CS4031
Compiler Construction
Lecture 3

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Token

- Lets take the example in which we count the numbers of tokens

```
int main ( )
{
//here we declare the variables
int a1 = 30 , b1 = 40; // variable declaration
if ( b1 > a1)
return b1;
else
return a1;
}
```

Token

- Lets take the example in which we count the numbers of tokens

int main ( )

We have four tokens

int

main

(

)

Token	Token count
int	1
Main	2
(	3
)	4

Token

- Lets take the example in which we count the numbers of tokens

```
int main ( )
{
//here we declare the variables
int a1 = 30 , b1 = 40; // variable declaration
```

Here we ignore the comments and white spaces

Error Recovery in Lexical Analysis

Generally, errors often detected in a lexical analysis are as follows:

- Numeric literals that are too long
- Long identifiers (often a warning is given)
- Input characters that are not in the source language

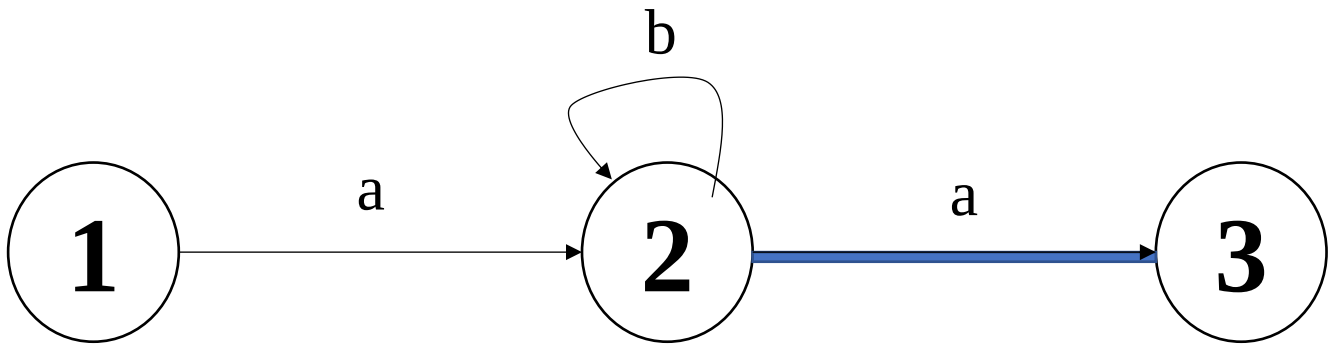
Strategy to implement the lexical analyzer

- Designing a lexical analyzer either by hand or automated tools mainly involves two steps

1. Describe rules for tokens using regular expressions.
2. Design a recognizer for such rules, that is, for tokens. Designing a recognizer corresponds to converting regular expressions to Finite Automata. The processing can be speeded if the regular expression is represented in Deterministic Finite Automata. This involves the following steps:

- Convert regular expression to NFA with ε
- Convert NFA with ε to NFA without ε
- Convert NFA to DFA

Table Encoding



	A	B
1	2	Error
2	3	2
3	error	Error

Simulating FAs

```
trans_table[NSTATES][NCHARS];
accept_states[NSTATES];
state = INITIAL;
while(state != err){
 c = input.read();
 if(c == EOF ) break;
 state=trans_table[state][c];
}
return accept_states[state];
```

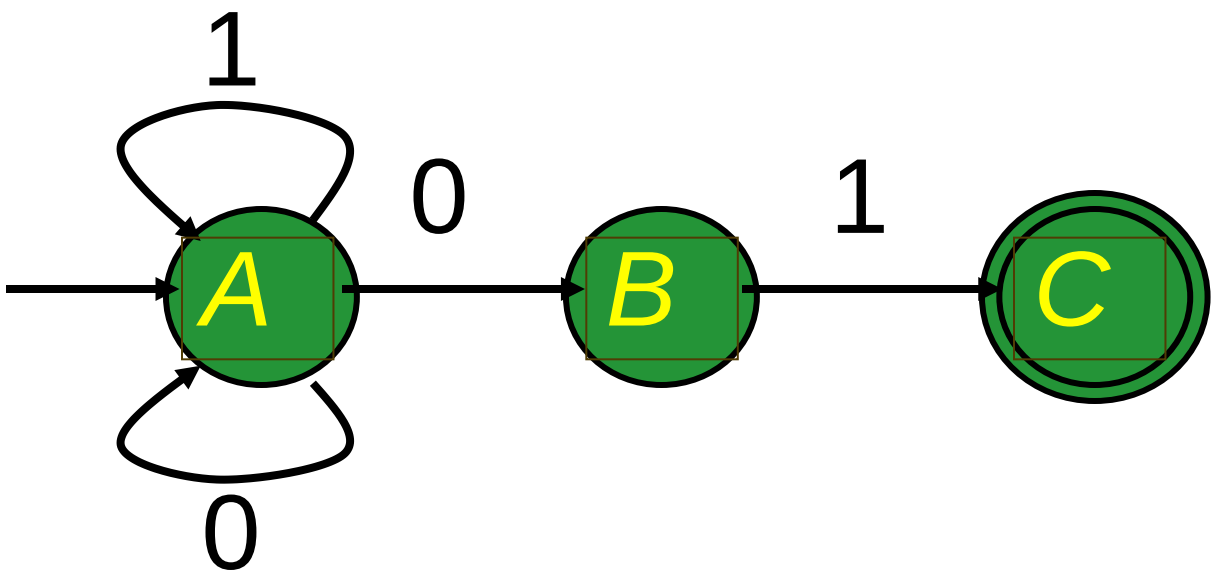
Some Previous Concept

- In this step, we revise the concept of automata

1. DFA
2. NFA
3. Epislon NFA
4. NFA to DFA
5. Minimization of DFA

NFA

- NFA can get into multiple states
- Rule: NFA accepts if it can get in a final state



Execution of DFA

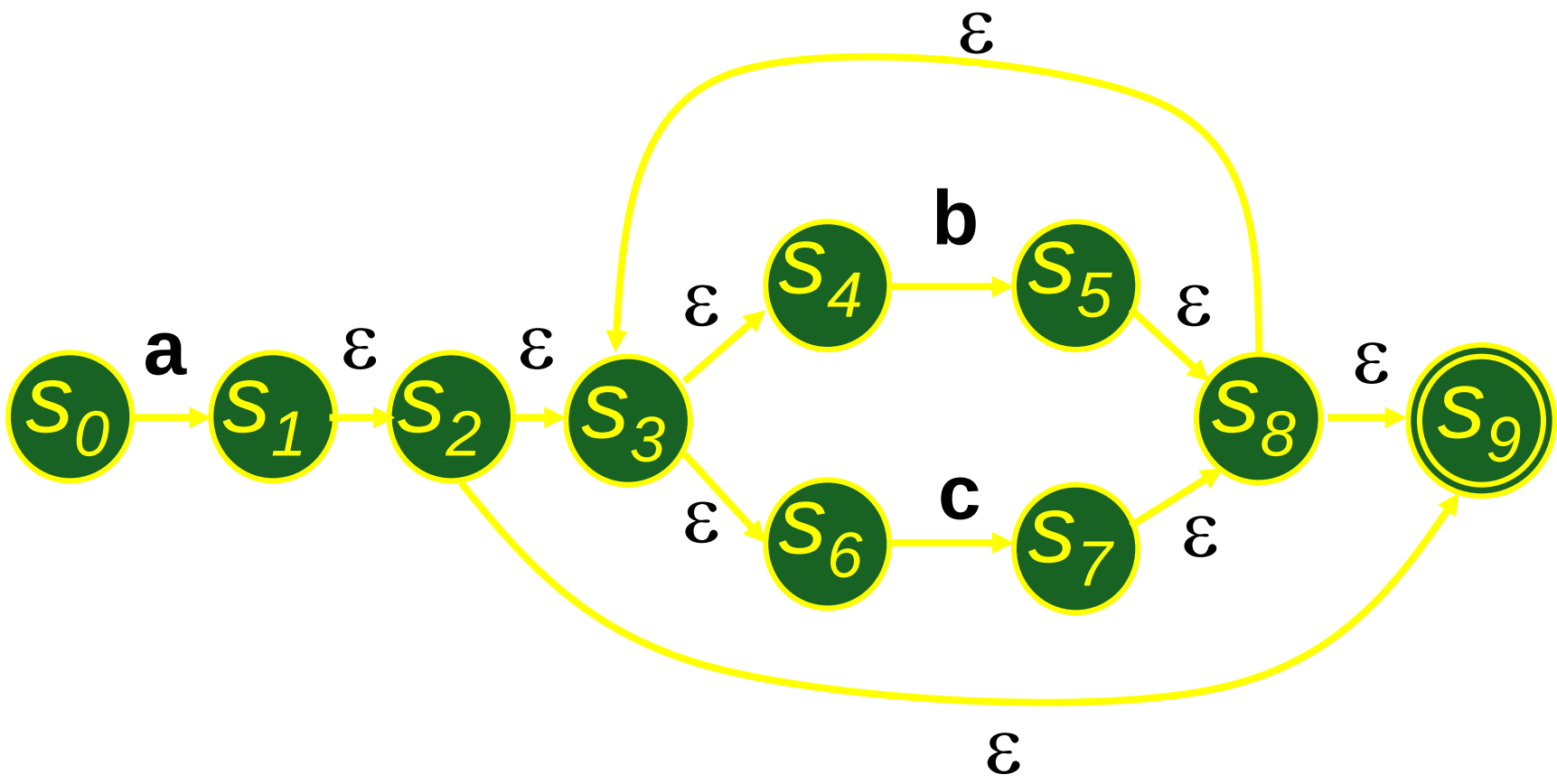
A DFA

- can take *only one path* through the state graph.
- Completely determined by input.

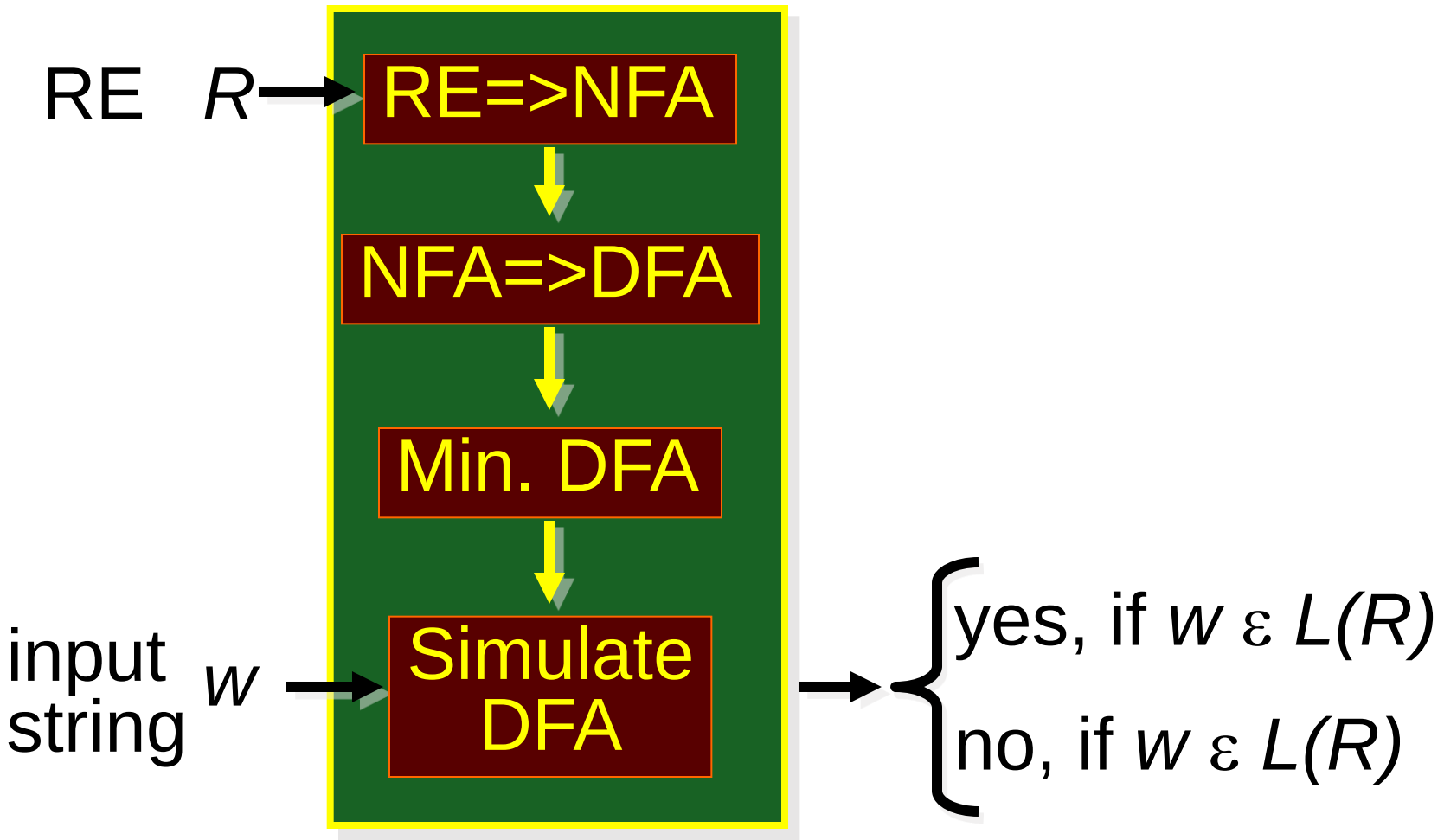
NFA vs DFA

- 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.

Construction of RE to NFA a(b+c)*



Optimized Acceptor



Using flex

- Provide a specification file
- Flex reads C or C++ output file contain the scanner.
- The file consist of three sections

1. C or C++ and flex definition
2. Token definitions and actions
3. User code

Problem scanner

- In order to recognize a token, a scanner has to look ahead several characters from the current character many times.
- For example
- “char” is a keyword in C, while the term “chap” may be a variable name.
- When the character “c” is encountered, the scanner cannot decide whether it is a variable, keyword, or function name until it reads three more characters.

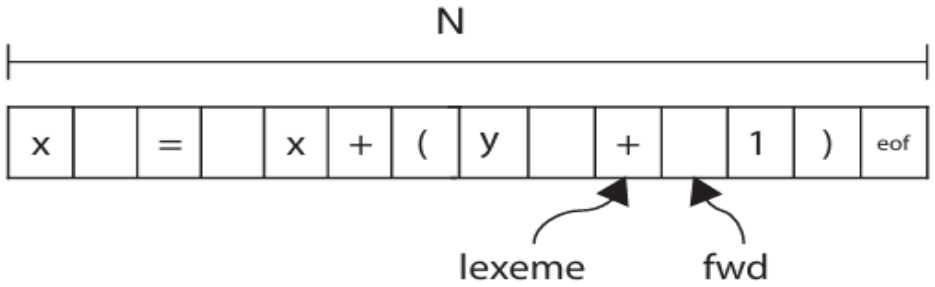
Input buffering

- Input buffering is an important concept in compiler design that refers to the way in which the compiler reads input from the source code.

Need of Buffering

- In many cases, the compiler reads input one character at a time, which can be a slow and inefficient process.

Input buffering is a technique that allows the compiler to read input in larger chunks, which can improve performance and reduce overhead.



Buffering

- Input string to source code (as we discussed previously we stored the whole string in the buffer for lexical analyzer phase to create the tokens)

```
Int ( )
{

}
```

Token Lexemes												
i	n	t		m	a	i	n	(	)		{	}

Buffering

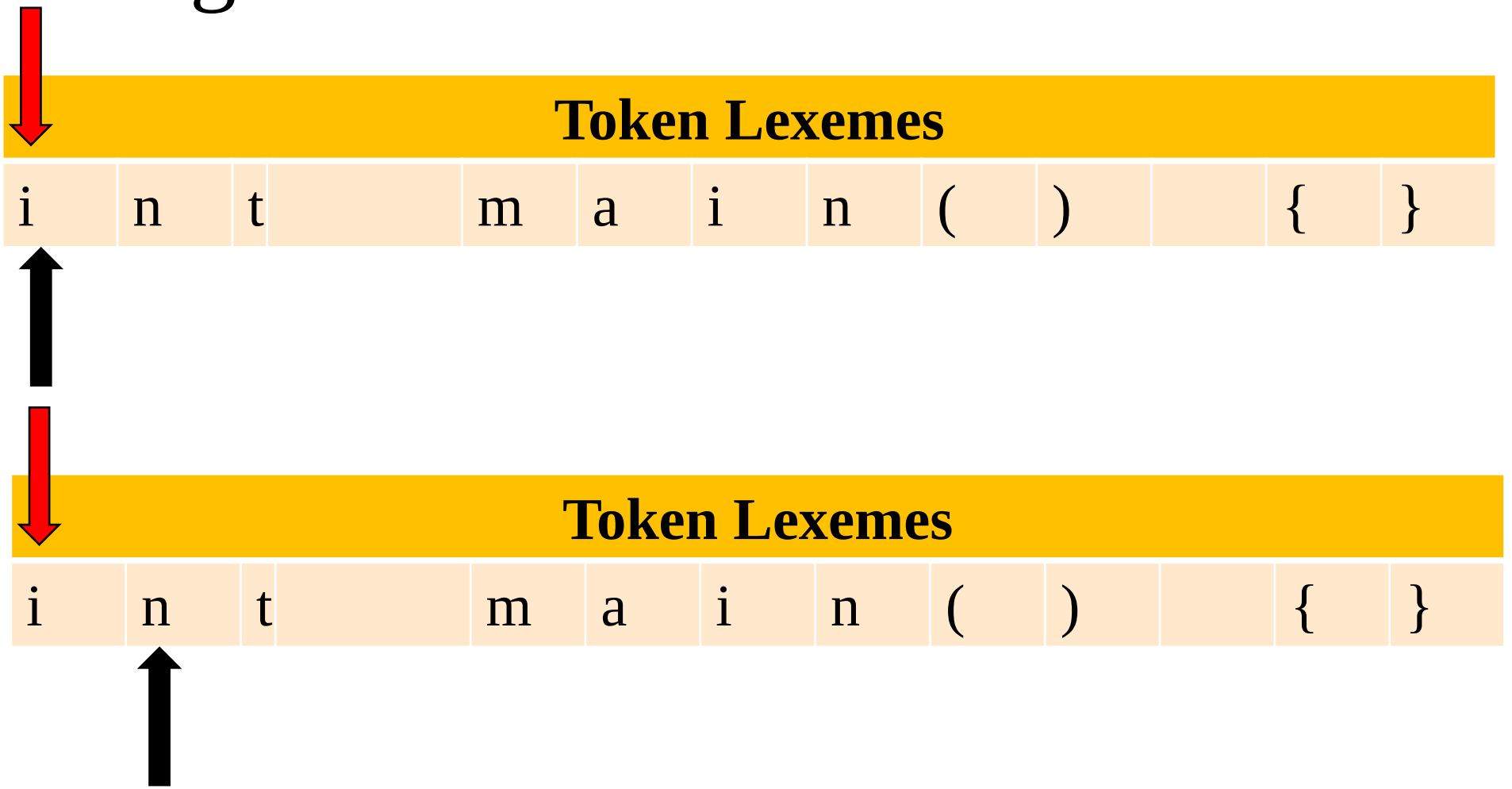
- Lexem Begin Pointer (red color pointer)

  Points to the beginning of every token

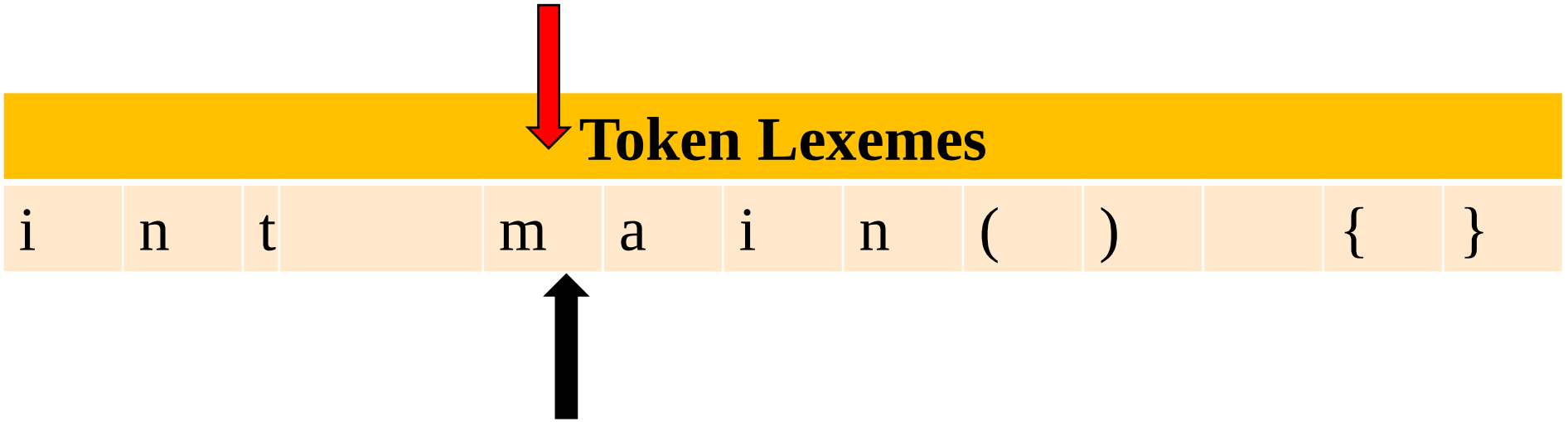
- Forward Pointer (Black color pointer)

  points every next character in the input string

Buffering



Buffering



Working of input buffering

- Two pointer should be used for buffering
- Two pointers “lexeme” and “fwd” to the input buffer are maintained.
- The string of characters enclosed between the two pointers is the current lexeme.
- To find the lexeme, first initialize both the pointers with the first character of the input.
- Keep incrementing the fwd pointer until a match for the pattern is found.
- Once a character not matching is found, stop incrementing the pointer and extract the string between the “lexeme” and “fwd” pointers.

Types of Input Buffering

- There are two types of input buffering

1. Buffer Pairs
2. Sentinel Value

Buffer Pairs

- Buffer is to divide the buffer into two halves. Means N character halves.

1st Half

2nd Half

Sentinel Value

- A sentinel is a special character, often used at the end of the input buffer, to mark its boundaries.
- Sentinels are essential for two primary reasons:
- 1. Boundary Detection
- 2. Safety Against Buffer Overflow

Sentinels Buffering

- Sentinels are essential for two primary reasons:
- 1. Boundary Detection: Sentinels provide a clear demarcation between the end of the input buffer and the start of the output buffer. This ensures that the compiler knows where to stop reading characters and where to begin forming lexemes.
- 2. Safety Against Buffer Overflow: Sentinels act as guards against buffer overflow. By signaling the end of the input buffer, they prevent the input buffer from overflowing into the output buffer, which could lead to data corruption and unpredictable behavior.

Lexical Analyzer Generator

- The lexical analysis process can automated, we only need to specify

1. Regular expression to tokens
2. Rule priorities for multiple longest match cases

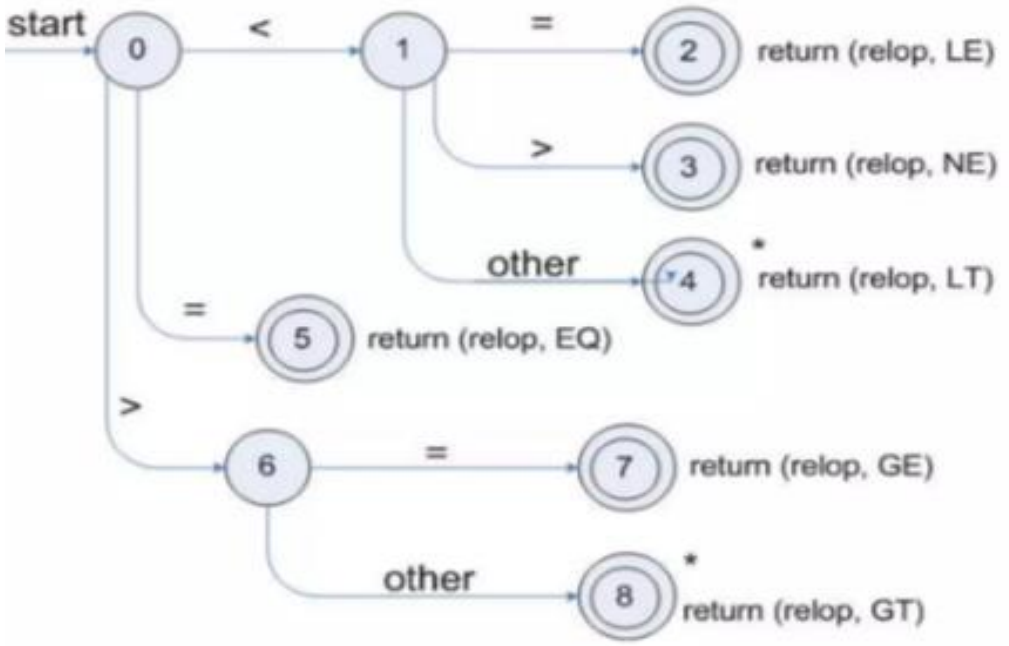
Flex is used to generates the lexical analyzer in C or C++

Recognition of tokens

- To understand the recognition of tokens, consider the regular expression of tokens together with the attributes values

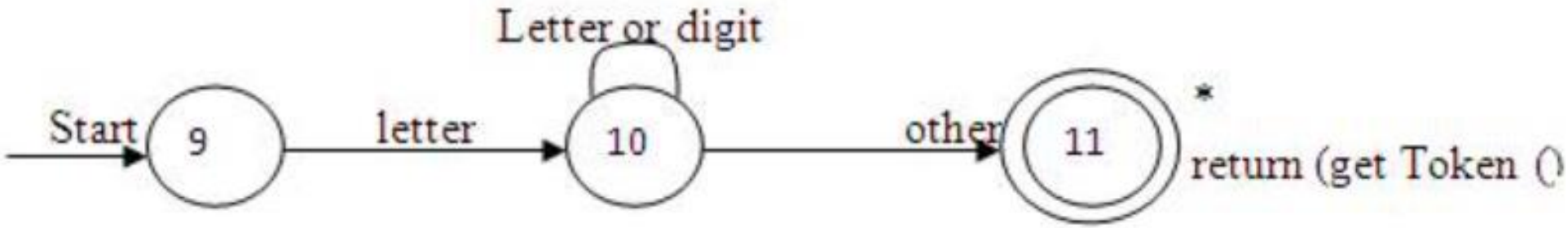
Lexemes	Token Names	Attribute Value
<	**Relop**	**LT**
<=	**Relop**	**LE**
=	**Relop**	**EQ**
<>	**Relop**	**NE**
>	**Relop**	**GT**
>=	**Relop**	**GE**

- Transition diagram for **relop**



Relop means relational operator

Transition Diagram for Identifiers



Introduction to Lex

- Lex- Definition Section

  contain definition and included code

- Definition is like macros and have the following form

Name	Translation
Digit	[0-9]
Number	{digit} {digit}*

- Included code is all code included %{ %}

Lex Program

- A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Definition

- The definition section contains the declaration of variables, regular definitions, manifest constants.
- In the definition section, text is enclosed in “%{ %}” brackets.
- Anything written in this brackets is copied directly to the file lex.yy.c
- Example:
- #include<stadio.h>
- #include<tokendef.h>

Rules

- The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets.
- The rule section is enclosed in **“%% %%”**

Rules

"void" {return(TOK_VOID);}

"int" {return(TOK_INT);}

"if"   {return(TOK_IF);}

"else" {return(TOK_ELSE);}

"while"{return(TOK_WHILE)};

"<="   {return(TOK_LE);}

Rules

">=" {return(TOK_GE);}

"==" {return(TOK_EQ);}

"!=" {return(TOK_NE);}

{D}+ {return(TOK_INT);}

{id} {return(TOK_ID);}

[\n]|[\t]|[ ] ;

%%

YYWRAP ( )

- yywrap( )
- Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex.

User Code Section

```
int main ( )
{
printf(“enter a string”);
yylex( );
}
```

Command to install Flex in Ubuntu

- Sudo apt-get update
- Sudo apt install flex
- Sudo apt install bison