

The background of the slide features a gradient of blue at the top, transitioning into a series of overlapping, wavy bands of yellow and light blue. These bands curve across the lower half of the image, creating a sense of movement and depth.

Chapter 3 - Lexical Analysis

Recognition of tokens

- We have learnt how to express patterns for particular tokens using regular expressions.
- We must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

- Consider an example grammar in Pascal language:

stmt \rightarrow **if** *expr* **then** *stmt*
 | **if** *expr* **then** *stmt* **else** *stmt*
 | ϵ
expr \rightarrow *term* **relop** *term*
 | *term*
term \rightarrow **id**
 | **number**

Figure 3.10: A grammar for branching statements

Listing the terminals of the grammar

- As far as the lexical analyzer is concerned the terminals of the grammar would be **if**, **then**, **else**, **relop**, **id**, and **number** which are also the names of tokens to be generated.

The image shows a grammar for branching statements with the following productions:

<i>stmt</i>	→	if <i>expr</i> then <i>stmt</i>
		if <i>expr</i> then <i>stmt</i> else <i>stmt</i>
		ε
<i>expr</i>	→	<i>term</i> relop <i>term</i>
		<i>term</i>
<i>term</i>	→	id
		number

Figure 3.10: A grammar for branching statements

- The patterns for these tokens are described using regular definitions as follows:

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> ⁺
<i>number</i>	→	<i>digits</i> (. <i>digits</i>) ? (E [+ -] ? <i>digits</i>) ?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> (<i>letter</i> <i>digit</i>) *
<i>if</i>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	< > <= >= = <>

Figure 3.11: Patterns for tokens of Example 3.8

Resolving conflicts b/w keywords resembling identifiers

- For this language, the lexical analyzer will recognize the keywords `if`, `then`, and `else`, as well as lexemes that match the patterns for *relop*, *id*, and *number*.
- To simplify matters, we make the common assumption that keywords are also *reserved words*: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

Identifying whitespace characters

- We assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

$$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$$

- Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names.
- Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.

The following table shows for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	–	–
if	if	–
then	then	–
else	else	–
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	number	the numerical value
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Figure 3.12: Tokens, their patterns, and attribute values

Transition Diagrams

- Building flowcharts known as transition diagrams is an intermediate step in construction of a Lexical Analyzer.
- Transition diagrams are derived from the regular expressions of patterns to be matched.
- Transition diagram is a collection of nodes called states.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a symbol or set of symbols

Transition Diagrams cont..

- For eg: if we are in some state 's' , and if next symbol to be read is 'a'(pointed by forward pointer), then we look for an edge labelled 'a' and enter the state of transition diagram that the edge leads to, thereby advancing the forward pointer.

Naming conventions:

1. Accepting state → double circle

1. Indicates that lexeme has been found
2. Token as well as associated attribute value(if any) to be sent to parser.

Transition Diagrams cont..

2. * \rightarrow retract

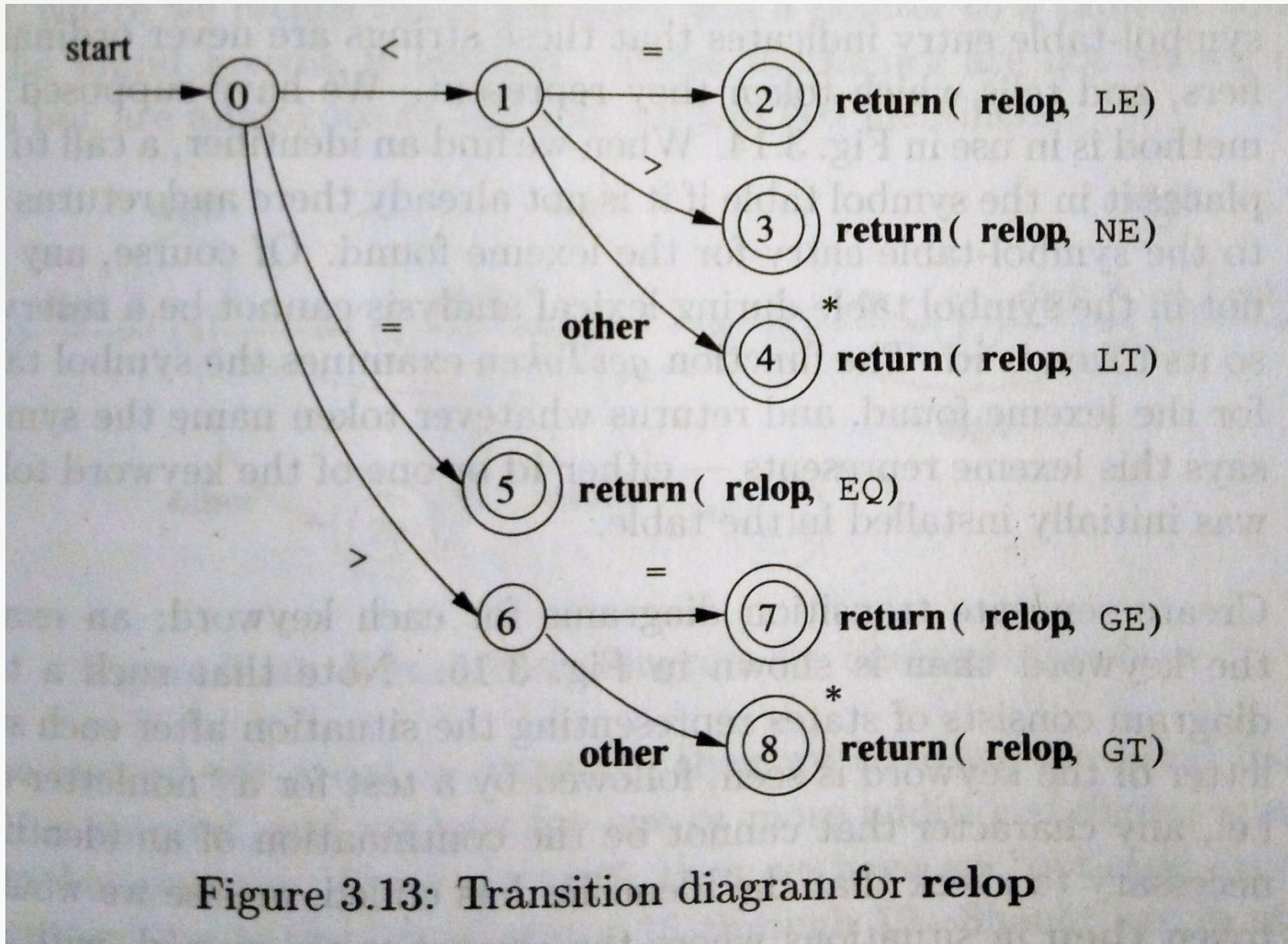
** \rightarrow retract two locations

3. start state , indicated by drawing \rightarrow on the particular state.

Transition diagram for Relop

- [..\videos\TD_RELOP_1.mp4](#)

Transition Diagram for relop



Transition Diagram for unsigned numbers

- ..\videos\TD_NUM1.mp4
- ..\videos\TD_NUM2.mp4

Transition Diagram for unsigned numbers

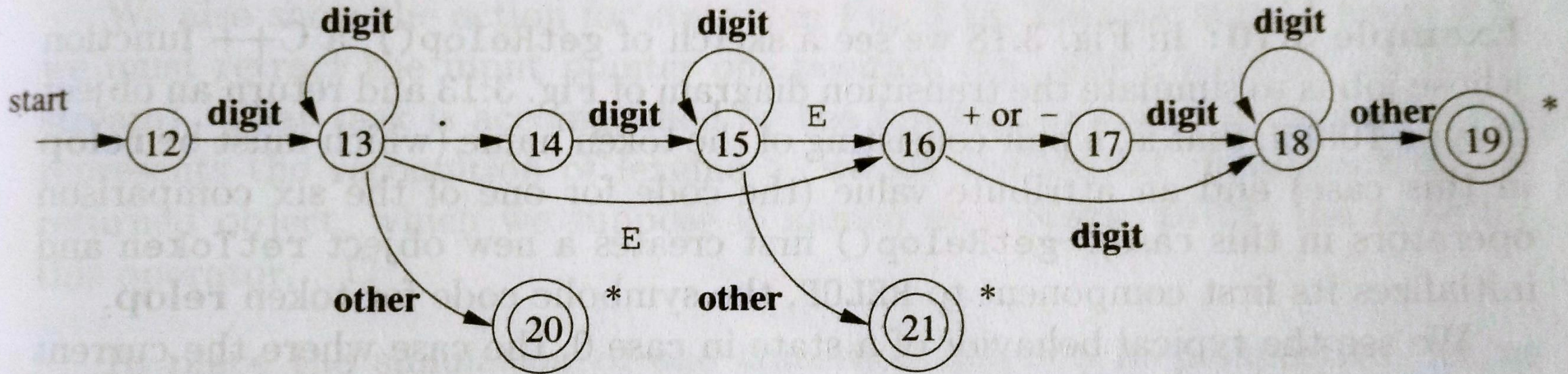
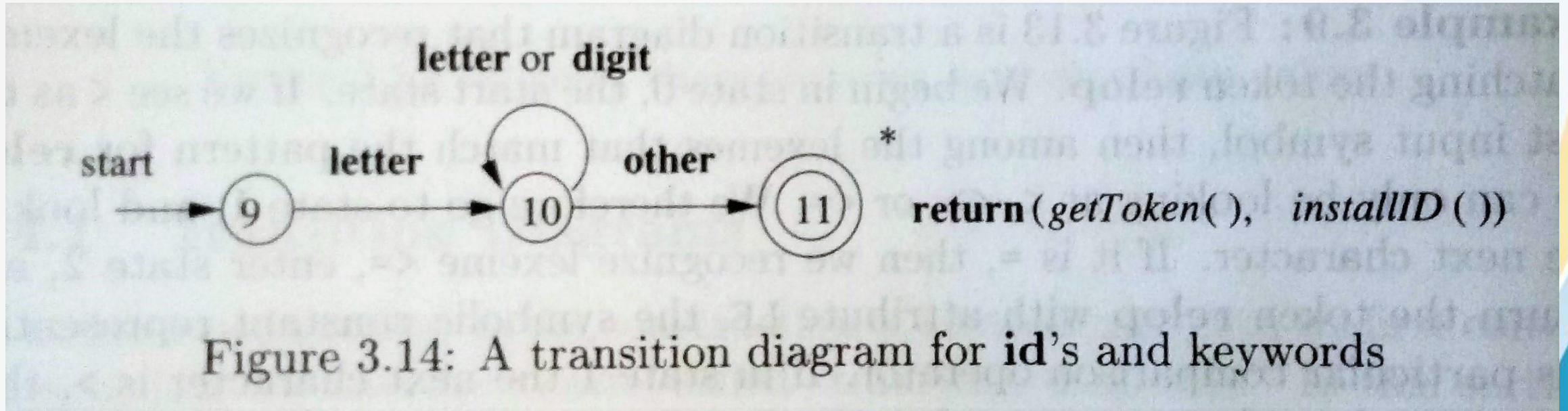


Figure 3.16: A transition diagram for unsigned numbers

NOTE: state 19,20 and 21 must return tokens

Recognition of Reserved words and Identifiers



Recognition of Reserved words and Identifiers

- There are two ways we can handle reserved words that look like identifiers

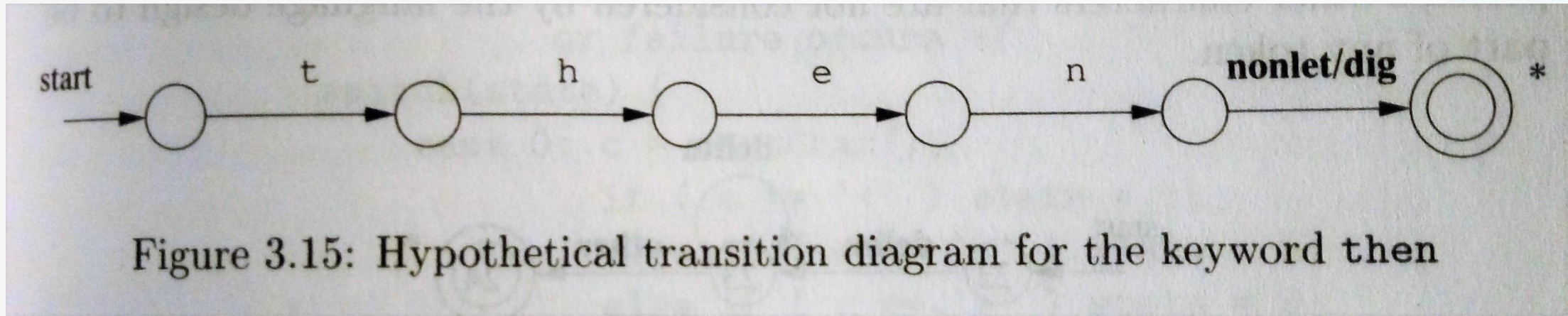
1. Install the reserved words initially in the symbol table

1. When we find an identifier, a call to *installID()* is made
2. If it is not a keyword, then it is placed in the symbol table if it is not already there and a pointer to the symbol-table entry for the lexeme is returned.
3. If it is a keyword, then the token name is returned using *getToken()*.

If same id appears twice or more times in the program, then *getToken()* returns id.

Recognition of Reserved words and Identifiers

- There are two ways we can handle reserved words that look like identifiers
2. Create separate transition diagram for every keyword.



Transition Diagram for whitespace

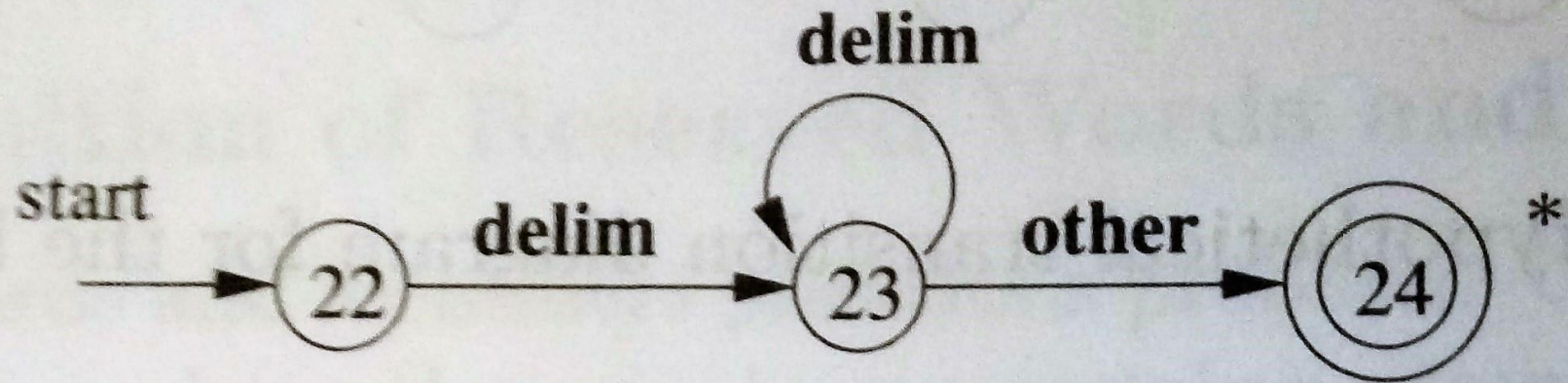


Figure 3.17: A transition diagram for whitespace

Transition diagram Summary

Regular expression are declarative specifications

- Transition diagram is an **implementation**
- A transition diagram consists of
 - An input alphabet belonging to Σ
 - A set of states S
 - A set of transitions state $i \rightarrow$ state j
 - A set of final states F
 - A start state n
- Transition $s1 \xrightarrow{a} s2$ is read: in state $s1$ on input a go to state $s2$
- If end of input is reached in a final state then accept
- Otherwise, reject

Architecture of a transition diagram based Lexical Analyzer

- Next step in construction of lexical analyzer is implementing the transition diagram.
- We will see **Implementation of RELOP transition diagram.**
- Function named *getRelop()*, C++ function to simulate the transition diagram is shown in next slide.
- *getRelop()* returns an object of type TOKEN which is pair consisting of token name and attribute value.
- *getRelop()* first creates new object named *retToken* and initializes its first component to RELOP.

Implementation of relop transition diagram

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

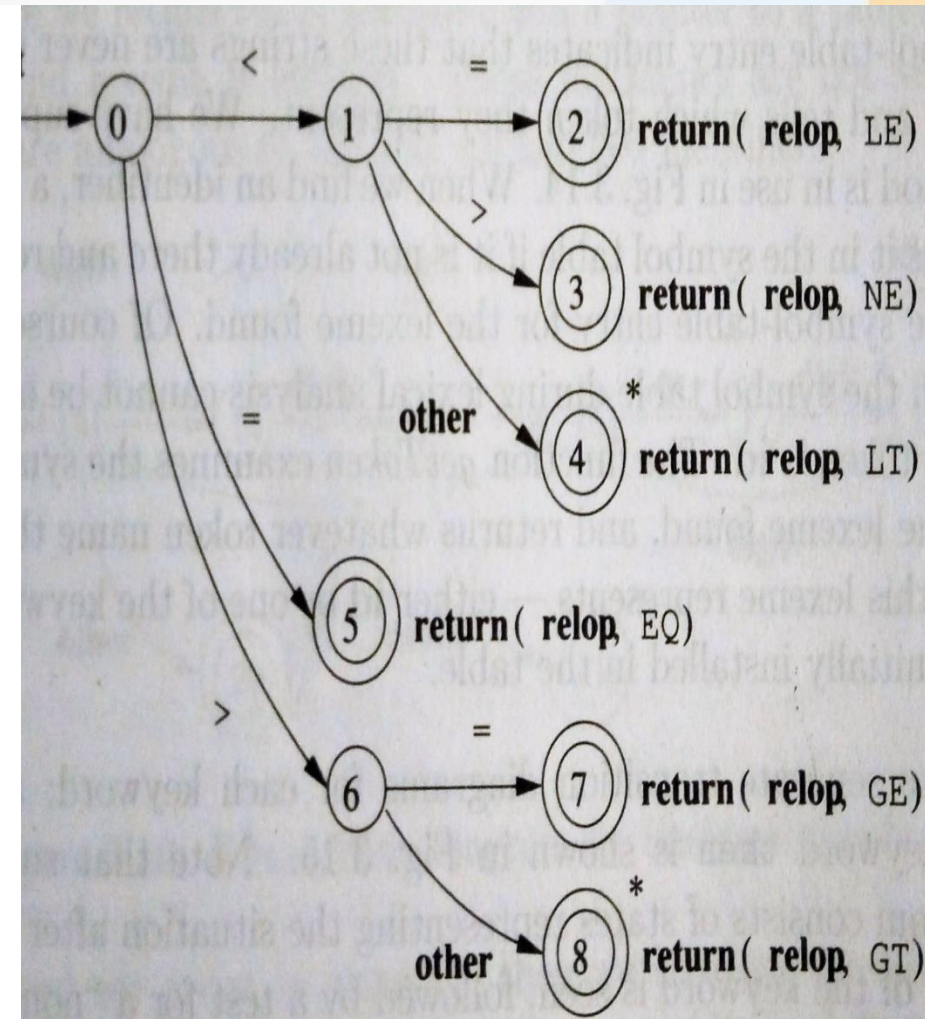


Figure 3.13: Transition diagram for `relop`

Implementation of relop transition diagram[contd]

- Case 1: `c=nextchar();`
 `If(c=='=') state =2;`
 `else if(c=='>') state = 3;`
 `else state=4;`

Implementation of relop transition diagram[contd..]

- *nextChar()* obtains the next character from the input and assigns it to variable 'c'.
- *fail()* is called when character read is not relational operator
 - Forward pointer is reset to lexemeBegin in order to allow another transition diagram to be applied.
 - It could initiate an error correction phase, if no other transition diagram left.
- *retract()* is used to move/retract input pointer one position back.

Implementation Lexical analyzer based on transition diagram [contd..]

- Three ways to simulate entire Lexical analyzer
 1. Could arrange all transition diagram for each token sequentially. I.e. Transition diagram for keywords, id, operator, special symbols etc are listed sequentially.
 - fail() resets the forward pointer and starts the transition diagram in sequence, each time it is called.

Implementation Lexical analyzer based on transition diagram [contd..]

2. Run all transition diagrams “in parallel” i.e input is fed to all transition diagram, the one which matches is returned.
3. Preferred approach is, combine all transition diagram into one.

Homework#2

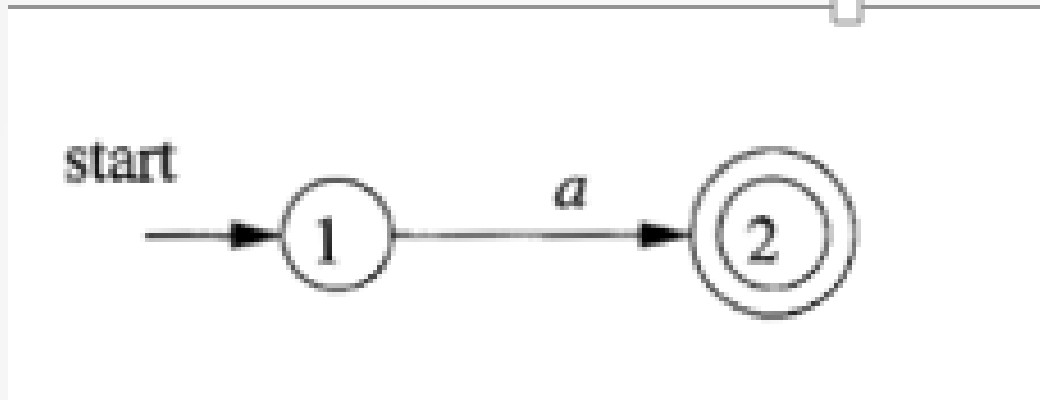
- Write the **remaining switch cases** of pseudocode for implementing transition diagram for RELOP(slide 22).
- Write the pseudocode for implementing transition diagram for regular expression: $a(a|b)^*a$
 - Draw transition diagram and then write C++ code, token name to be generated is PATTERN with no attribute value.

Finite automata

- Finite automata(FA) are recognizers; they say “yes” or “no”
 - Doesn’t return anything
- There are two types
 - Non deterministic finite automata(NFA)
 - Deterministic finite automata(DFA)

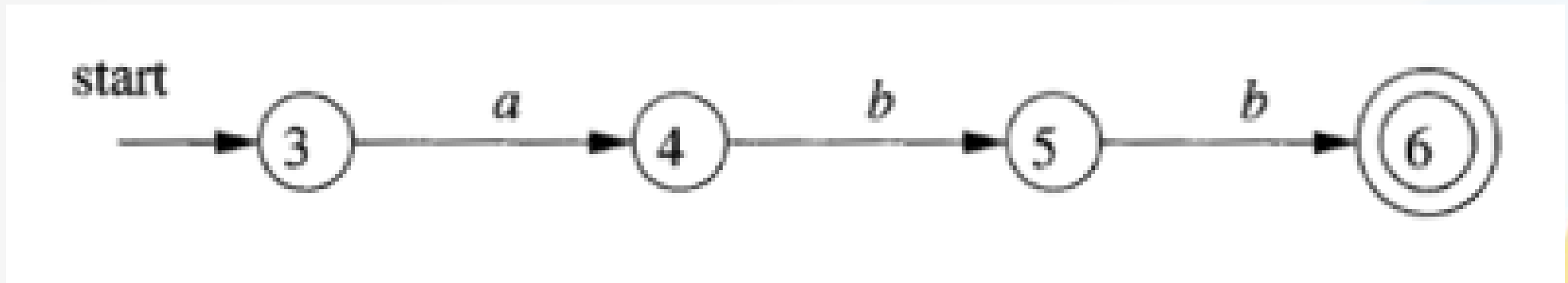
Few examples of NFA

- P1 : a



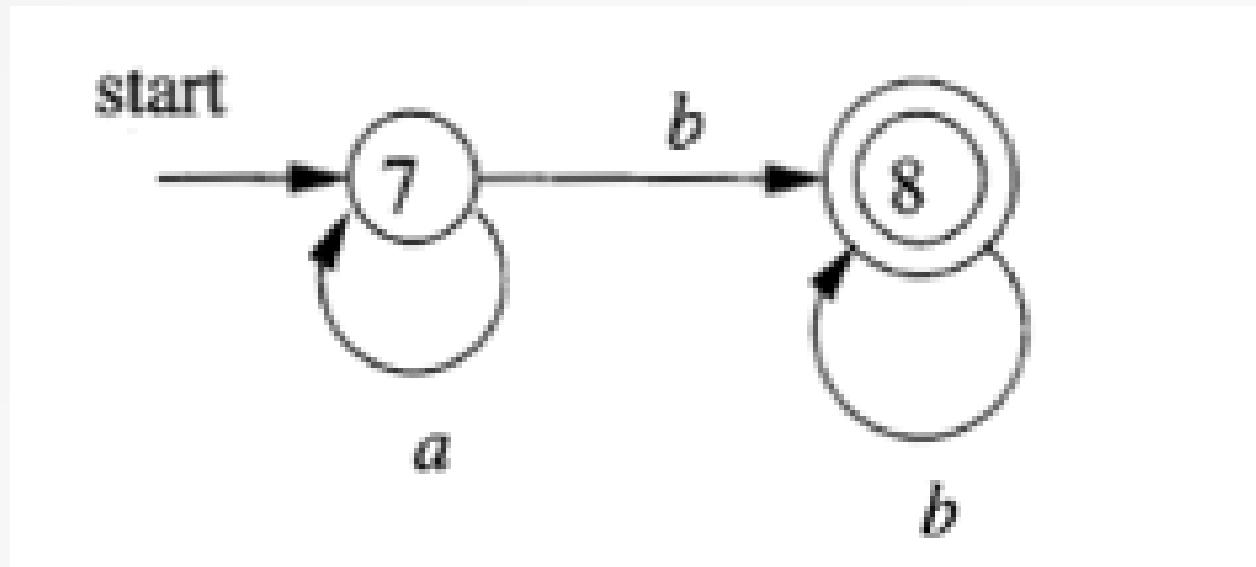
Few examples of NFA[contd]

- P2: abb



Few examples of NFA[contd]

- P3 : a^*b^+



The lexical analyzer generator Lex

- Scanners generally work by looking for patterns of characters in the input.
- For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter or an underscore followed by zero or more letters, underscores or digits, and the various operators are single characters or pairs of characters.
- A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp.

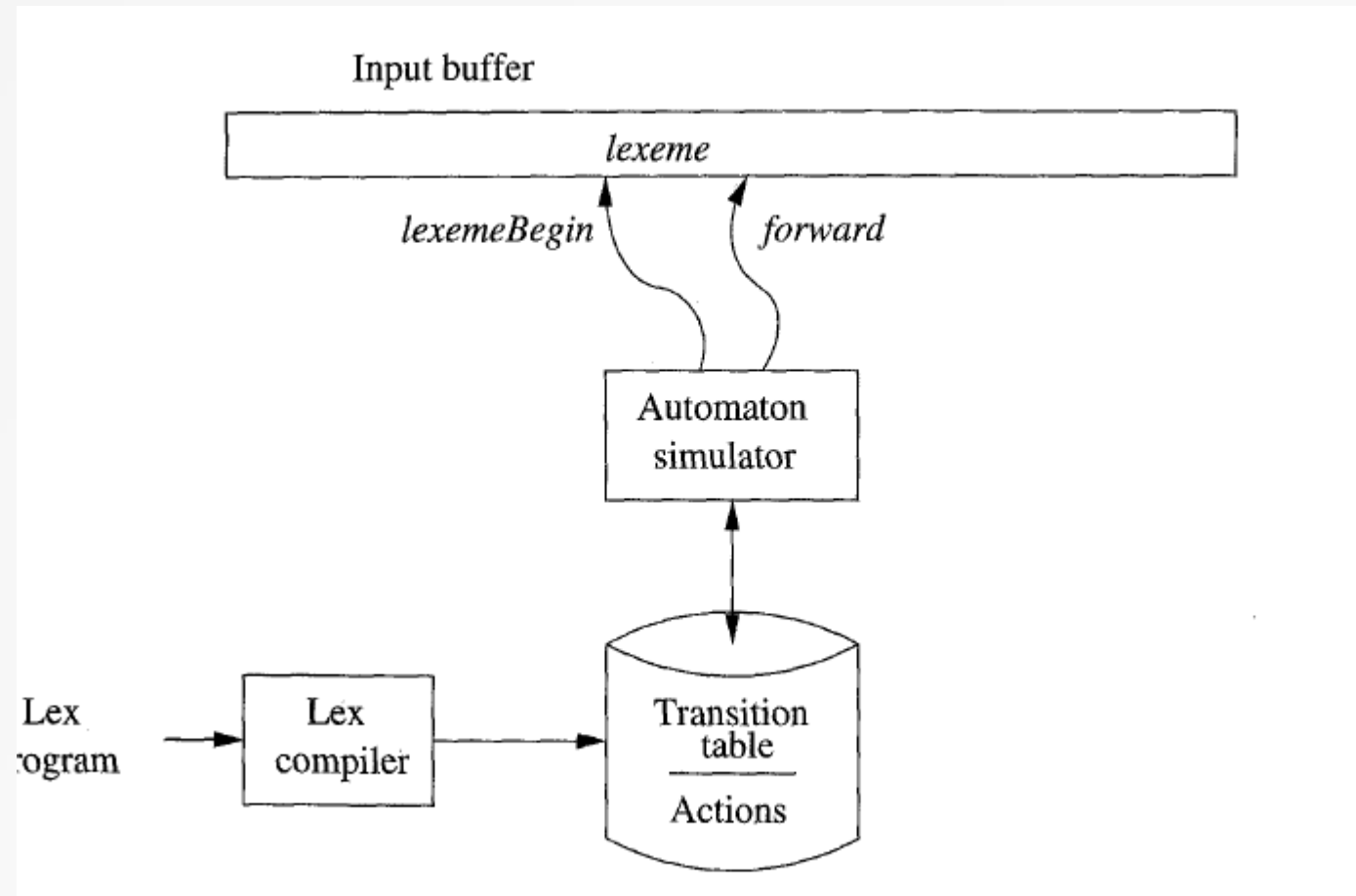
The lexical analyzer generator Lex

- A Lex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions.
- A Lex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.
- Lex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously.

PS: we will see LEX in detail in future classes

Design of LA generator using FA

Following fig shows architecture of LA generated by LEX



Design of LA generator using FA[contd]

- Lex compiler will read regexp from lex program and will generate transition table and actions.
- Component named “automaton simulator” will generate NFA or DFA.
- FA generated will read the input buffer using LexemeBegin and forward pointer.
- Based on the input, FA will identify lexeme and token is generated.

Design of LA generator using FA[contd]

- We need single automaton that will recognize lexemes matching any of the patterns in the program.
 - So we combine all the NFAs into one by introducing a new start state with epsilon transitions to each of start states of the NFAs.

NFA for patterns

a {A1 for p1}

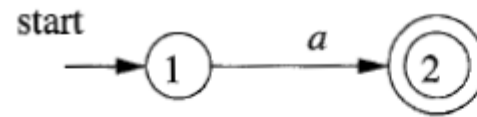
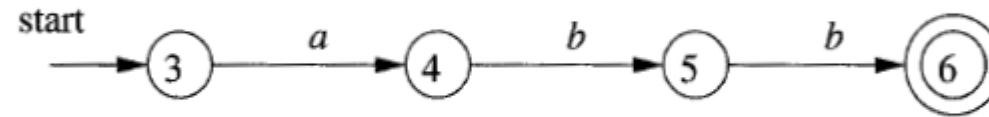
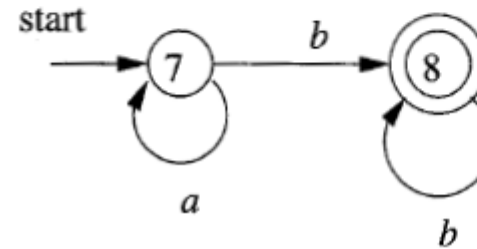


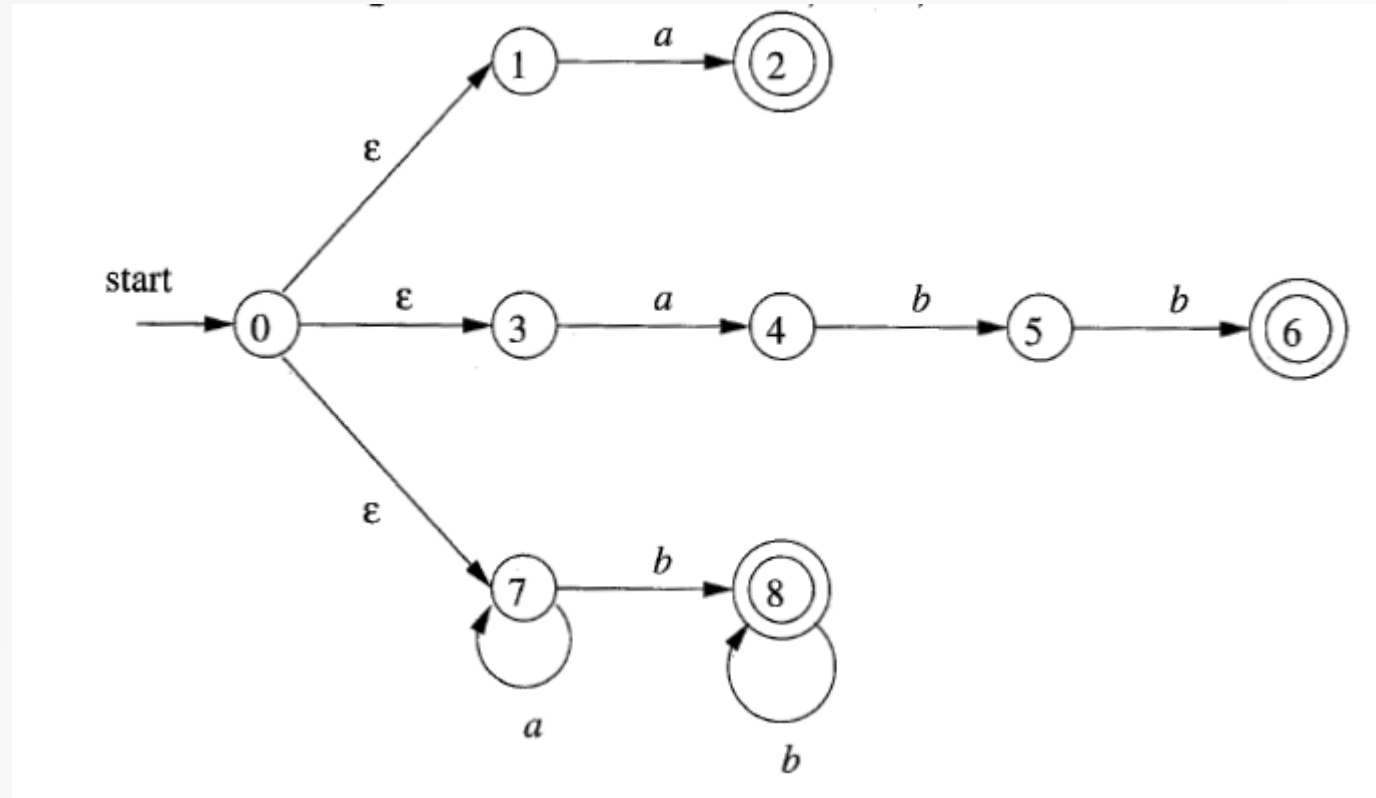
abb { A2 for p2}



a*b+ {A3 for p3}



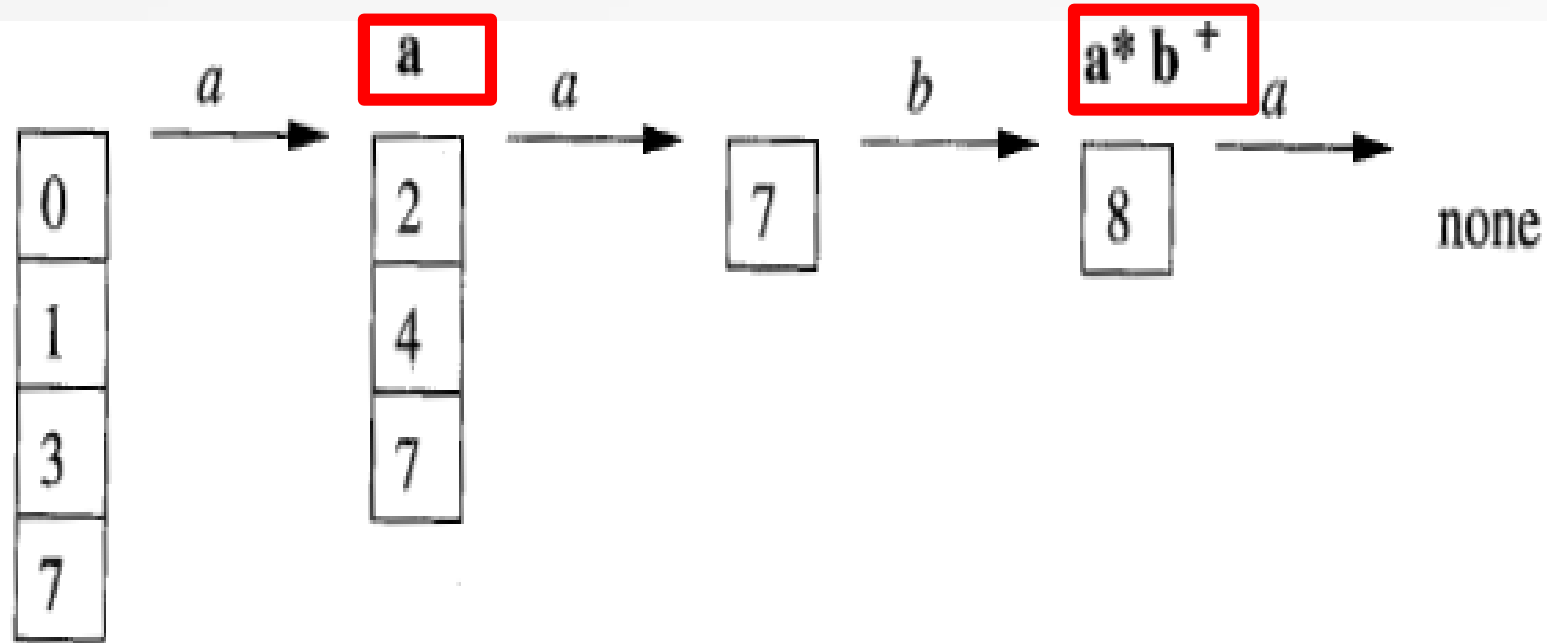
Combined NFA



Conflicts resolution in pattern matching

- Consider input as “abb”
 - Both p2 and p3 matches, which one to choose?
 - Choose the one which is listed first in the list → p2
- Consider input as “aaabbbb”
 - p1 matches ‘a’, ‘a’, ‘a’; “bbb” matches → p1p1p1p3
 - Also p1p1p2p3 is possible.
 - How to resolve this conflict?
 - Take longest matching pattern → p3

Pattern matching based on NFA



Input string:aaba

Longest prefix matched is aab

Pattern matched is p3