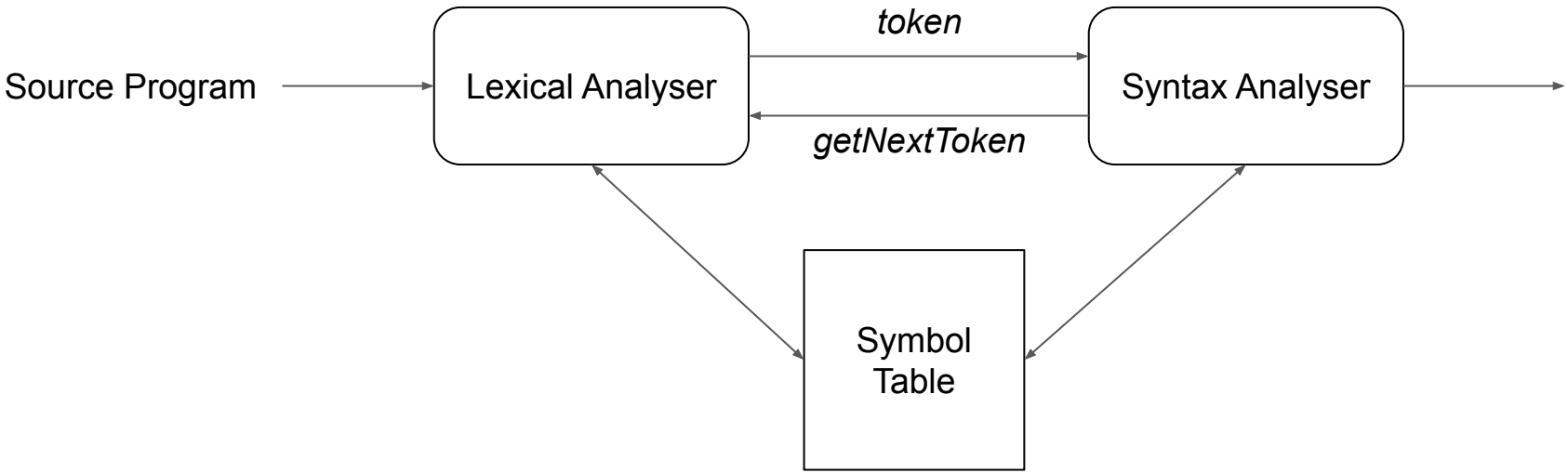
# Lexical Analysis

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

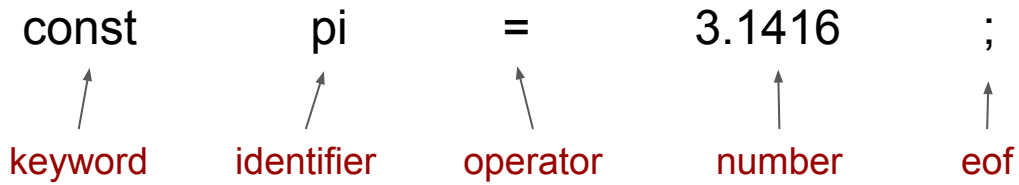
# Lexical Analysis

- Reads the source program character by character to produce tokens
- Doesn’t return a list of tokens at one shot
- Returns a token when the parser asks a token from it



# Token, Pattern and Lexeme

- Pattern:
  - A rule associated with each token.
  - E.g.: Regular Expression
- Token
  - Define a type of the matched pattern
  - E.g.: <number>
- Lexeme
  - A sequence of characters that is matched by the pattern for a token
  - E.g.: “3.1416”



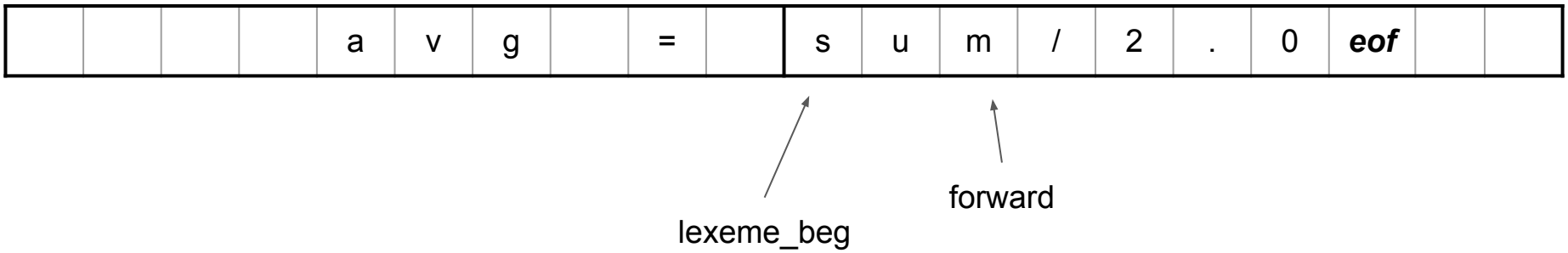
# Input Buffering

- Since lexical analyser deals with input (files), we need efficient techniques to read the input as well.

- Lookahead
  - There are times when the lexical analyzer needs to look ahead several characters beyond the lexerne for a pattern before a match can be announced,
  - Specialized buffering techniques are needed to reduce the amount of overhead required to process an input character.

# Input Buffering - Buffer pair

- Lets, we have two buffers that hold *N*-character each.
- If fewer than *N* characters remain in the input, then a special character ***eof*** is read into the buffer after the input characters
- Two pointers (*lexeme_beg* and *forward*) to the input buffer are maintained
  - The string of characters between the two pointers is the current lexeme



# Input Buffering - Buffer pair

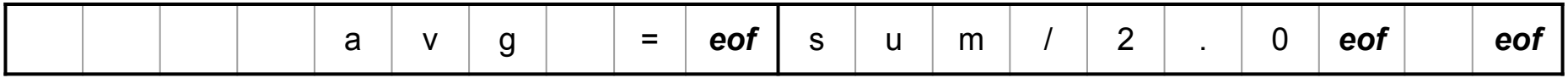
- If the forward pointer is about to move past the halfway mark, the right half is filled with *N* new input characters.
- If the forward pointer is about to move past the right end of the buffer, the left half is filled with *N* new characters and the forward pointer wraps around to the beginning of the buffer.
- What if the forward pointer has to travel more than the length of the buffer?

```
if forward at end of 1st half
    reload 2nd half;
    forward++;
elseif forward at end of 2nd half
    reload 1st half;
    forward = 0;
else
    forward++;
```

# Input Buffering - Sentinels

- We can reduce the two tests to one if we extend each buffer half to hold a sentinel character (e.g., eof) at the end.

```
forward++;
if forward = eof
    if forward at end of 1st half
        reload 2nd half;
        forward++;
    elseif forward at end of 2nd half
        reload 1st half;
        forward = 0;
    else   /*   eof   within   a   buffer
    signifying end of input */
        terminate lexical analysis
```



# Lexical Errors

- **Error Detection**
  - Lexical error occurs when the input is not accepted by any patterns
    - `E.g., 2add = 5`
  - `fi(true)` → Is this a lexical error?
    - No, lexical analysis cannot tell whether this is a misspelling of '*if*' or a valid identifier.
    - It will return identifier as token and let other layers deal with this error.
- **Error Handling**
  - Delete successive characters from the remaining input until the lexical analyzer can find a well-formed token → Panic mode
  - Inserting a missing character
  - Replacing an incorrect character by a correct character
  - Transposing two adjacent characters

# Pattern (RE) for token <number>

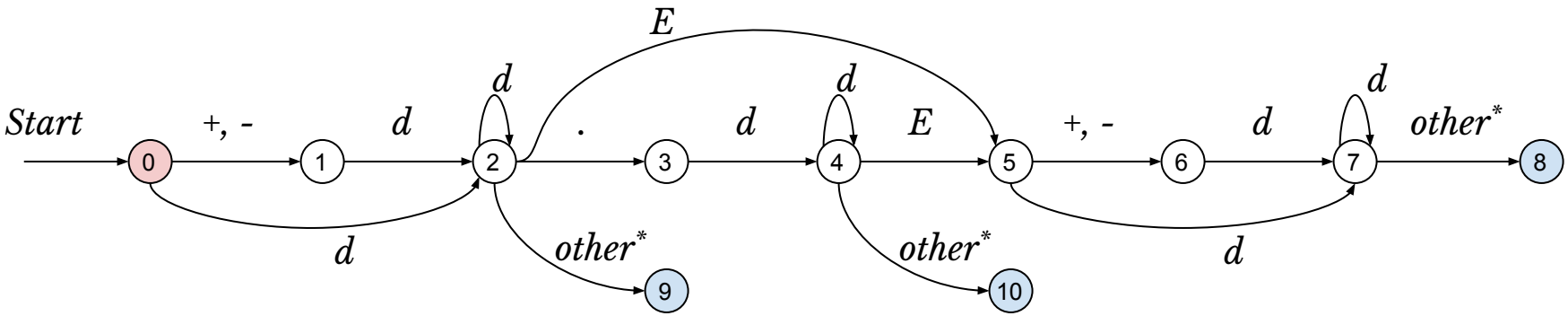
- digit → 0 | 1 | 2 | ..... | 9
- digits → $digit^{+}$
- fraction → (. digits)?
- exponent → (E (+ | -)? digits)?

- number → (+ | -)? digits fraction exponent
- number → (+ | -)? digits (. digits)? (E (+ | -)? digits)?

*d: digit*

# DFA for token <number>

number → (+ | -)? digits (. digits)? (E (+ | -)? digits)?




Check for following numbers

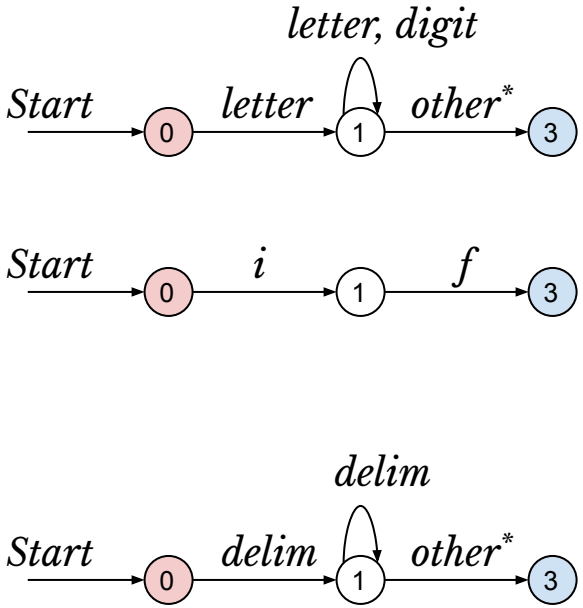
- 12, 12.3, 12.3E31, 12.3E-31, 12E31, 12E+31
- -12, -12.3 and so on.

***other*** → The lookahead symbol to find the boundary of a token.

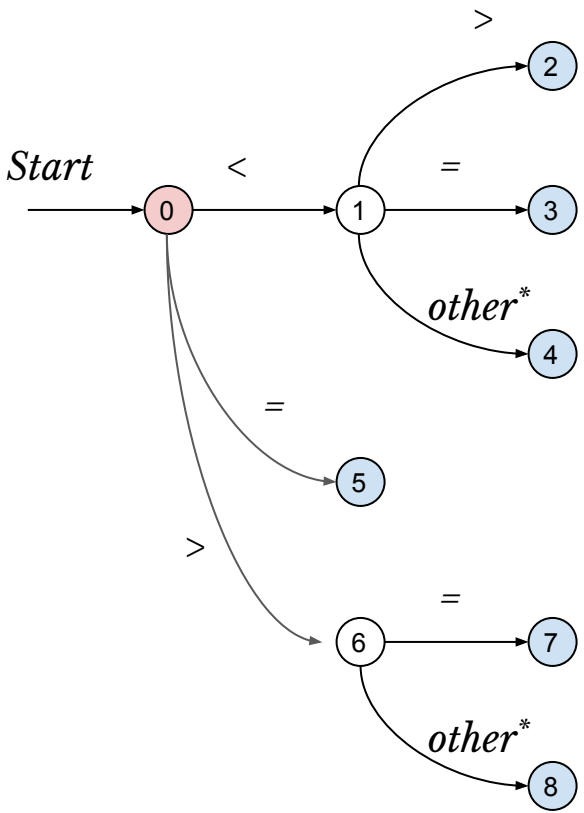
* → After reading the lookahead, backtrack to start looking for another token.

# Patterns (RE) and DFA for different tokens

- id → letter (letter | digit)*
- Keywords
  - if → i f
  - then → t h e n
- Whitespaces
  - delim→ blank | tab | newline
  - ws → delim$^{+}$
- relop → < | <= | > | >= | = | <>



# DFA for token <relop>



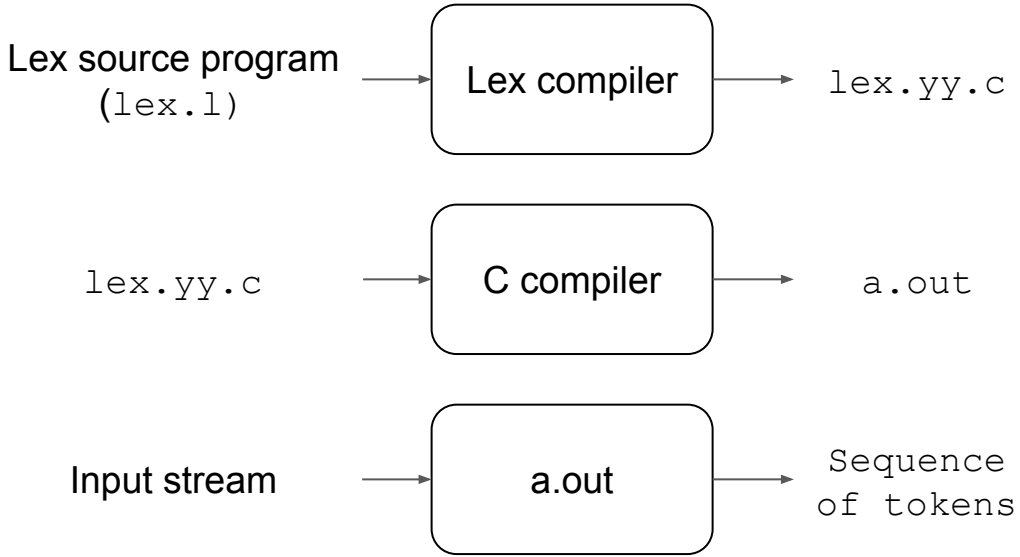
# Building a Lexical Analyser

- Implement all these DFA/REs into some programming language, e.g., C, Java.
  - Read the input
  - Find the pattern
  - Update symbol table
  - Detect/Handle errors

- Use a tool that does these for you.
  - E.g., Lex compiler or simple Lex

# Lex compiler or Lex

- Write a lex program in lex language and save it in some file (e.g., `lex.l`)
- Compile it with lex compiler. It will generate a C code (e.g., `lex.yy.c`)
- Compile the generated C code with C compiler. It will generate an object code `a.out`
- Execute the object code `a.out` on the input stream



# Lex specifications

- A lex program consists of three parts
  - Declaration
    - Declarations of variables, regular definitions, etc.
  - Transition rules
    - Patterns and actions (C code)
  - Auxiliary functions
    - Some helper functions (C code)

```
Declaration
%%
Transition rules
%%
Auxiliary functions
```

**A typical lex file: `lex.l`**

# A sample lex program - Declaration

```
%{
/* definitions of manifest constants
LT, LE, EQ, NE* GT, GEB
IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim  [ \t \n]
ws     {delim}+
letter [A-Za-z]
digit      [0-9]
id     {letter}({letter} | {digit})*
number {digit}+ {\.{digit)+}? {E[+|-]? {digit}+}?
```

# A sample lex program - Transition rules

```
%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = install_id(); return(ID};}
{number}    {yylval = install_num(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}
```

# A sample lex program - Auxiliary functions

```
%%
install_id()
{
    /* procedure to install the lexeme, whose first character
    is pointed to by yytext and whose length is yyleng, into
    the symbol table and return a pointer thereto */
}
install_num()
{
    /* similar procedure to install a lexeme that is a number
    */
}
```