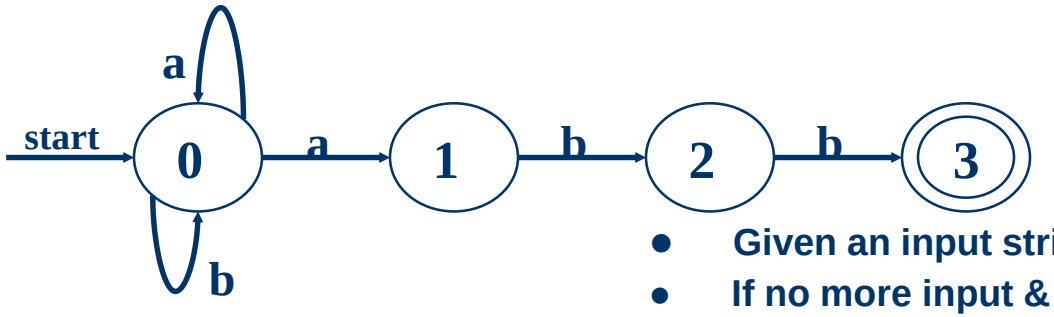
State	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

∈ (null) moves possible



Switch state but do not use any input symbol

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE:

Input: ababb

***move*(0, a) = 1**

***move*(1, b) = 2**

***move*(2, a) = ? (undefined)**

REJECT !

-OR-

***move*(0, a) = 0**

***move*(0, b) = 0**

***move*(0, a) = 1**

***move*(1, b) = 2**

***move*(2, b) = 3**

ACCEPT !

How Does An NFA Work ?(cont'd)

An NFA can be represented diagrammatically by a labeled directed graph ,called **transition graph** ,in which the nodes are the states and the labeled edges represent the transition function. This graph looks like a transition diagram ,but the same character can label two or more transitions out of one state, and edges can be labeled by the especial ε as well as by input symbols.

How Does An NFA Work ?(cont'd)

The transition graph for an NFA that recognizes the language (a | b)*abb. The set of states of the NFA is { 0,1,2,3 } and the input symbol alphabet is { a, b} State 0 is distinguish as the start state and state 3 is indicated by a double circle.

How Does An NFA Work ?(cont'd)

- When describing an NFA we use the transition graph representation. In a computer, the transition function of an NFA can be implemented in several different ways.

How Does An NFA Work ?(cont'd)

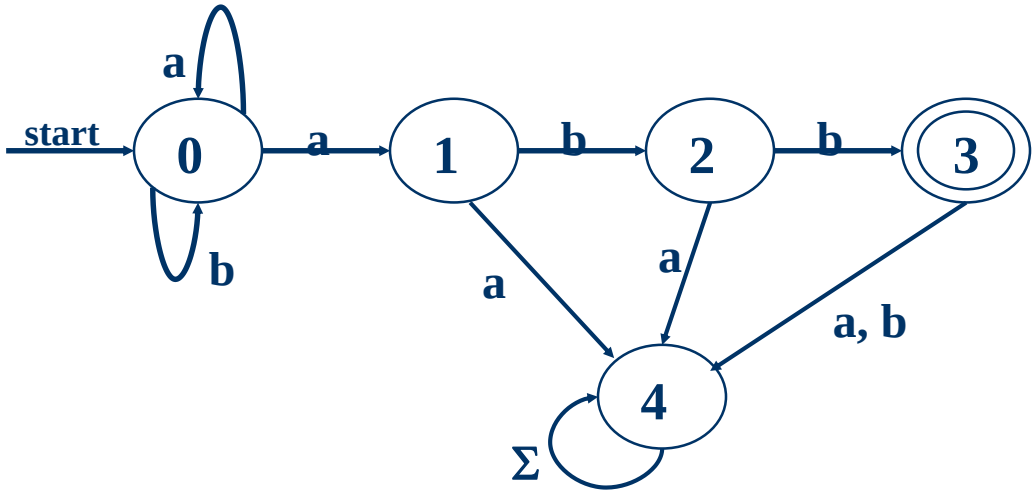
- The easiest implementation is a transition table in which there is a row for each state and a column for each input symbol and ε if necessary. The entry for row *i* and symbol *a* in the table is the set of states(or more likely in practice, a pointer to the set of states) that can be reached by a transition from state *i* on input "a". the transition table for the NFA is shown in the above slides.

How Does An NFA Work ?(cont'd)

- The transition table representation has the advantage that it provides fast access to the transitions of a given state on a given character; its disadvantage is that it can take up a lot of space when the input alphabet is large and most transitions are to the empty set.

Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a "death" state, and transitioning all previously undefined transition to this death state.



NFA- Regular Expressions & Compilation

Problems with NFAs for Regular Expressions:

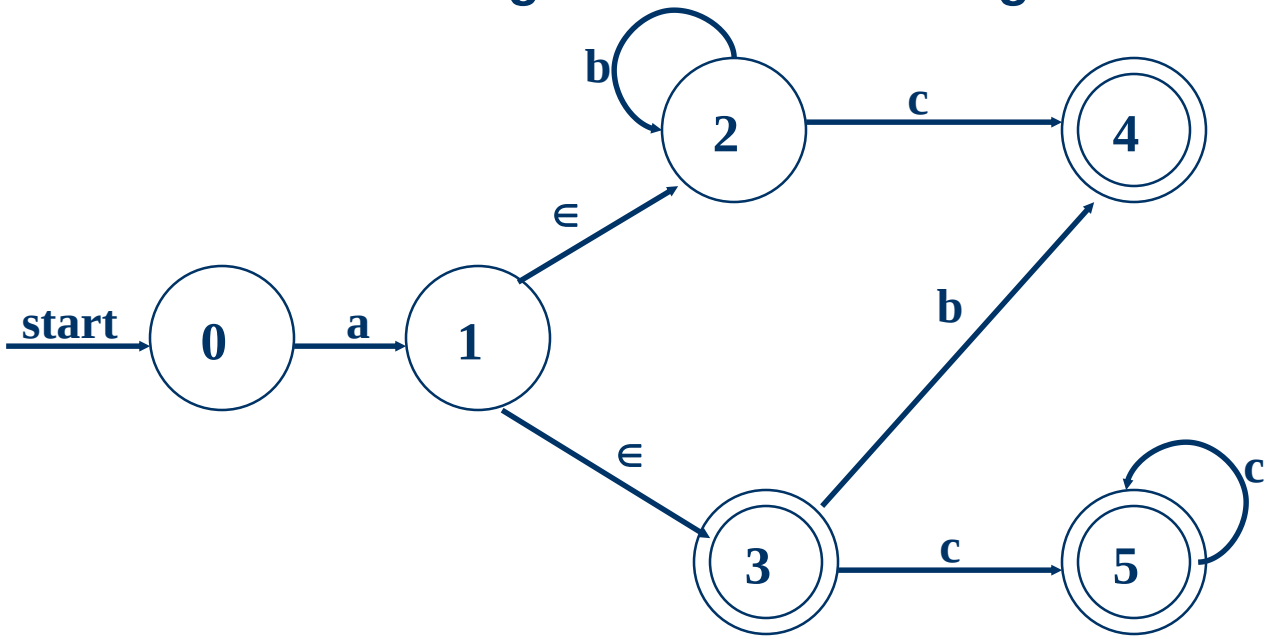
1. Valid input might not be accepted
2. NFA may behave differently on the same input

Relationship of NFAs to Compilation:

1. Regular expression "recognized" by NFA
2. Regular expression is "pattern" for a "token"
3. Tokens are building blocks for lexical analysis
4. Lexical analyzer can be described by a collection of NFAs. Each NFA is for a language token.

Second NFA Example

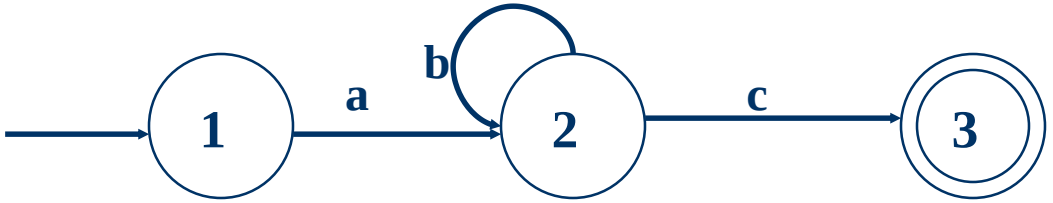
- Given the regular expression : (a (b*c)) | (a (b | c+)?)
- Find a transition diagram NFA that recognizes it.



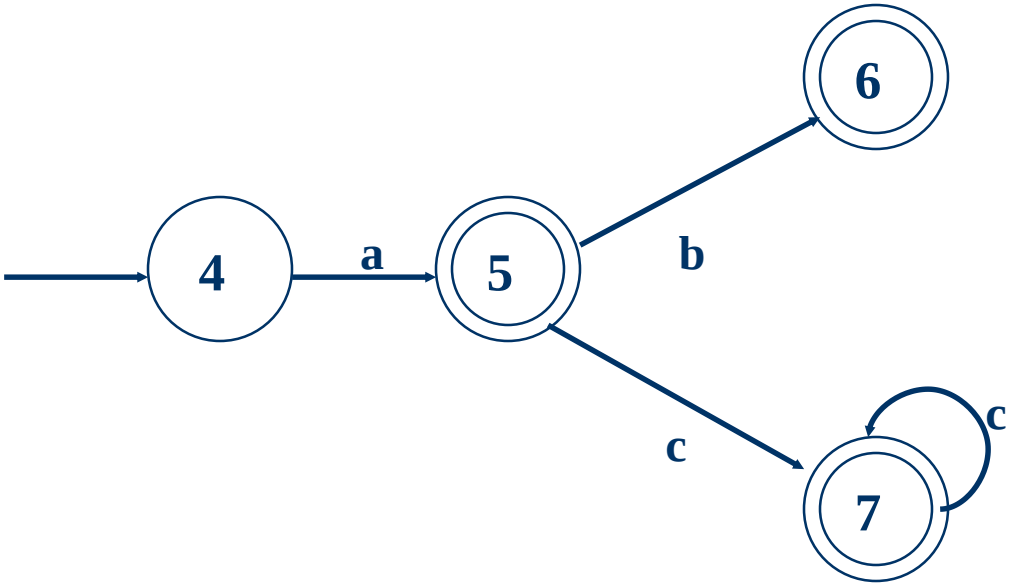
String abbc can be accepted.

Alternative Solution Strategy

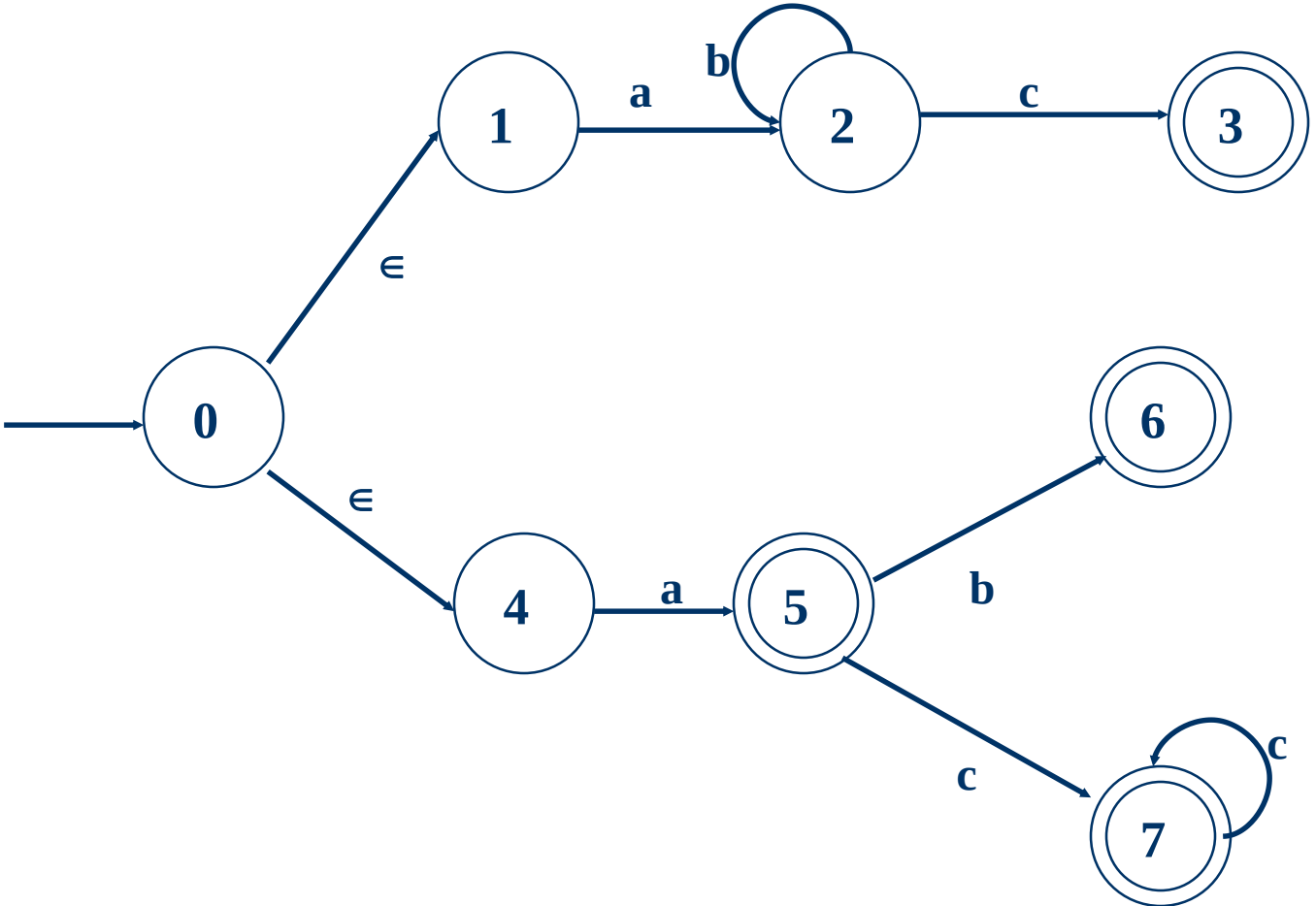
a (b*c)



a (b | c+)?

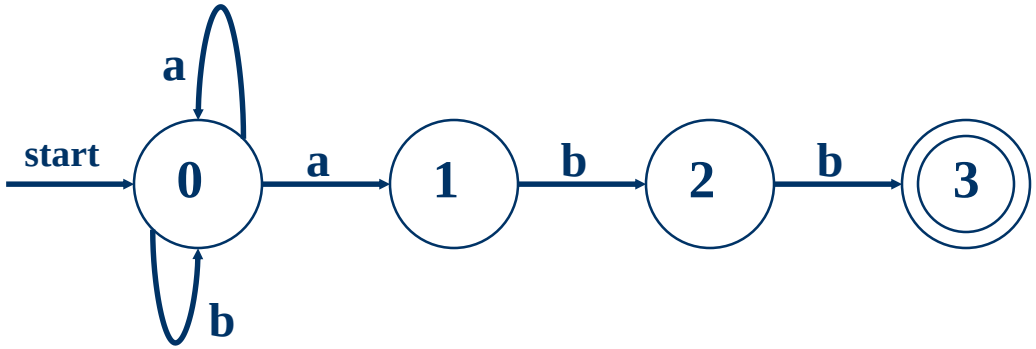


Using Null Transitions to “OR” NFAs



Other Concepts

Not all paths may result in acceptance.



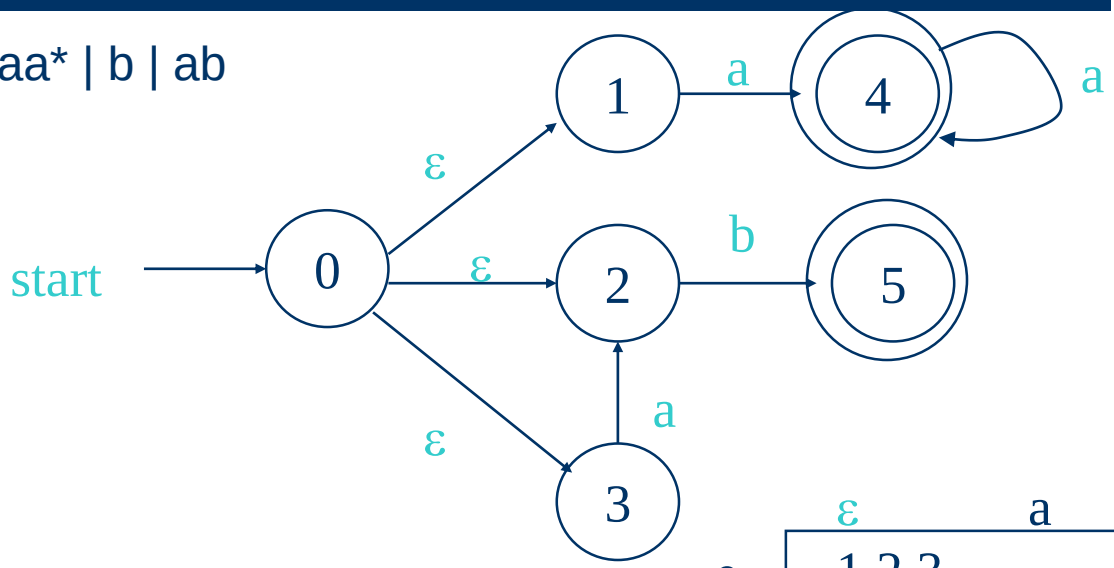
aabb is accepted along path : $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

BUT… it is not accepted along the valid path:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

NFA Example

Recognizes: aa* | b | ab



Can represent FA with either graph of transition table

	ε	a	b
0	1,2,3	-	-
1	-	4	Error
2	-	Error	5
3	-	2	Error
4	-	4	Error
5	-	Error	Error

Deterministic Finite Automata

- A DFA is an NFA with a few restrictions
  - No epsilon transitions(€)
  - For every state s, there is only one transition (s,x) from s for any symbol x in Σ
  - ADVANTAGES
  - Easy to implement a DFA with an algorithm!
  - Deterministic behavior

Simulating a DFA

INPUT:

An input string x terminated by end of file character eof(or any other delimiter). A DFA ‘d’ with start state s^o and a set of accepting states F.

OUTPUT:

The answer “yes” if ‘d’ accepts x, “no” other wise

Simulating a DFA

METHOD:

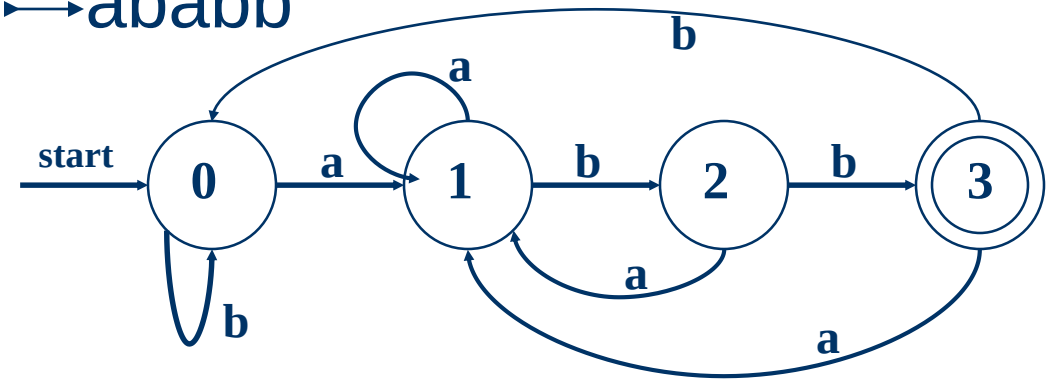
Apply the algorithm to the input string x .The function move(s,c) gives the state to which there is a transition from state s on input character c . The function nextchar returns the next character of the input string x.

Simulating a DFA(cont'd)

- s = s0
- c = nextchar;
- while c ≠ eof do
- s = move(s,c);
- c = nextchar;
- end;
- if s is in F then return "yes"
- else return "no"

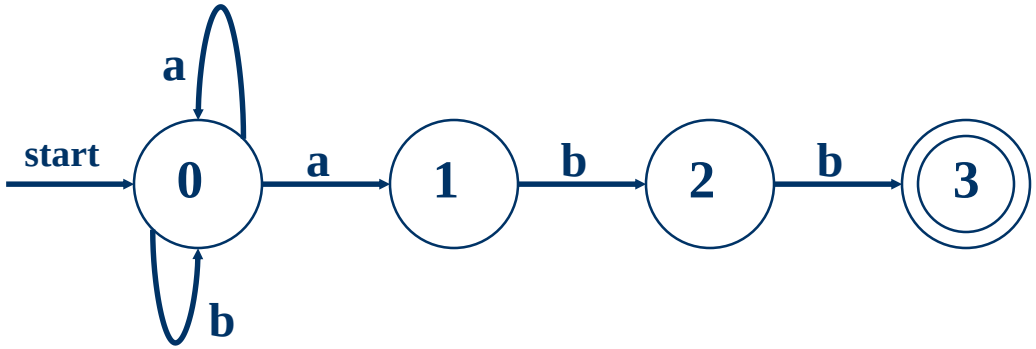
Following transition graph of a DFA accepting the language (a|b)*abb as that accepted by the NFA .With this DFA and input string ababb algorithm follows the sequence of state 0,1,2,1,2,3 and return “yes”.

- String⟶ababb



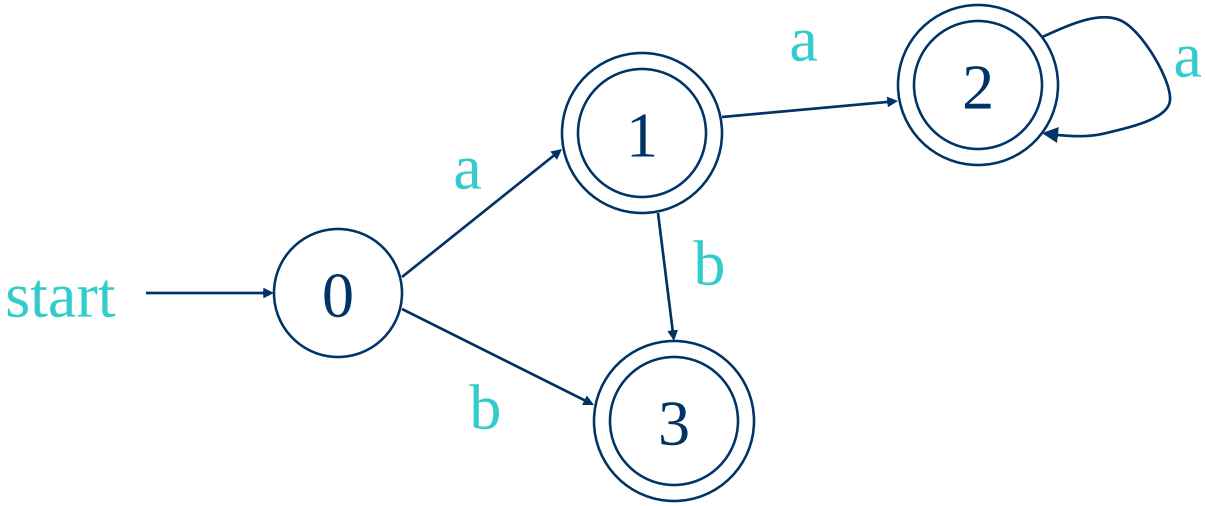
Recall the original NFA:

DFA accepting (a|b)*abb



DFA Example

Recognizes: aa* | b | ab



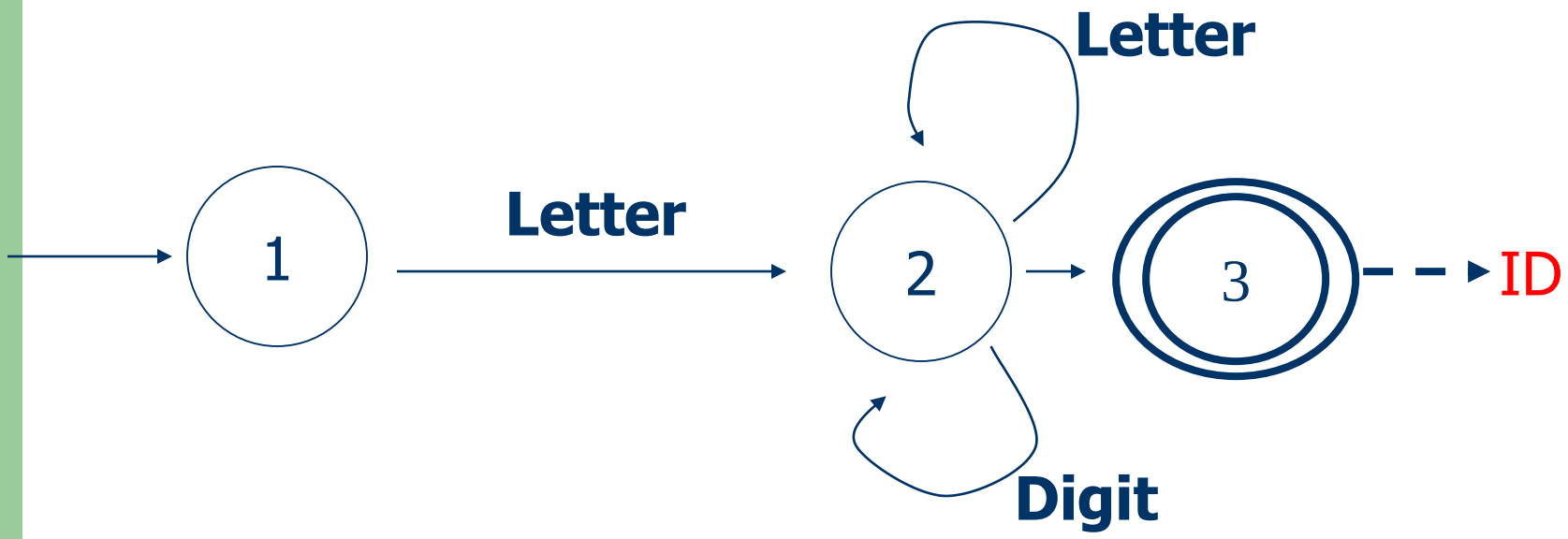
Finite State Machines(Cont'd)

The pattern for identifiers as commonly defined in programming languages is given by the following regular definition.

ID = Letter (Letter | Digit)*

This represent a string that begins with a letter and continues with any sequence of letters and / or digits. The process of recognizing such a string can be described by the diagram.

Finite State Machines(Cont'd)



Finite State Machines(Cont'd)

In the diagram ,the circles numbered 1 and 2 represent states, which are locations in the process of recognition that record how much of the pattern has already been seen. The arrowed lines represents transitions that record a change from one state to another upon a match of the character or characters by which they are labeled.

Finite State Machines(Cont'd)

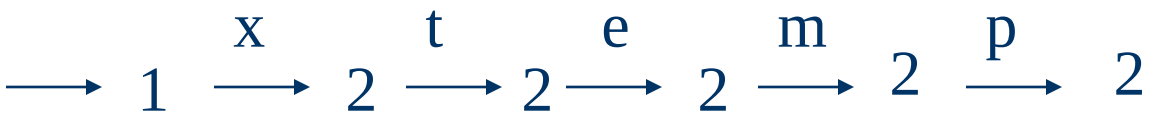
In the sample diagram , the state 1 is the start state or the state at which the recognition process begins. By convention the start state is indicated by drawing an un labeled arrowed line to it. On state 2 any number of letters and /or digits may be seen and a match to these return us to state 2.

Finite State Machines(Cont'd)

The states that represent the end of the recognition process in which we can declare success are called **Accepting States** and are indicated by drawing a double line border around the state in the diagram.There may be more than one of these.In the the sample diagram state 3 is an accepting state indicating that after a letter is seen ,any subsequent sequence of letters and digits represents a legal identifier.

Example

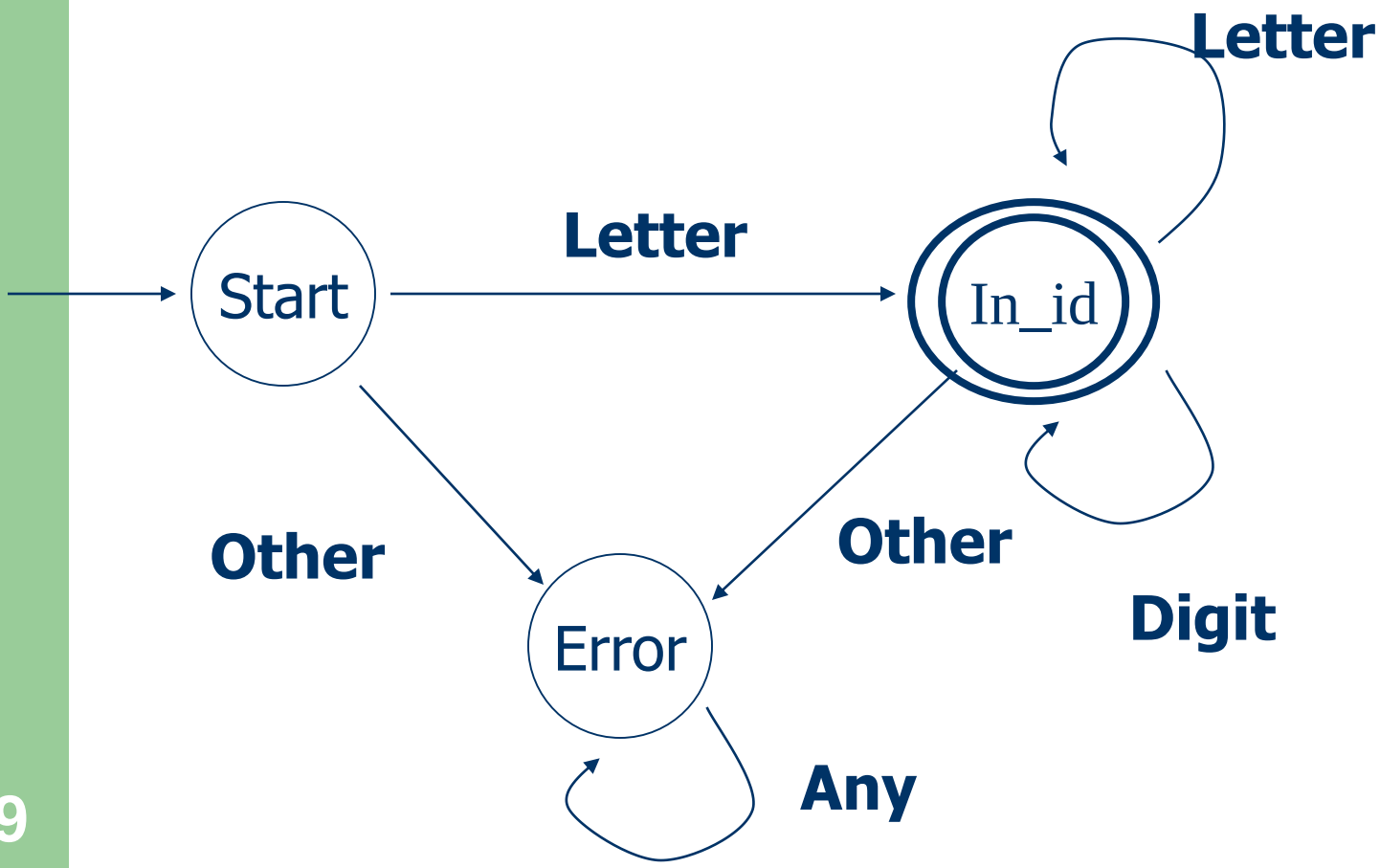
The process of recognizing an actual character string as an identifier can now be indicated by listing the sequence of states and transitions in the diagram that are used in the recognition process. For example,the process of recognizing **Xtemp** as an identifier can be indicated as follows.



Where Are The Missing Transitions?

The answer is that they represent errors that is ,in recognizing an identifier we can not accept any characters other than letters from the start state and letters or numbers after that .The convention is that these error transitions are not drawn in the diagram but are simply assumed to always exist.If we were to draw them the diagram for an identifier would look as show in next slide.

Where Are The Missing Transitions?(cont'd)



Where Are The Missing Transitions?(cont'd)

In the figure,we have labeled the new state error (Since it represents an erroneous occurrence),and we have labeled the error transitions **OTHER.**By convention ,other represents any character not appearing in any other transition from the state where it originates.Thus the definition of **OTHER** coming from the start state is

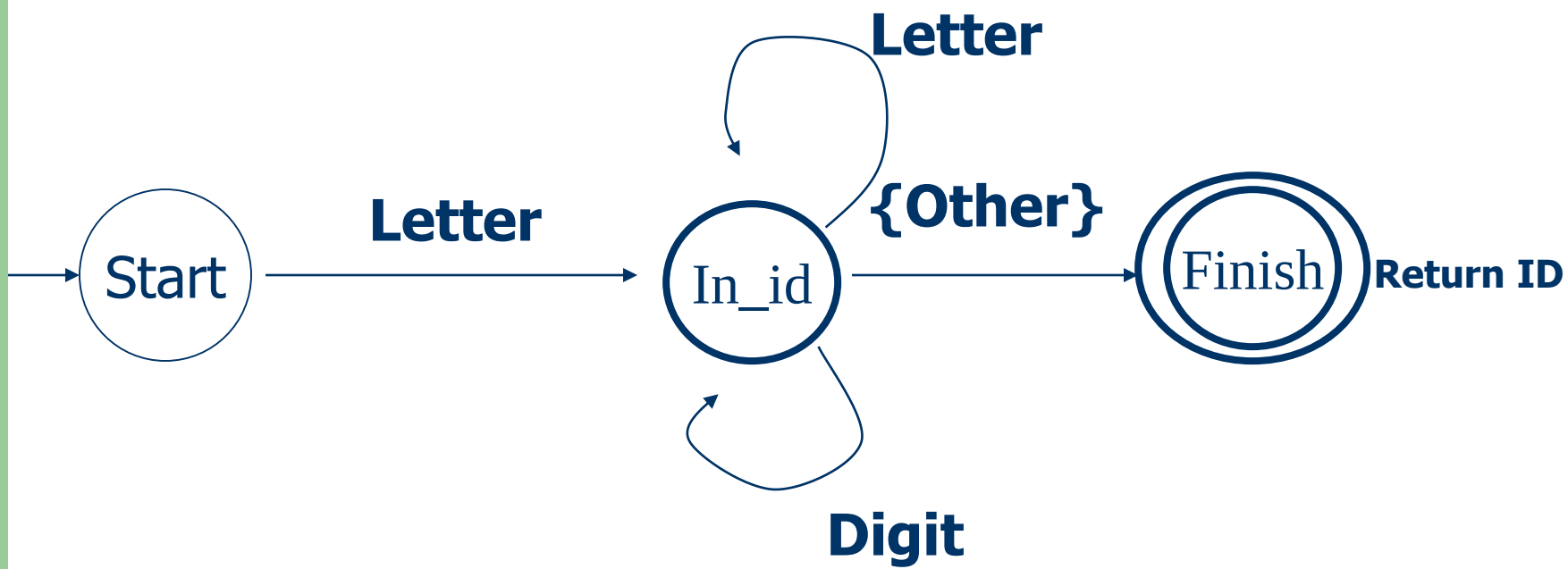
Other = - Letter

Where Are The Missing Transitions?(cont'd)

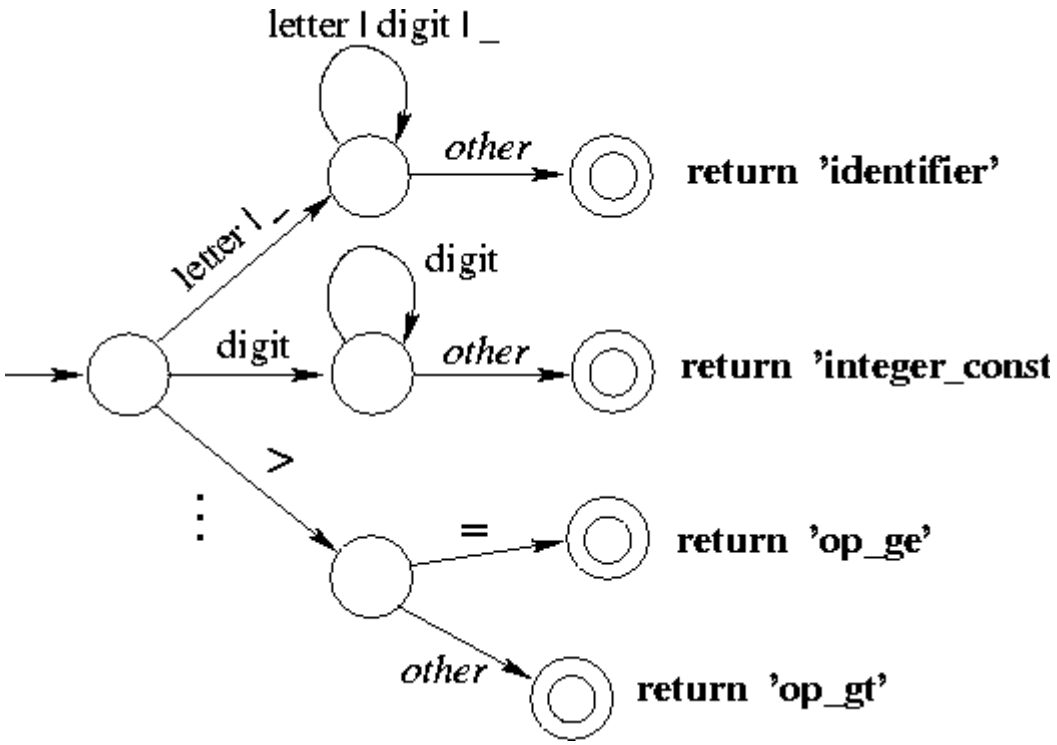
The definition of other coming from the stat in_id is

Other = - (Letter|Digit)

Where Are The Missing Transitions?(cont'd)

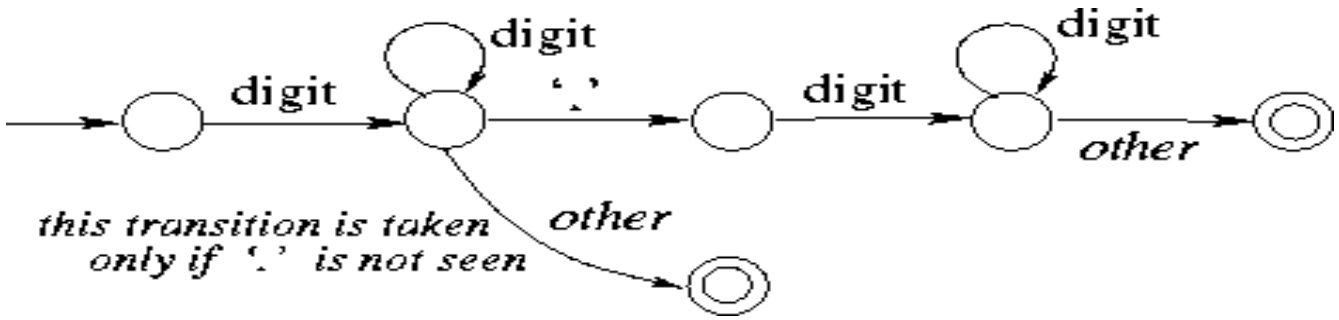


Structure of a Scanner Automaton



How much should we match?

- In general, find the longest match possible.
- E.g., on input 123.45, match this as
  - num_const(123.45)
- rather than
  - num_const(123), “.”, num_const(45).



THANKS