CS 6109 Compiler Design

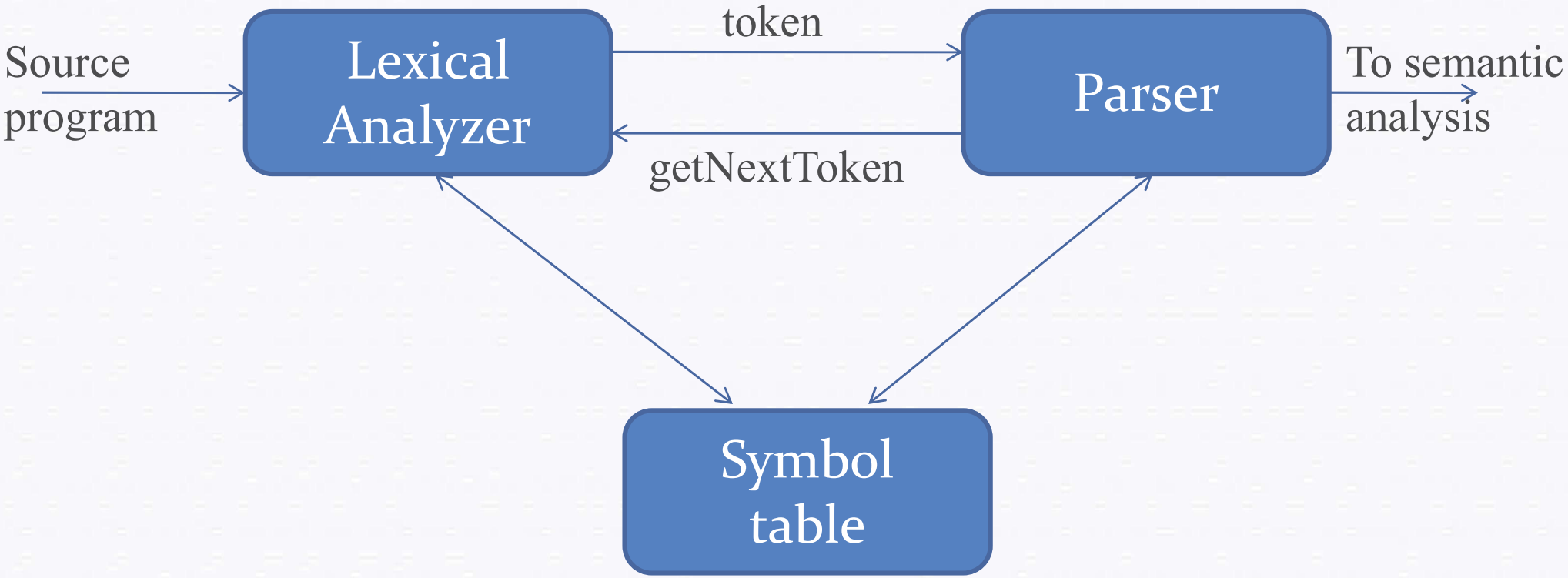
Dr. Arockia Xavier Annie R
Asst. Professor, DCSE
CEG, Anna University
Chennai -600025
Email: annie@annauniv.edu

Lexical Analysis

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Finite Automata
- Lexical analyzer generator
- Design of lexical analyzer generator

The role of lexical analyzer

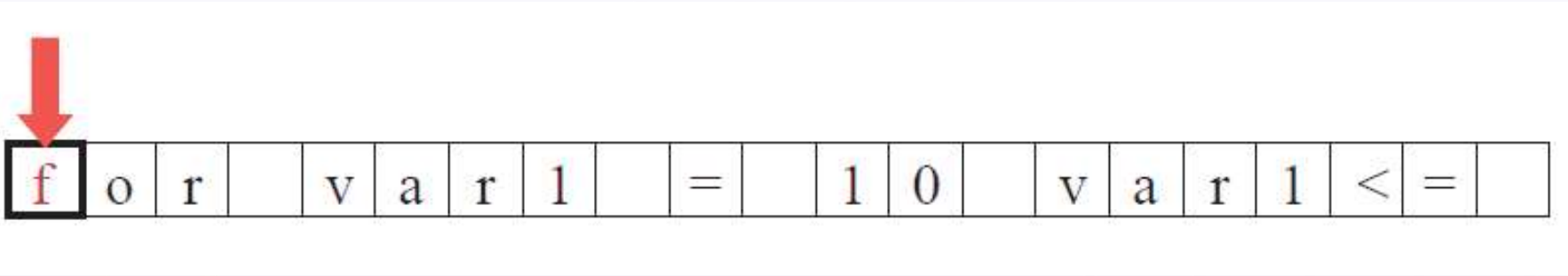


What is a Lexical Analyzer

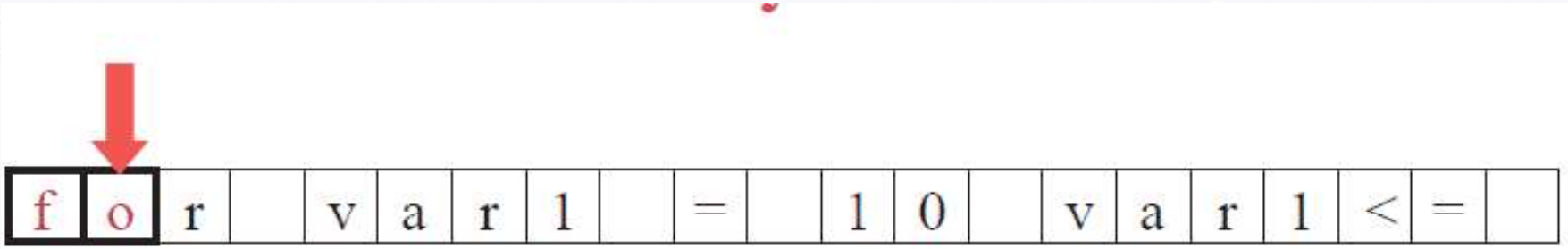
Source program text → Tokens

- Example of Tokens
  - Operators = + - > ({ := == <>
  - Keywords if while for int double
  - Numeric literals 43 4.565 -3.6e10 0x13F3A
  - Character literals 'a' '~' '\''
  - String literals "4.565" "Fall 10" "\"\" = empty"
- Example of non-tokens
  - White space space(' ') tab('\t') end-of-line('\n')
  - Comments /*this is not a token*/

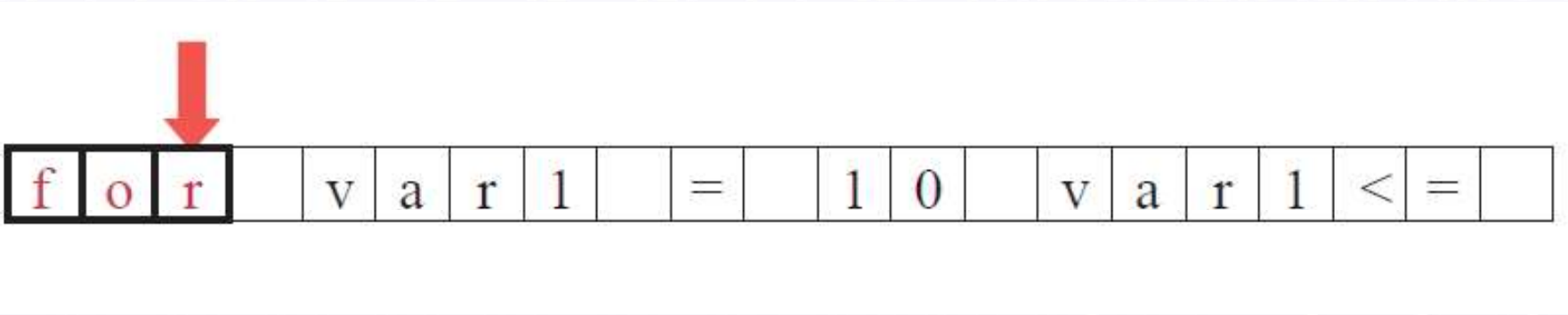
Lexical Analyzer in Action



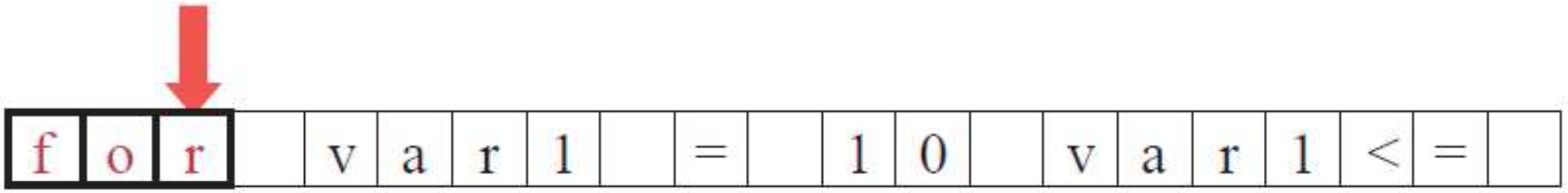
Lexical Analyzer in Action



Lexical Analyzer in Action

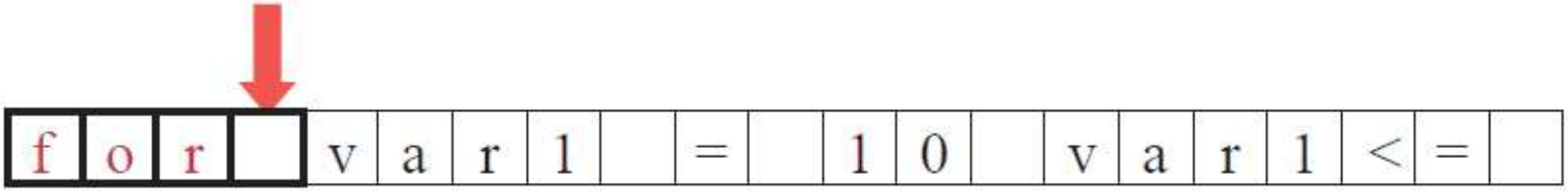


Lexical Analyzer in Action



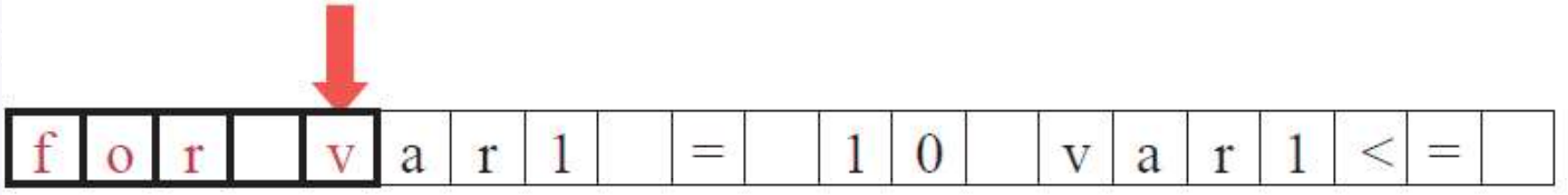
for_key

Lexical Analyzer in Action



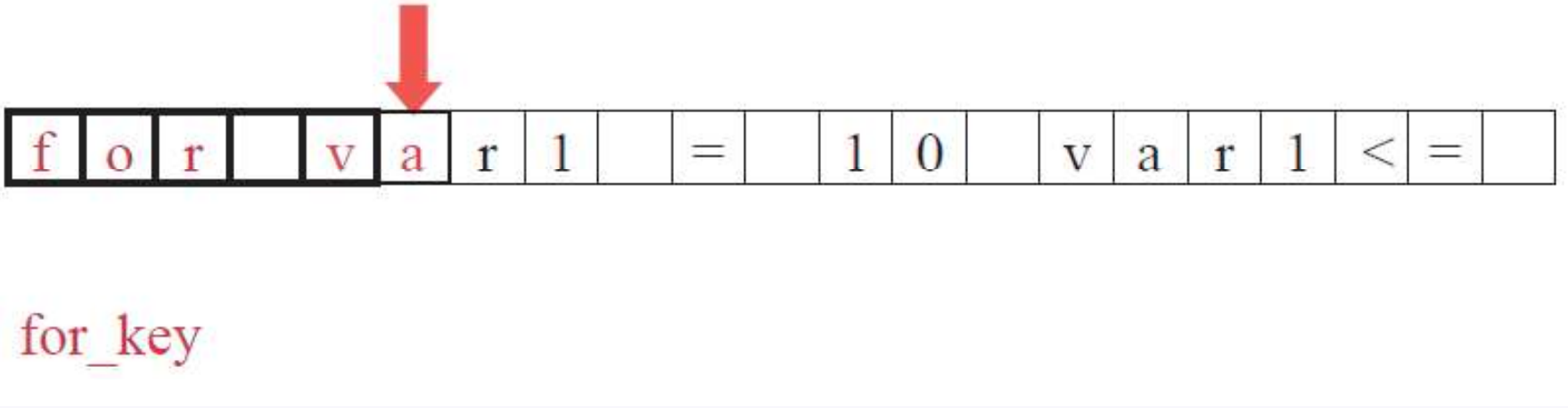
for_key

Lexical Analyzer in Action

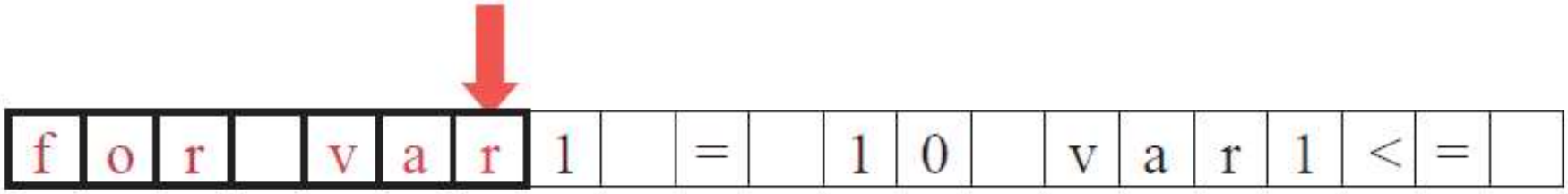


for_key

Lexical Analyzer in Action

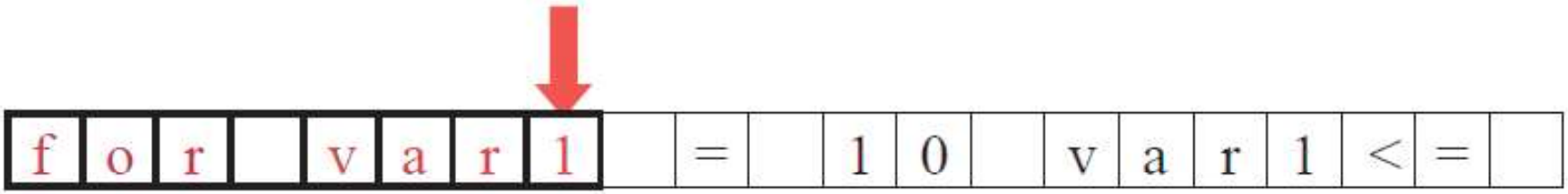


Lexical Analyzer in Action



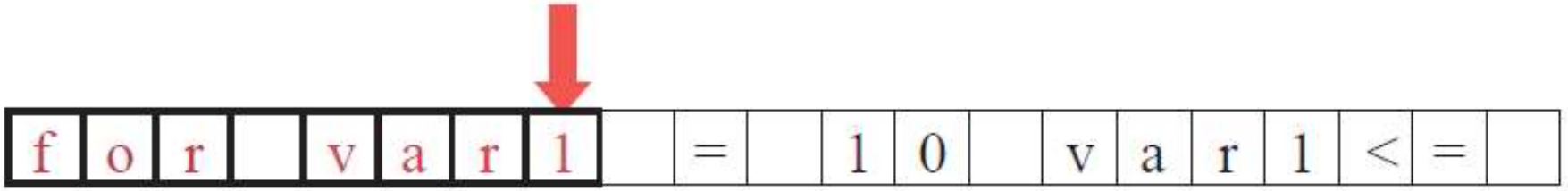
for_key

Lexical Analyzer in Action



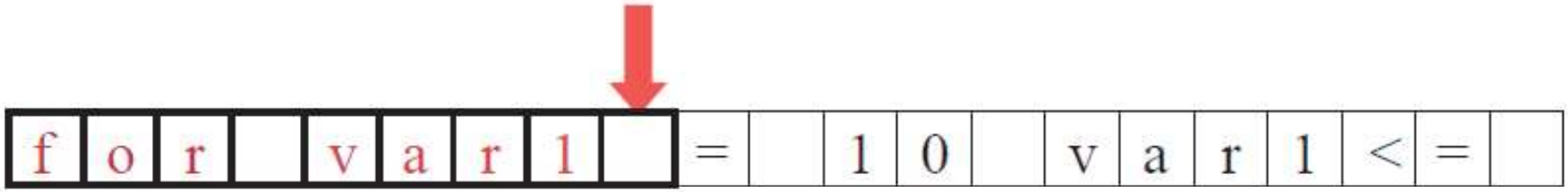
for_key

Lexical Analyzer in Action



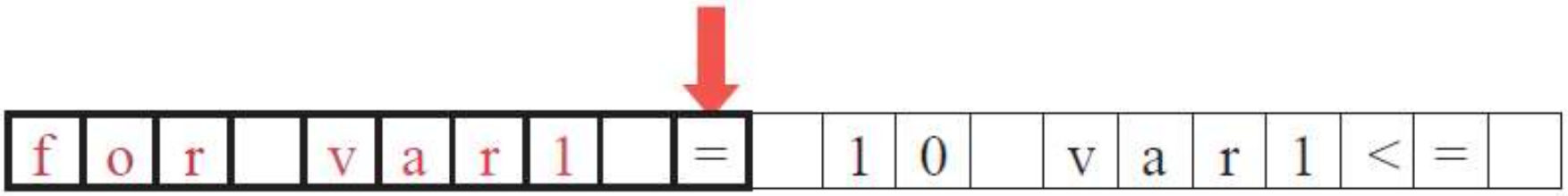
for_key ID("var1")

Lexical Analyzer in Action



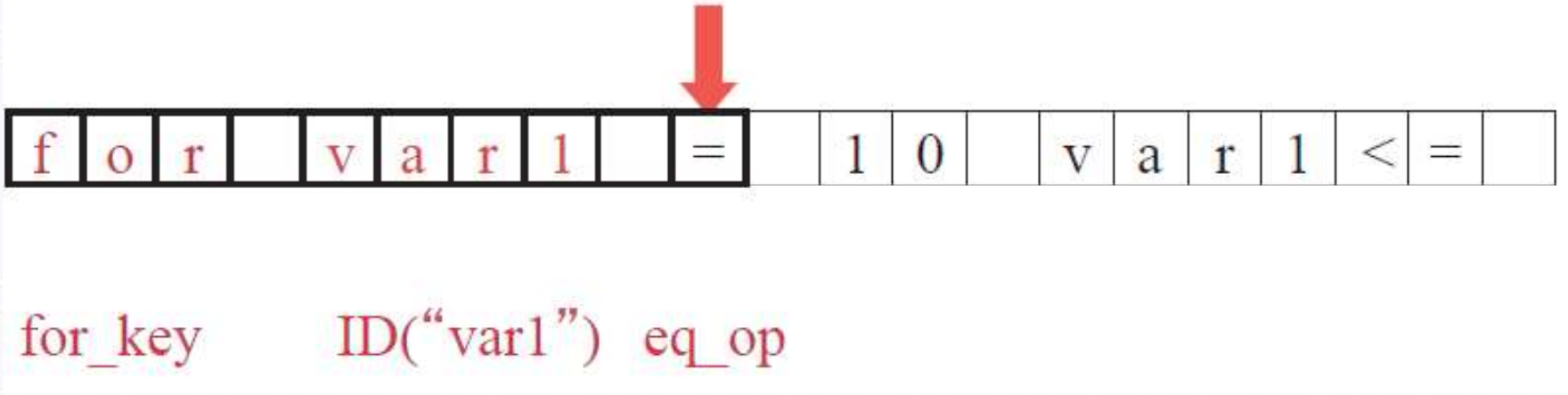
for_key ID("var1")

Lexical Analyzer in Action

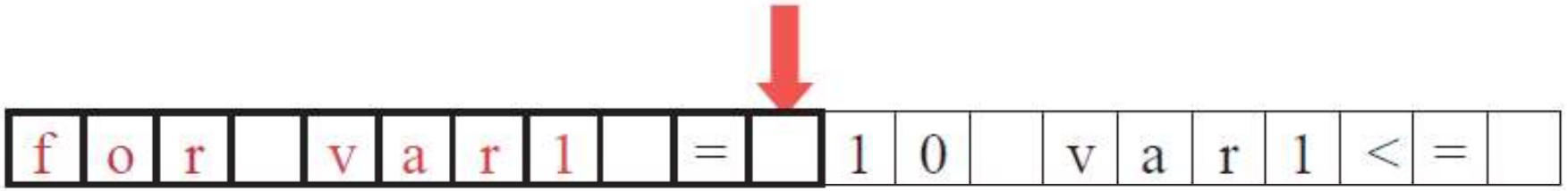


for_key ID("var1")

Lexical Analyzer in Action

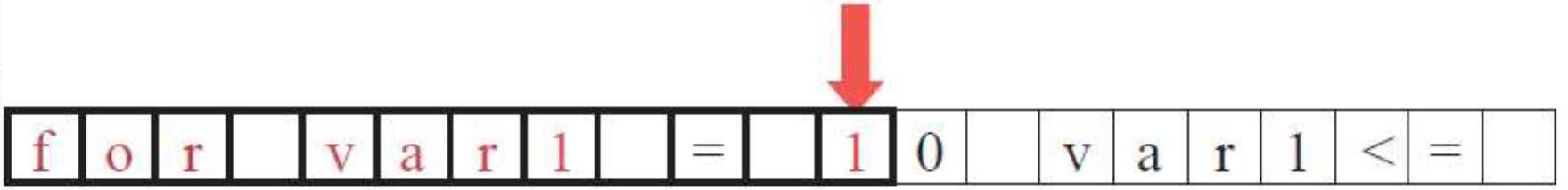


Lexical Analyzer in Action



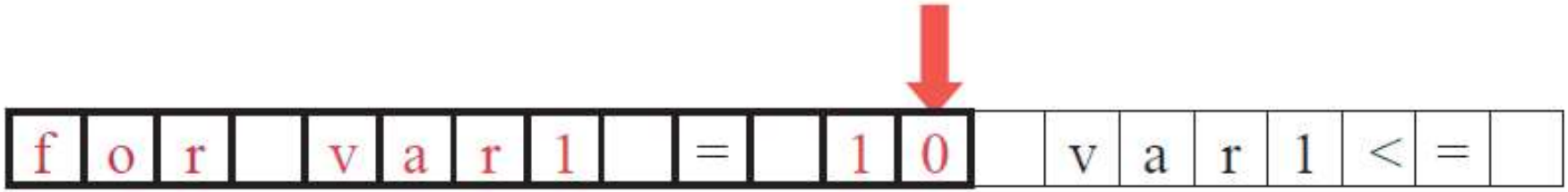
for_key ID("var1") eq_op

Lexical Analyzer in Action



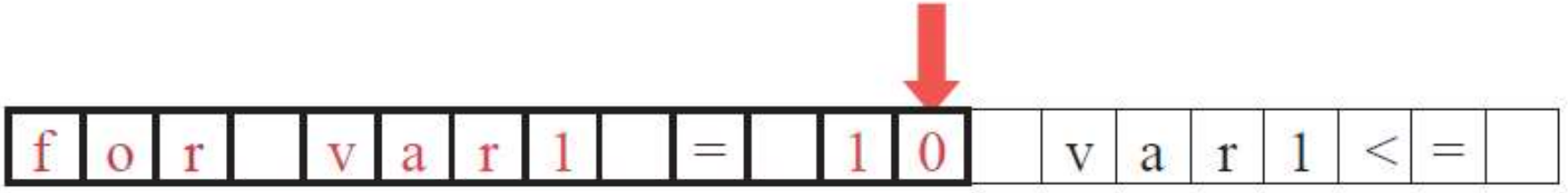
for_key ID("var1") eq_op

Lexical Analyzer in Action



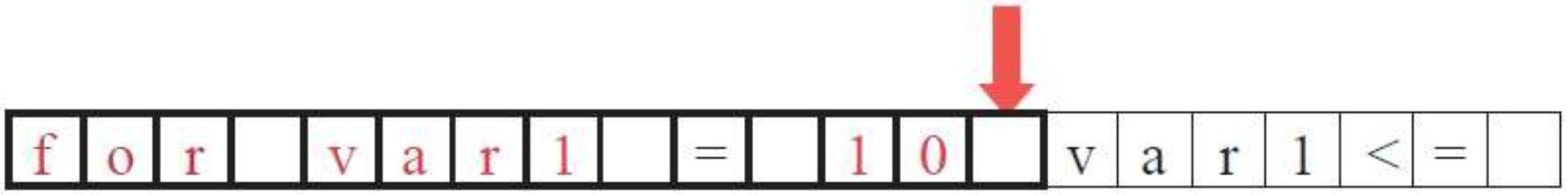
for_key ID("var1") eq_op

Lexical Analyzer in Action



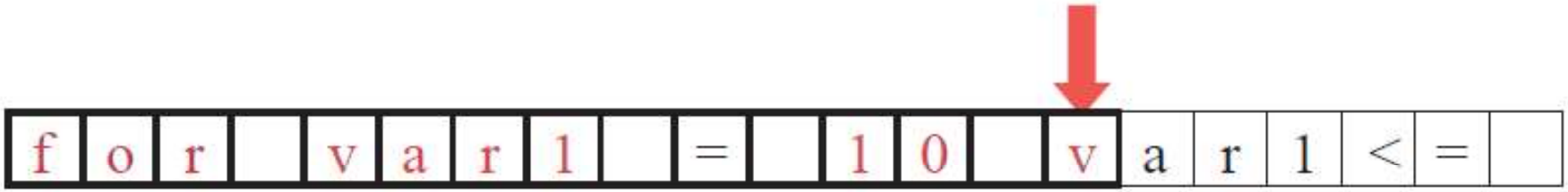
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action



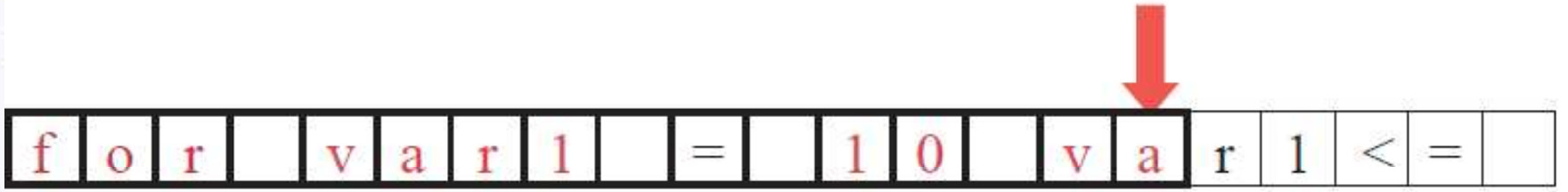
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action



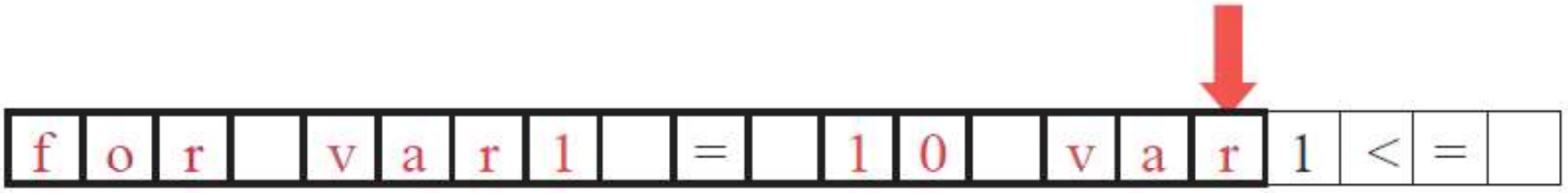
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action



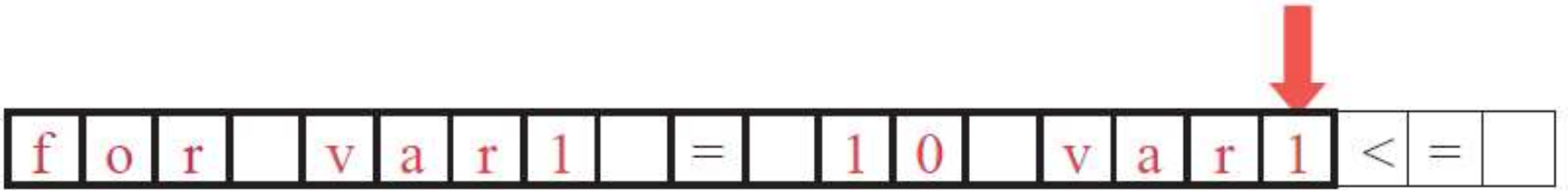
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action



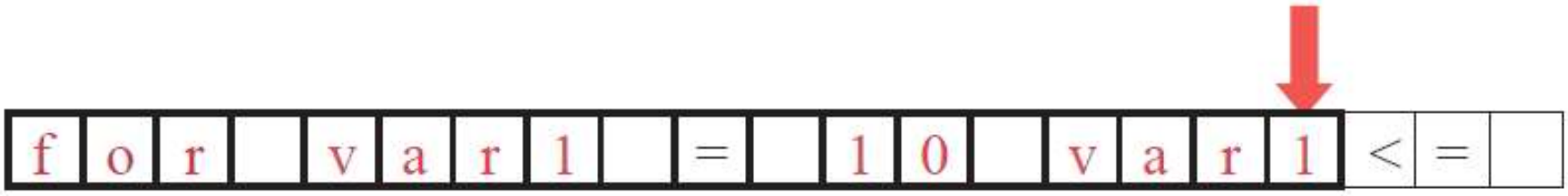
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action



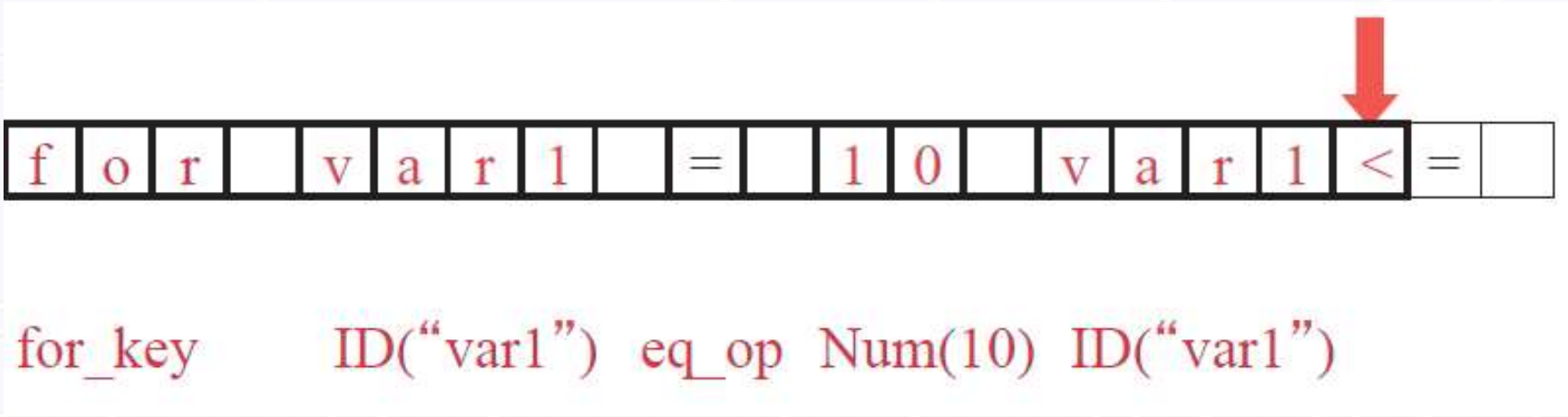
for_key ID("var1") eq_op Num(10)

Lexical Analyzer in Action

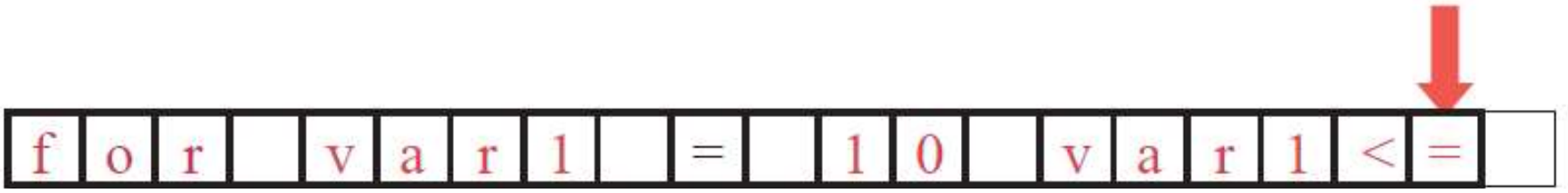


for_key ID("var1") eq_op Num(10) ID("var1")

Lexical Analyzer in Action

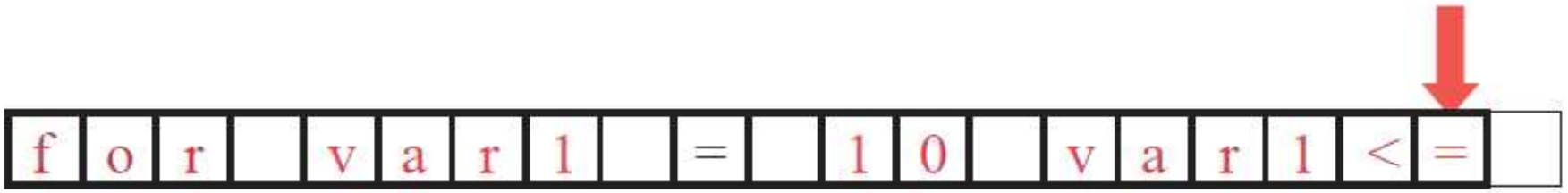


Lexical Analyzer in Action



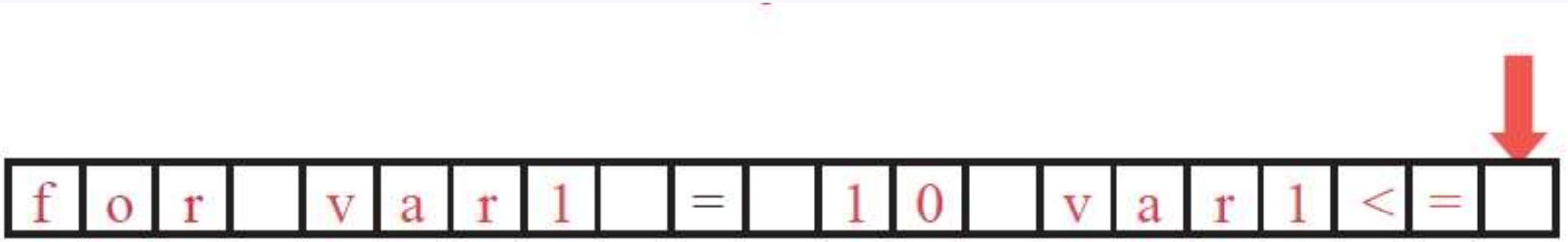
for_key ID("var1") eq_op Num(10) ID("var1")

Lexical Analyzer in Action



for_key ID("var1") eq_op Num(10) ID("var1") le_op

Lexical Analyzer in Action



for_key ID("var1") eq_op Num(10) ID("var1") le_op

Lexical Analyzer in Action

- Partition Input Program Text into Subsequence of Characters Corresponding to Tokens
- Attach the Corresponding Attributes to the Tokens
- Eliminate White Space and Comments

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token 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

Example

Token	Pattern	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"values are"

for (i = 0; i<10; i++)

Attributes for tokens- Example

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Finite Automata
- Lexical analyzer generator
- Design of lexical analyzer generator

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)* - Specifies 'identifier' token.
- Each regular expression is a pattern specifying the form of strings

Token

- Token represents a **set of strings** described by a pattern.
- Since a token represents more than one lexeme, additional information should be held for that specific lexeme. This additional information is the *attribute* of the token.
- Some attributes:
  - <id, attr> where attr is pointer to the symbol table
  - <assgop,_> no attribute is needed (if there is only one assignment operator)
  - <num, val> where val is the actual value of the number.
- Token type and its attribute uniquely identifies a lexeme.
- ***Regular expressions*** are widely used to specify patterns.

Terminology of Languages (1/2)

- **Alphabet** : a finite set of symbols (ASCII characters)

- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.

Terminology of Languages (2/2)

- **Language**: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - s^n = s s s .. s (n times) $s^0 = \varepsilon$

Operations on Languages

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Example

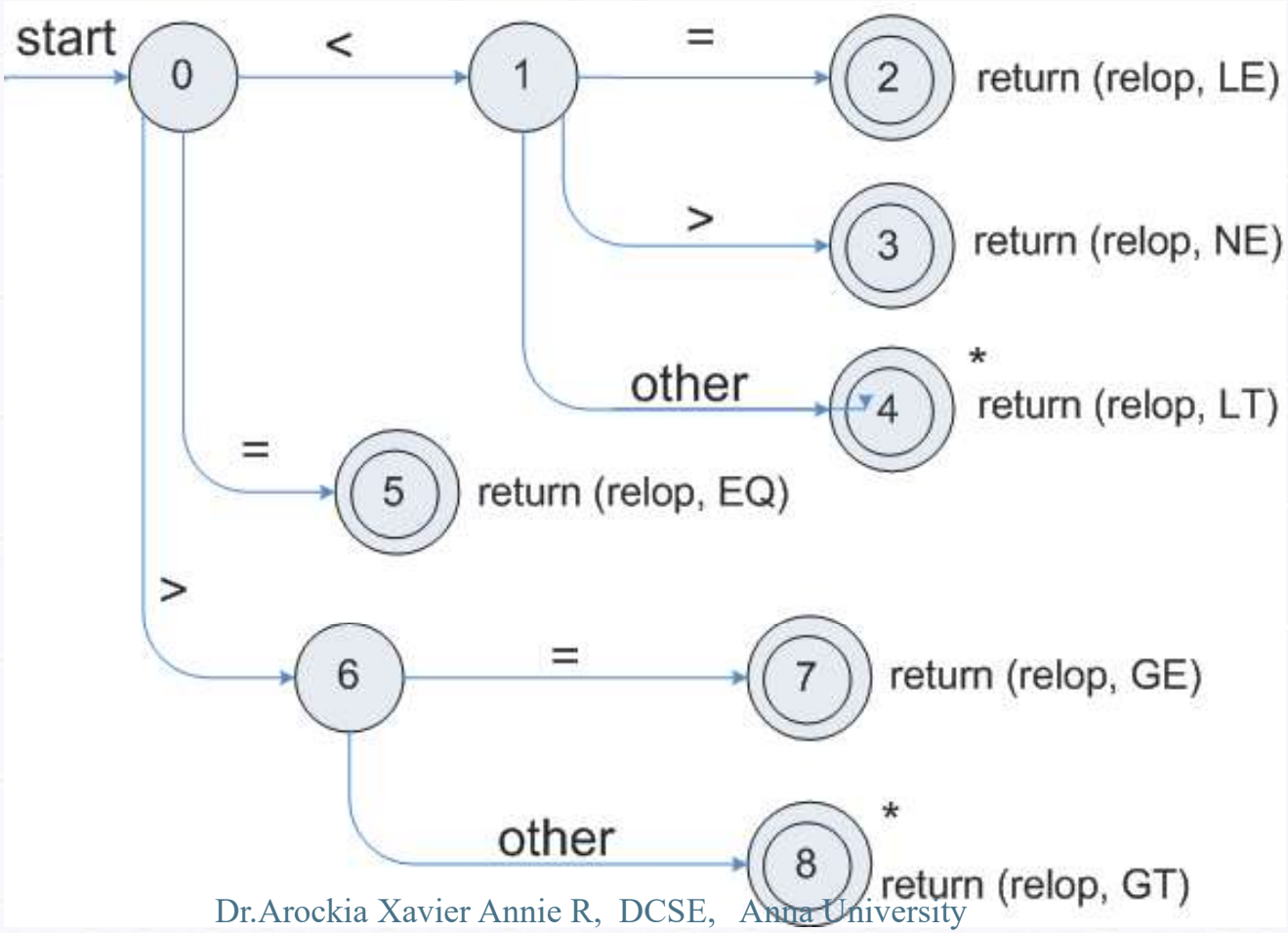
- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Finite Automata
- Lexical analyzer generator
- Design of lexical analyzer generator

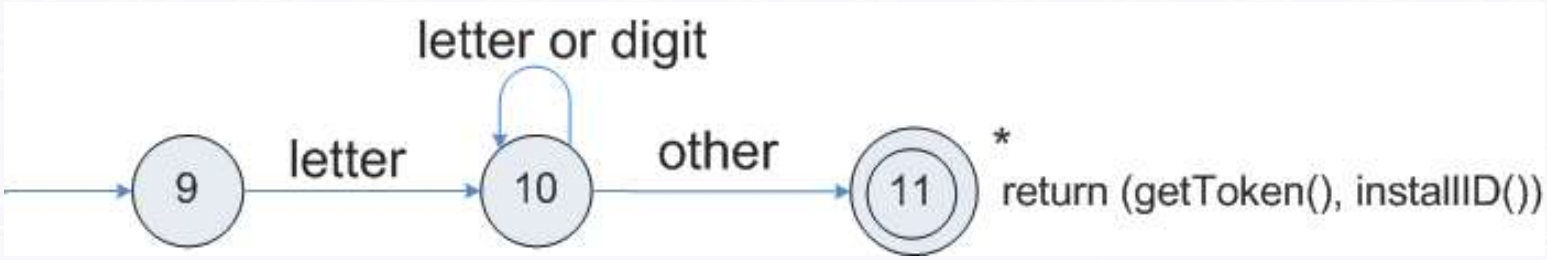
Transition diagrams

- Transition diagram for relop



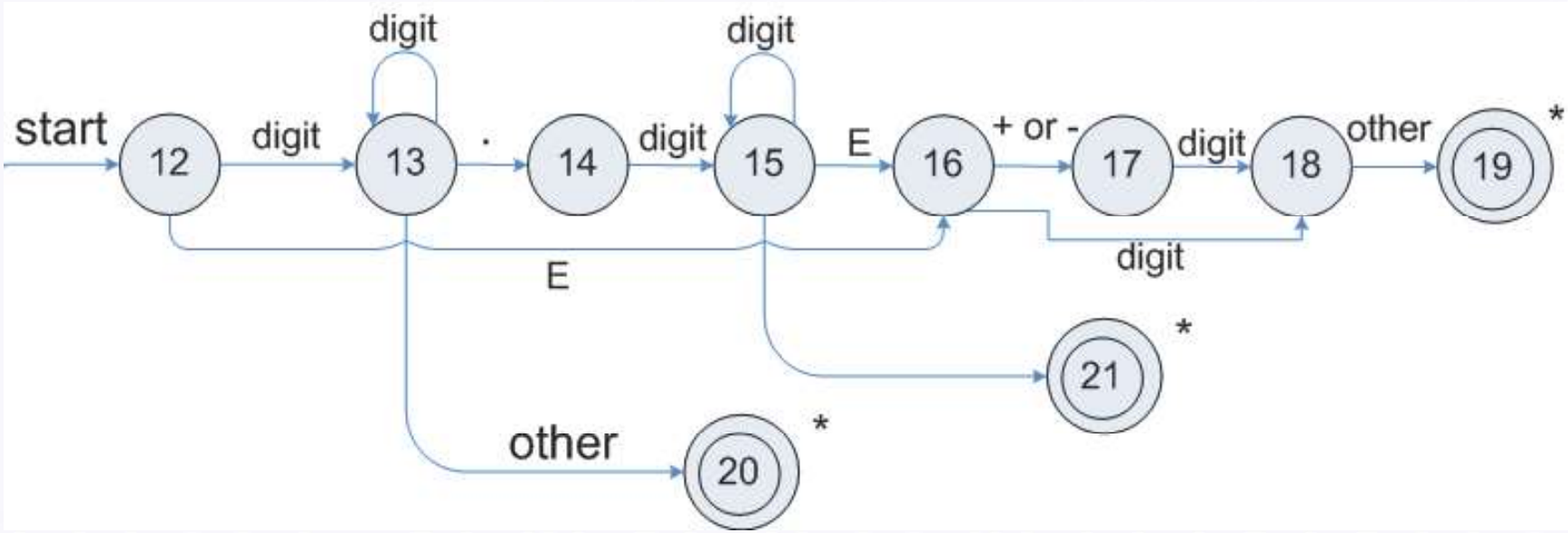
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



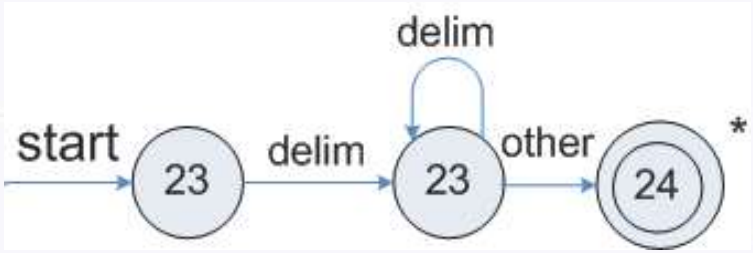
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Algebraic Properties of RE

AXIOM	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r\|\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$
- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon, 0, 00, 000, 0000, \ldots\}$
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.
- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols

$\ldots$ $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

not recursive

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter $\rightarrow$ A | B | … | Z | a | b | … | z

  digit $\rightarrow$ 0 | 1 | … | 9

  id $\rightarrow$ letter (letter | digit) *

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|…|Z|a|…|z) ((A|…|Z|a|…|z) | (0|…|9)) *

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Finite Automata
- Lexical analyzer generator
- Design of lexical analyzer generator

Finite Automata

- A *recognizer* for a *language* is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton.*
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.

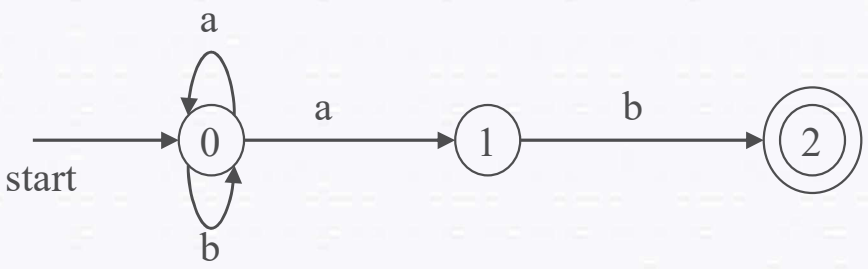
Finite Automata

- NFA or DFA?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used in lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_o - a start (initial) state
  - F – a set of accepting states (final states)
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (Example)



Transition graph of the NFA

0 is the start state s_0
{2} is the set of final states F
$\Sigma = \{a,b\}$
S = {0,1,2}

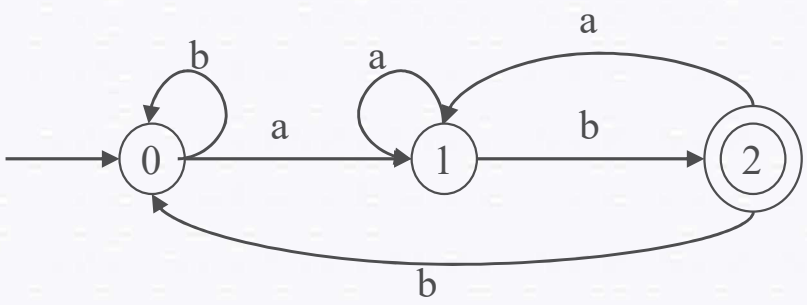
Transition Function:

	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

The language recognized by this NFA is $(a|b)^*$ a b

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA
  - no state has ε- transition
  - for each symbol a and state s, there is at most one labeled edge a leaving s.
    i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by this DFA is also $(a|b)^*$ a b

Implementing a NFA

```
S ← ε-closure({s0})          { set all of states can be accessible from s0 by
                               ε-transitions }
c ← nextchar
while (c != eos) {
    begin
        s ← ε-closure(move(S,c))     { set of all states can be
                                       accessible from a state in S
        c ← nextchar                   by a transition on c }
    end
if (S∩F != Φ) then           { if S contains an accepting state }
    return "yes"
else
    return "no"
```

Implementing a DFA

- Le us assume that the end of a string is marked with a special symbol (say eos). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0                      { start from the initial state }
c ← nextchar                { get the next character from the
                                        input string }
while (c != eos) do         { do until the endof the string }
   begin
        s ← move(s,c)       { transition function }
        c ← nextchar
   end
if (s in F) then            { if s is an accepting state }
   return "yes"
else
   return "no"
```

Recap

- Lexical Analyzer
  - Scanner
  - Tokens – Patterns – Lexemes
  - Transition diagrams
  - Regular Expressions and Regular Definitions
  - Finite Automata
    - NFA and DFA
- Next Class
- RE to NFA / NFA to DFA / RE to DFA

Questions???