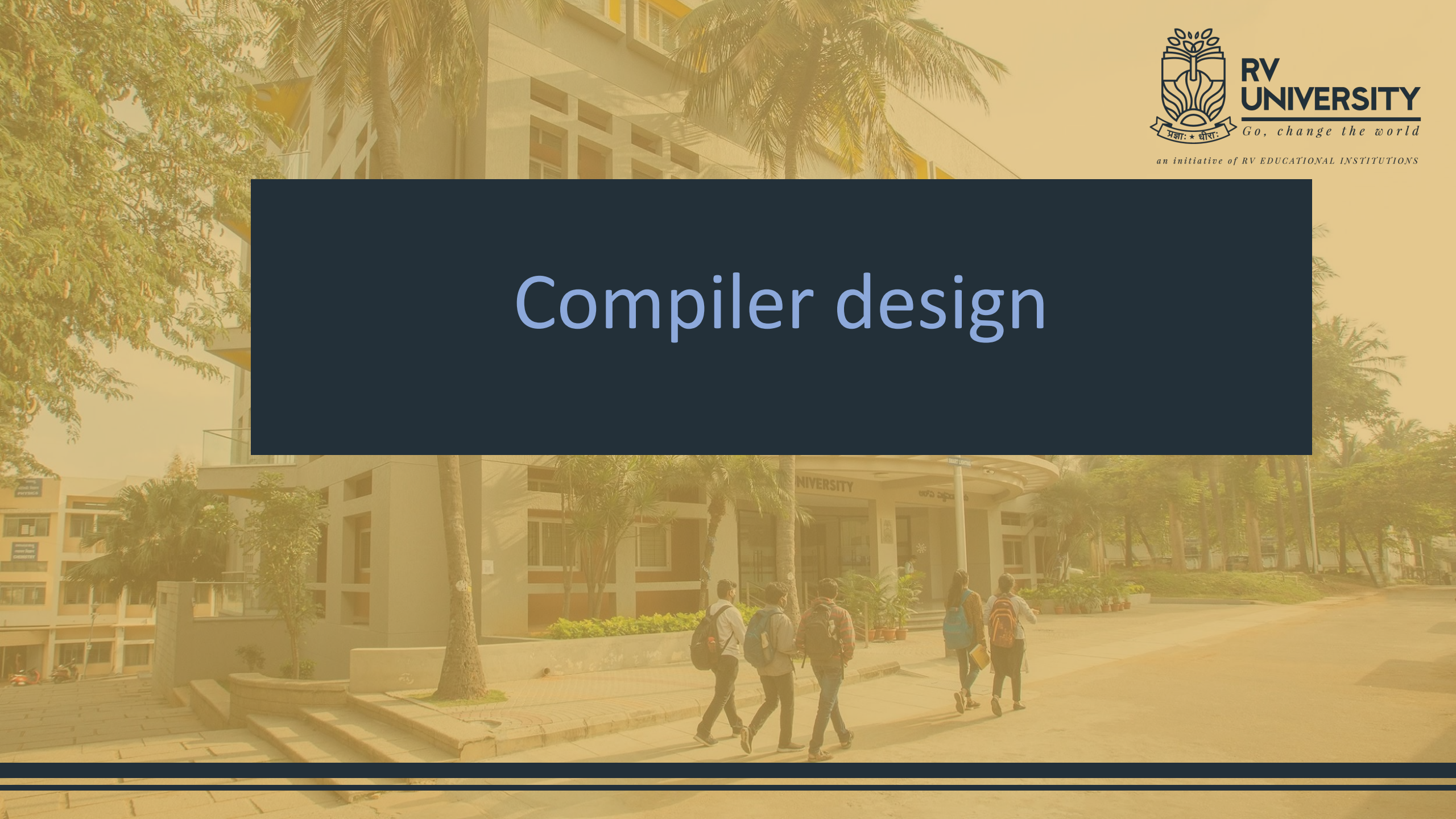
RV UNIVERSITY

Go, change the world

an initiative of RV EDUCATIONAL INSTITUTIONS

NEW-AGE GLOBAL UNIVERSITY FOR LIBERAL EDUCATION



Compiler design

Lexical Analysis

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

- The Role of t he Lexical Analyzer

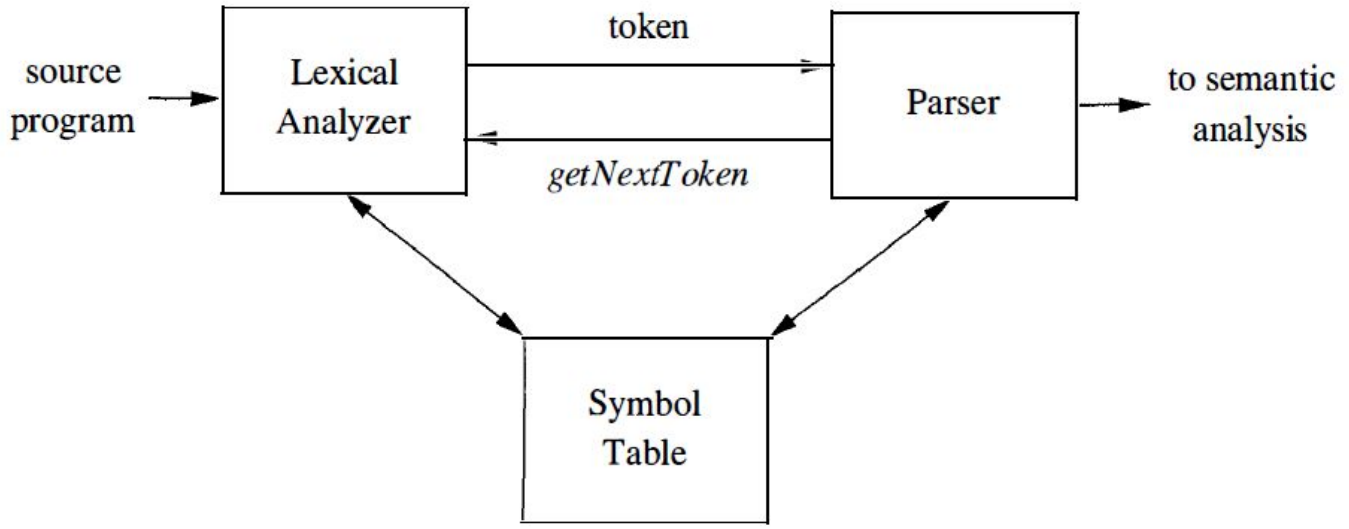


Figure 3.1: Interactions between the lexical analyzer and the parser

- Lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- Expansion of macros may also be performed by the lexical analyzer.
- Lexical analyzers are divided into a cascade of two processes
  Scanning and Lexical Analysis

```
Total=sub1+sub2*5;
Printf(”%d hai”,&x);
Int max(x,y)
Int x,y;
/*find max of x and y*/
{return(x>y?x:y);
}
```

Tokens, Patterns, and Lexemes

- A token is a pair consisting of a token name and an optional attribute value.
- A pattern is a description of the form that the lexemes of a token may take.

- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Specification of tokens

In theory of compilation regular expressions are used to formalize the specification of tokens

Regular expressions are means for specifying regular languages

Example:

letter(letter | digit)*

Each regular expression is a pattern specifying the form of strings

1. E is a regular expression, and L (E) is {t} , that is , the language whole sole member is the empty string.
2. If a is a symbol in Σ, then a is a regular expression, and L(a) = {a}, that is , the language with one string, of length one, with a in its one position.
3 If r and s are two regular expressions with languages L(r) and L(s), then
- r|s is a regular expression denoting the language L(r) UL(s), containing all strings of L(r) and L(s)
4.rs is a regular expression denoting the language L(r)L(s), created by concatenating the strings of L(s) to L(r)
5.r* is a regular expression denoting (L(r))*, the set containing zero or more occurrences of the strings of L(r)
6.(r) is a regular expression corresponding to the language L(r)

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Regular definition

$$
\begin{array}{lcl}
d_1 & \rightarrow & r_1 \\
d_2 & \rightarrow & r_2 \\
 & \cdots & \\
d_n & \rightarrow & r_n
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}
$$

Examples with Σ= {0, 1}

. (0/1)(0/1)*: All nonempty binary strings

. 0*10*10*10*: All binary strings possessing exactly three 1s

(0/1)*: All binary strings including the empty string

0(0/1)*0: All binary strings of length at least 2, starting and ending with Os

(0/1)*0(0|1)(0|1) (0/1): All binary strings with at least three characters in which the third-last character is always 0

Floating point number definition

(+|-|Ɛ) digit (digit)*(. digit (digit)* |Ɛ)((E(+|-|Ɛ) digit (digit)*)|Ɛ)

Branching statements

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

Figure 3.10: A grammar for branching statements

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]?}\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

ws □ (blank I tab I newline)+

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Figure 3.12: Tokens, their patterns, and attribute values

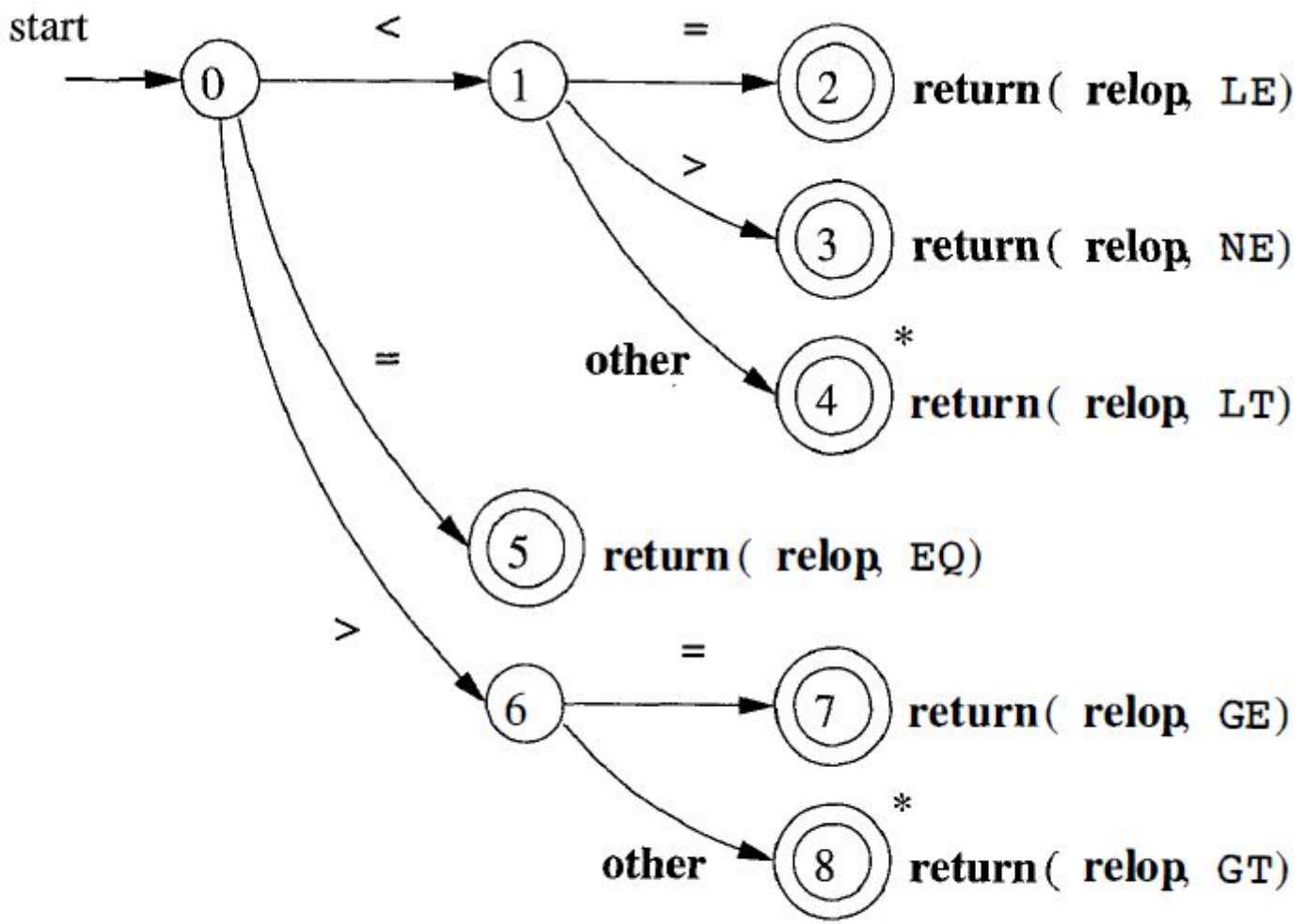


Figure 3.13: Transition diagram for **relop**

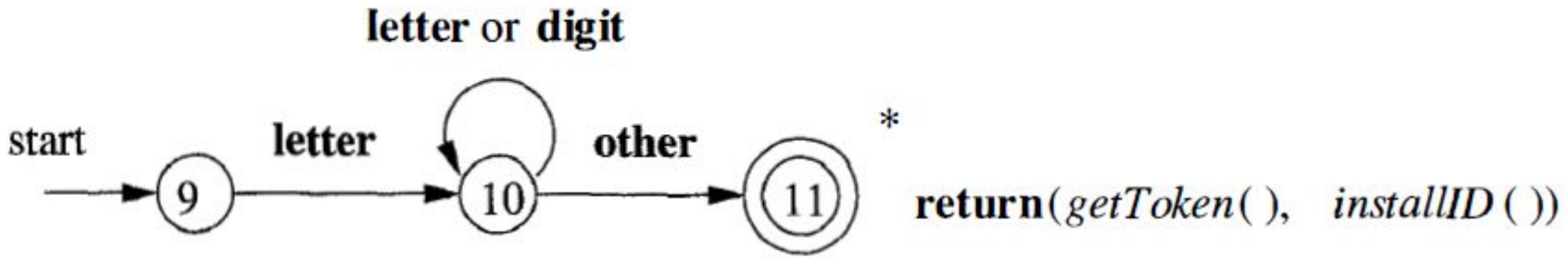
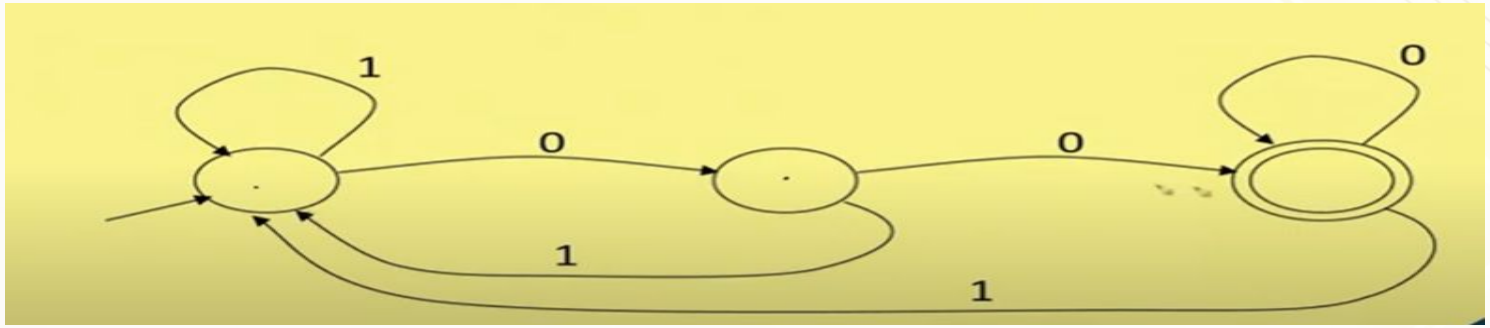


Figure 3.14: A transition diagram for **id**'s and keywords

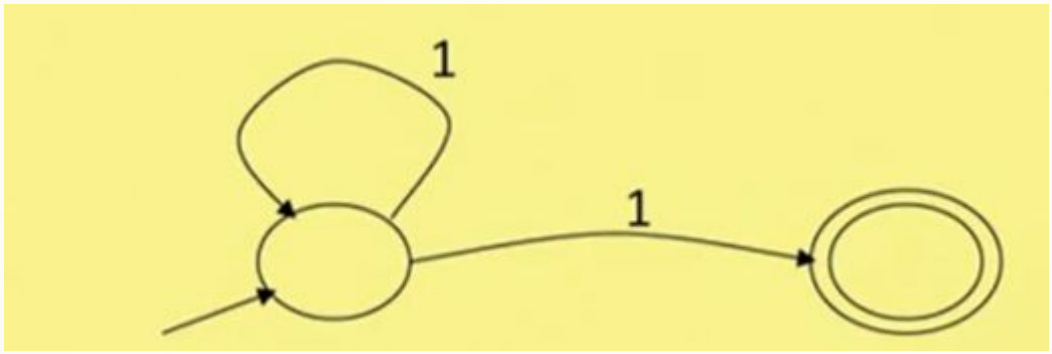
A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state
Simple Example

A finite automaton accepting any number of 1's followed by a single 0

• Alphabet: {0,1}
Any number of 1's followed by 0

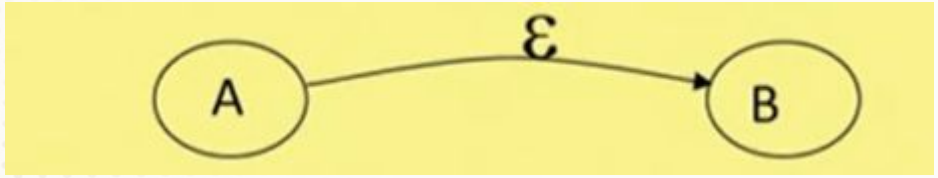

1
0
0
0
1
1

Alphabet still (0,1)



The operation of the automaton is not completely defined by the input

Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

Deterministic Finite Automata (DFA)
One transition per input per state
No ε-moves

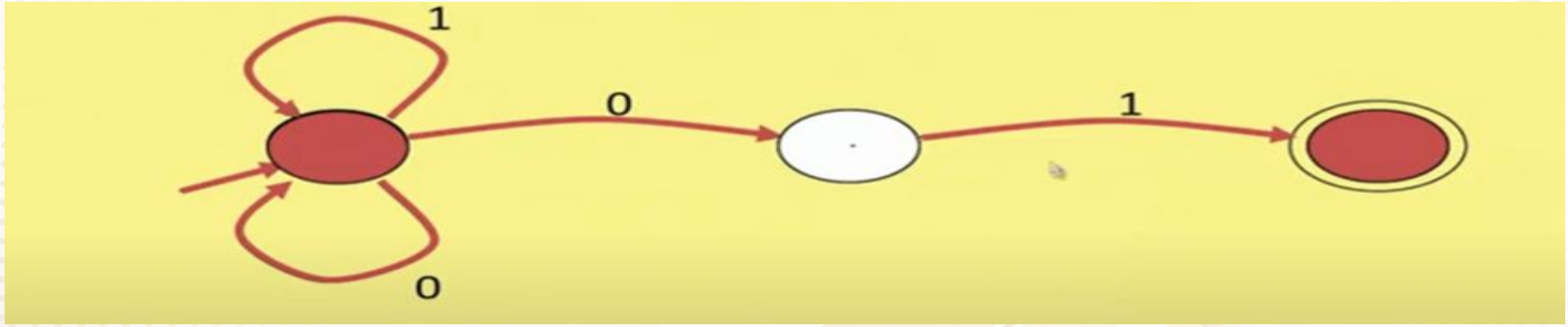
Nondeterministic Finite Automata (NFA)
Can have multiple transitions for one input in a given state
Can have ε-moves

Finite automata have finite memory
--Need only to encode the current state.

Execution of Finite Automata
A DFA can take only one path through the state graph
Completely determined by input.

NFAs can choose
Whether to make ε-moves
Which of multiple transitions for a single input to take.

Input: 1 0 1
NFA accepts if it get into final state



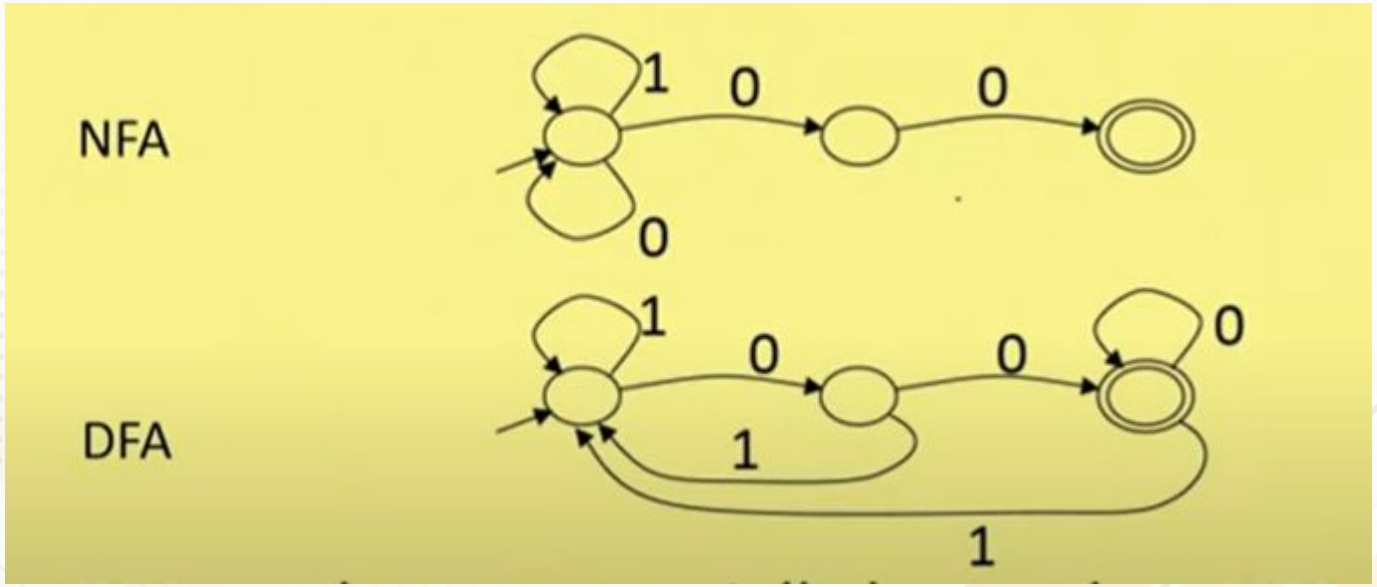
NFA vs. DFA (1)

• NFAs and DFAs recognize the same set of languages (regular languages)

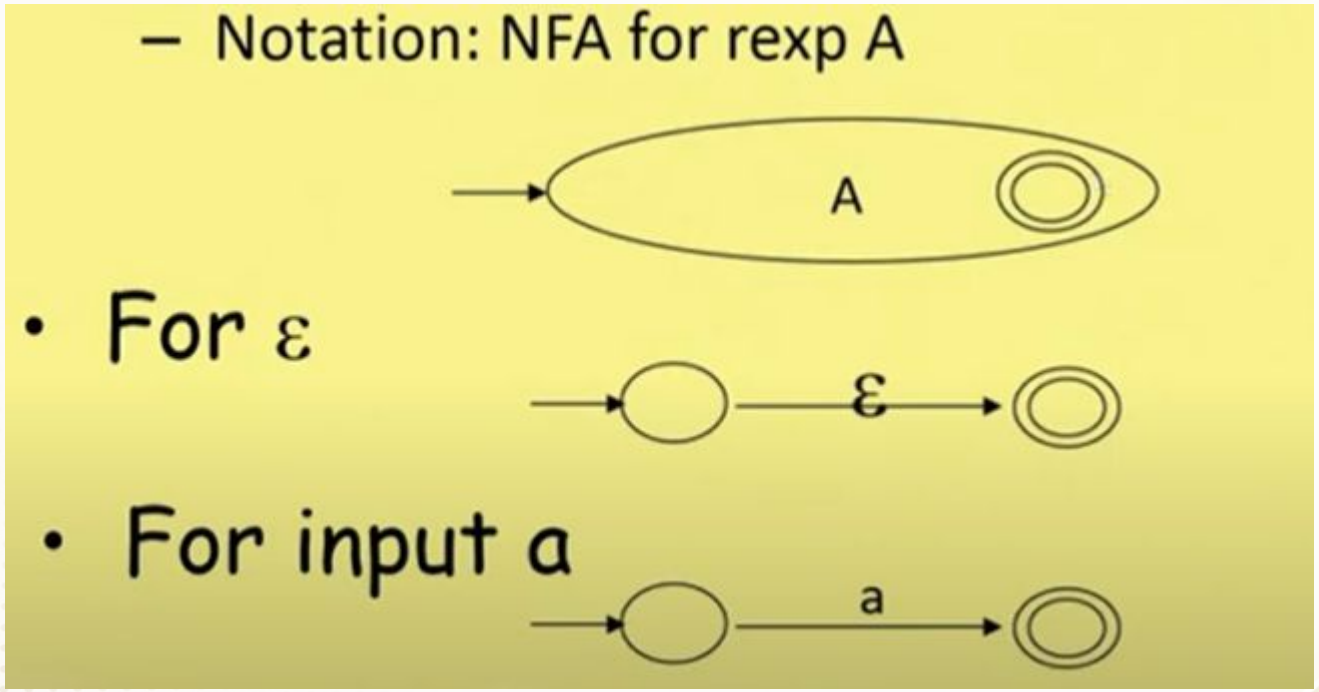
• DFAs are easier to implement

There are no choices to consider

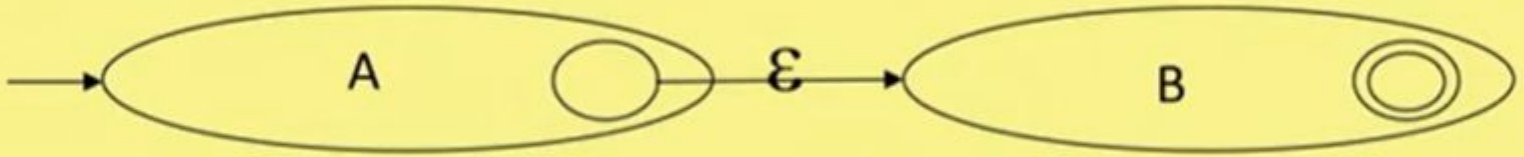
For a given language the NFA can be simpler than the DFA



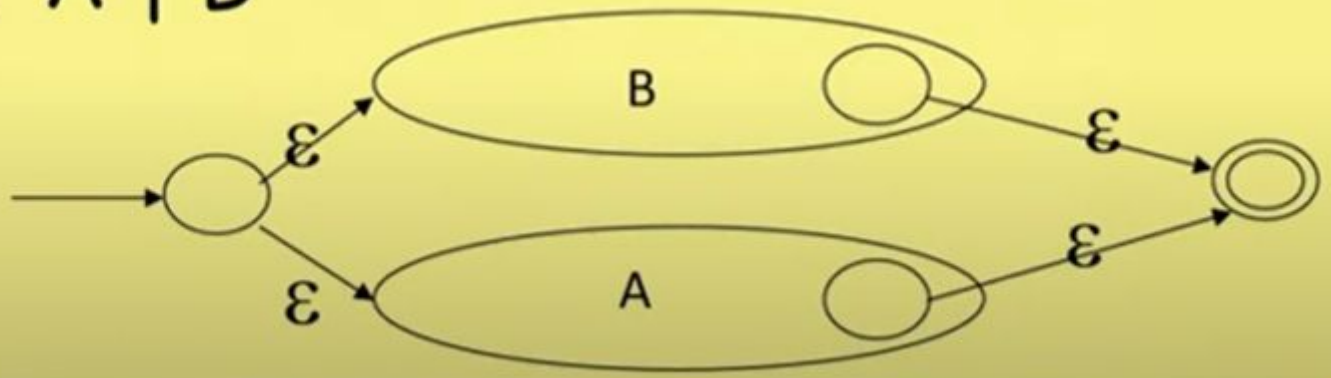
Regular expression to finite automaton



- For AB



- For A | B



For A*