Lexical Analysis
Part-1

Lexical Analysis

- Basic Concepts & Regular Expressions
  - What does a Lexical Analyzer do?
  - How does it Work?
  - Formalizing Token Definition & Recognition
- Reviewing Finite Automata Concepts
  - Non-Deterministic and Deterministic FA
  - Conversion Process
    - Regular Expressions to NFA
    - NFA to DFA
- Relating NFAs/DFAs /Conversion to Lexical Analysis

Lexical Analysis

- The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++ and gives row number and column number of the error.

- Suppose we pass a statement through lexical analyzer –

  a = b + c ;

- It will generate token sequence like this:

  id1=id2+id3;

- Where each id refers to it's variable in the symbol table referencing all details

Lexical Analysis

Sample Program in C:

```
int main()
{
  // 2 variables
   int a, b;
   a = 10;

   return 0;
 }
```

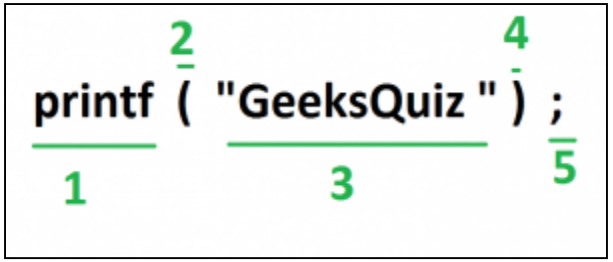
Note: You can observe that we have omitted comments.

All the valid 18 tokens are:

'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';'
'a' '=' '10' ';' 'return' '0' ';' '}'

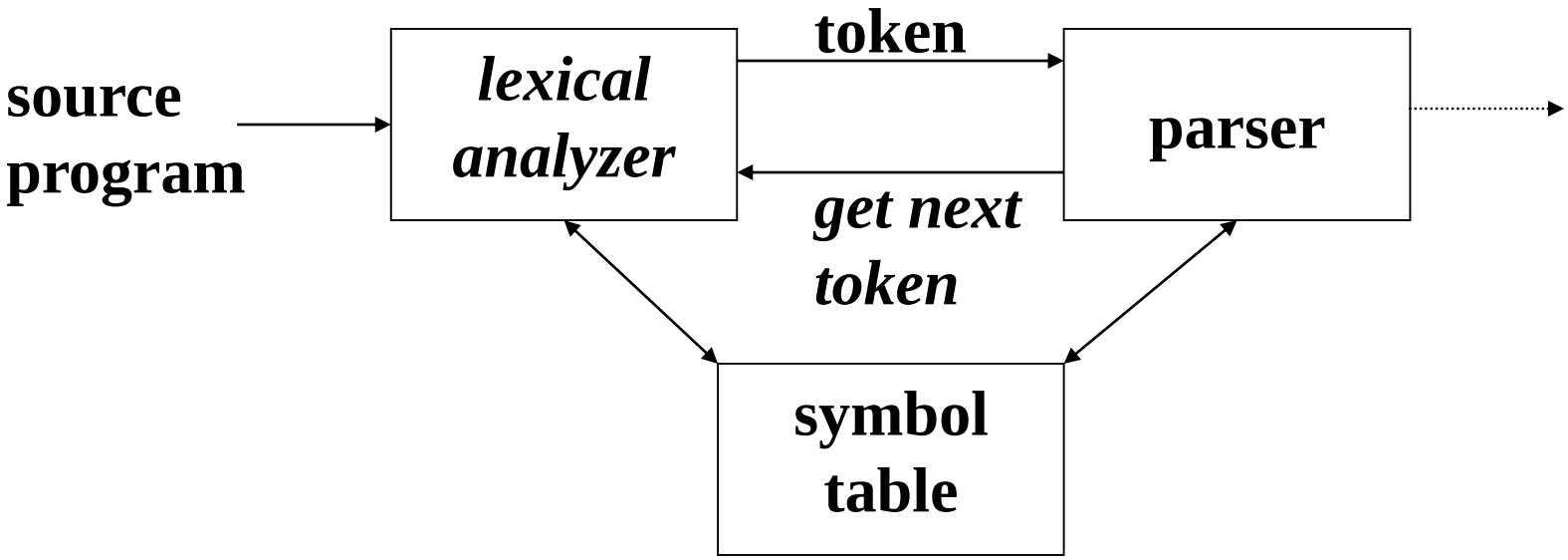
Lexical Analysis

As another example, consider below printf statement in C:



There are 5 valid token in this printf statement.

Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser.

Lexical Analyzer in Perspective

- Identify the words: Lexical Analysis
  - Converts a stream of characters (input program) into a stream of tokens.
    - ✓ Also called Scanning or Tokenizing

What is a token?

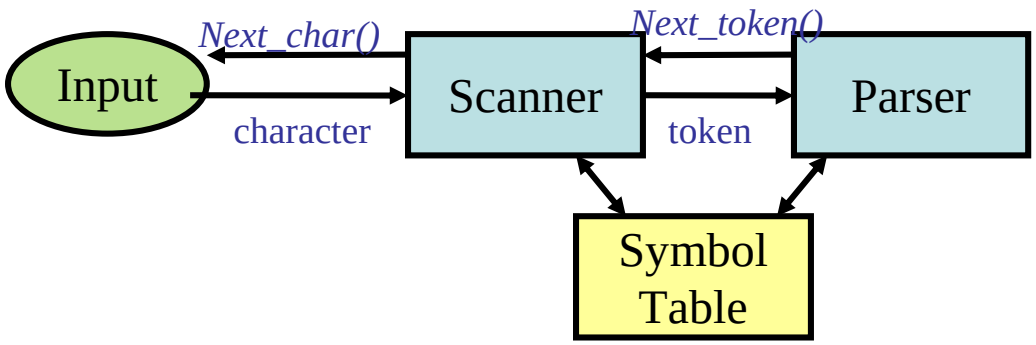
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- ✓ Keywords: Examples-for, while, if etc.
- ✓ Identifier: Examples-Variable name, function name, etc.
- ✓ Operators: Examples '+', '++', '-' etc.
- ✓ Separators: Examples ',' ';' etc

- Identify the sentences: Parsing.
  - Derive the structure of sentences: construct parse trees from a stream of tokens.

Interaction of Lexical Analyzer with Parser



- Often a subroutine of the parser
- Secondary tasks of Lexical Analyzer
  - Strip out comments and white spaces from the source
  - Correlate error messages with the source program
  - Preprocessing may be implemented as lexical analysis takes place

Lexical Analyzer vs. Parser

- **LEXICAL ANALYZER**
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Generate Errors
  - Send Tokens to Parser
- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Rep. of Source
  - Generate Errors
  - And More…. (We'll see later)

What Factors Have Influenced the Functional Division of Labor ?

- Separation of Lexical Analysis From Parsing Presents a Simpler Conceptual Model
  - A parser embodying the conventions for comments and white space is significantly more complex that one that can assume comments and white space have already been removed by lexical analyzer.
- Separation Increases Compiler Efficiency
  - Specialized buffering techniques for reading input characters and processing tokens…
- Separation Promotes Portability.
  - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

Introducing Basic Terminology

- What are Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - Recall File and OS Wildcards ([A-Z]*.*)
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

Introducing Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	characters i, f
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

TOKEN STREAM

- Tokens are terminal symbol in the grammar for the source language
  - **keywords, operators, identifiers, constants, literal strings, punctuation symbols etc.**
- Source:

  if (x == -3.1415) /* test x */ then ...
- Token Stream:

  < IF >
  < LPAREN >
  < ID, "x" >
  < EQUALS >
  < NUM, -3.14150000 >
  < RPAREN >
  < THEN >
  ...

Attributes for Tokens

Tokens influence parsing decision; the attributes influence the translation of tokens.

- **A token usually has only a single attribute**
  - **A pointer to the symbol-table entry**
  - **Other attributes (e.g. line number, lexeme) can be stored in symbol table**

Attributes for Tokens

Tokens influence parsing decision; the attributes influence the translation of 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,** >

<**num,** integer value 2>

Handling LEXICAL ERROR

- Error Handling is very localized, with Respect to Input Source
- Few errors can be caught by the lexical analyzer
  - Most errors tend to be "typos"
  - Not noticed by the programmer

    -- retunn 1,23;
  - ... Still results in sequence of legal tokens

    -- **<ID, "retunn"> <INT,1> <COMMA> <INT,23> <SEMICOLON>**
  - No lexical error, but problems during parsing!
  - Another example: ***fi (a == f(x))***
- In what Situations do Errors Occur?
  - Lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of remaining input.

Recovery from lexical errors

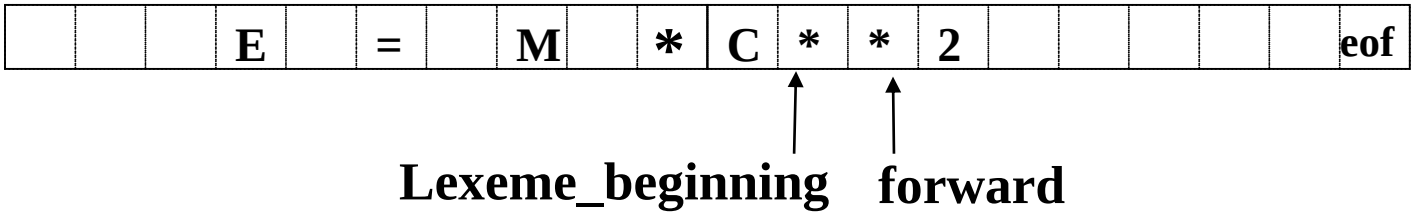
- Panic mode recovery
  - Delete successive characters from the input until the lexical analyzer can find a well-formed token
  - May confuse the parser
  - The parser will detect syntax errors and get straightened out (hopefully!)
- Other possible error-recovery actions
  - Deleting an extra character
  - Inserting a missing character
  - Replacing an incorrect character by a correct character
  - Swapping two adjacent character

Managing Input Buffers

- Option 1: Read one char from OS at a time.
- Option 2: Read N characters (buffer size) per system call
  - e.g., N = 4096
- Manage input buffers in Lexical Analyzer
  - More efficient
- The input character is thus read from secondary storage, but reading in this way from secondary storage is costly; hence buffering technique is used.
- A block of data is first read into a buffer, and then second by lexical analyzer.
- Thus, Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced
- But! Due to look ahead we need to push back the lookahead characters
  - Need specialized buffer managing technique to improve efficiency

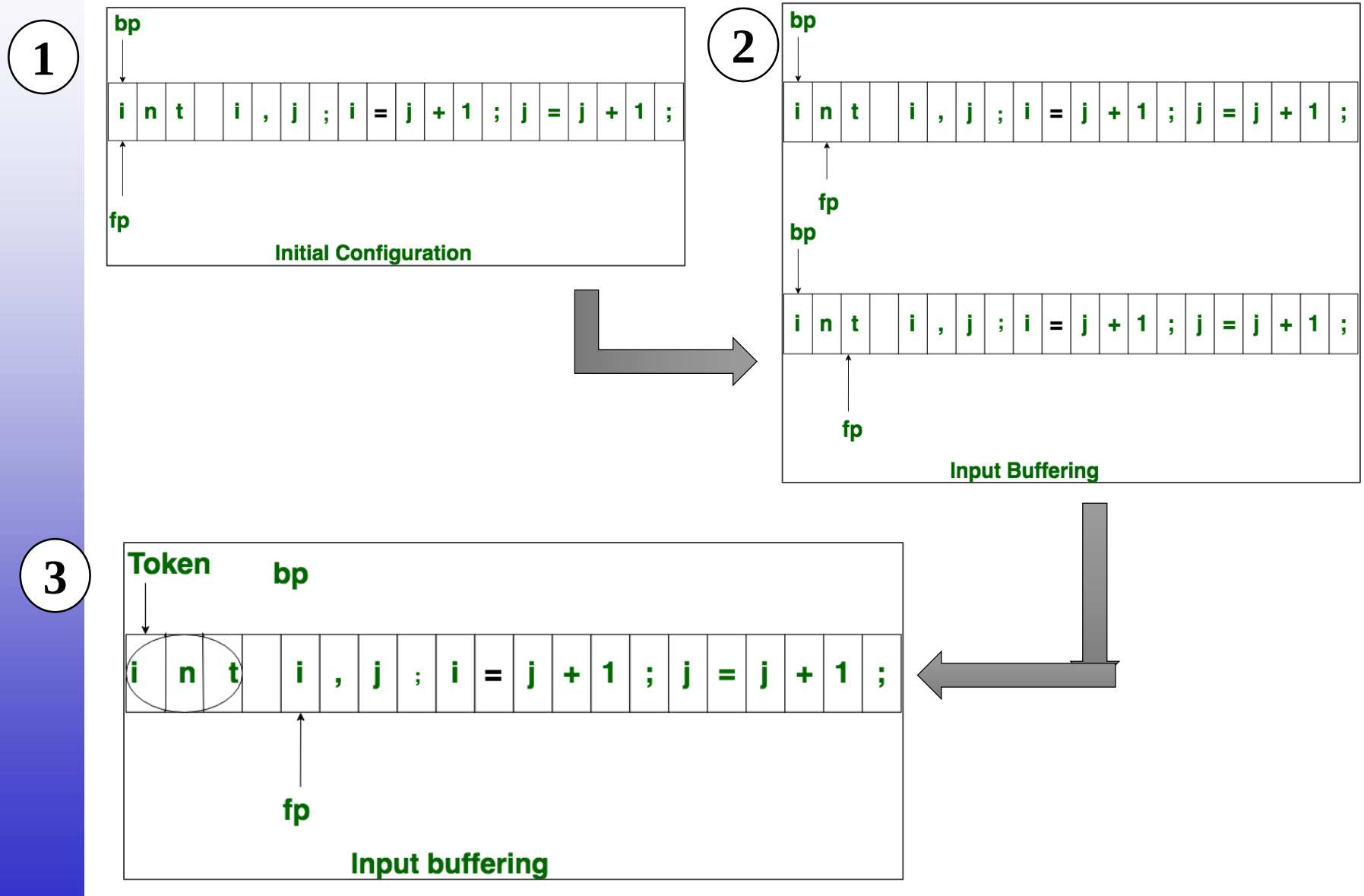
Special Buffering Technique

- The lexical analyzer scans the input from left to right one character at a time. Two pointers **lexeme_beginning(bp)** and **forward(fp)** to the input buffer are maintained.
- The string of characters between the pointers is the current lexeme.
- Initially both pointers point to first character of the next lexeme to be found. Forward pointer scans ahead until a match for a pattern is found
- Once the next lexeme is determined, the forward pointer is set to the character at its right end.
- After the lexeme is processed both pointers are set to the character immediately past the lexeme



- Comments and white space can be treated as patterns that yield no token

MANAGING INPUT BUFFERS



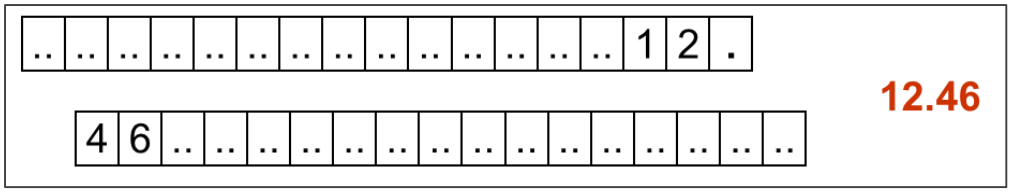
MANAGING INPUT BUFFERS

There are two methods used in this context:

- One Buffer Scheme
- Two Buffer/ Buffer Pair Scheme

Limitations of One Buffer Scheme:

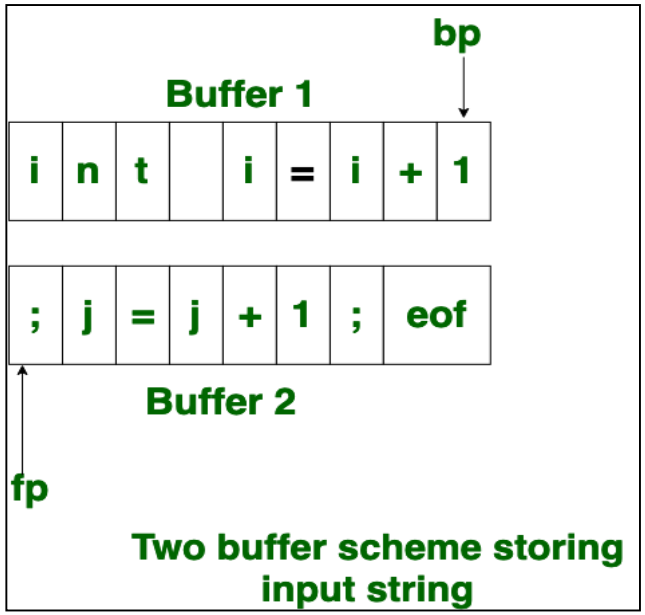
In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



MANAGING INPUT BUFFERS

Two Buffer/Buffer Pair Scheme:

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled.



Code to advance forward pointer

```
if  forward at the end of first half then  begin
        reload second half ;
        forward  : =  forward  + 1;
end
else if  forward  at end of second half  then begin
         reload first half ;
         move  forward  to biginning of first half
end
else  forward  : =  forward  + 1;
```

Pitfalls:

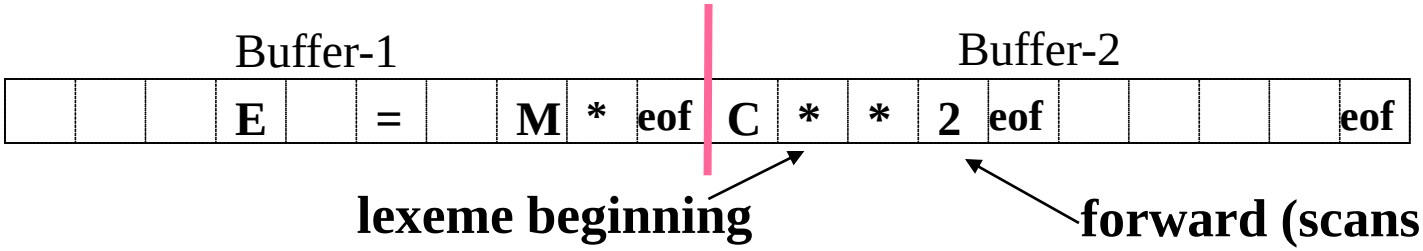
1. This buffering scheme works quite well most of the time but with it amount of lookahead is limited.
2. Limited lookahead makes it impossible to recognize tokens in situations where the distance, forward pointer must travel is more than the length of buffer.

Problem?

Except at the end of buffer halves, for each advance of *forward* requires 2 tests

```
if  forward at the end of first half then  begin
        reload second half ;
        forward  : =  forward  + 1;
end
else if  forward  at end of second half  then begin
         reload first half ;
         move  forward  to beginning of first half
end
else  forward  : =  forward  + 1;
```

Algorithm: Buffered I/O with Sentinels




```
forward : = forward + 1 ;
if forward ↑ = eof then begin
   if forward at end of first half then begin
        reload second half ;
        forward : = forward + 1
   end
   else if forward at end of second half then begin
        reload first half ;
        move forward to beginning of first half
   end
   else / * eof within buffer signifying end of input * /
        terminate lexical analysis
end
```

We extend each buffer half to hold a sentinal [*eof*] character at the end

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

Given Tokens, What are Patterns ?

if → **if**

then → **then**

else → **else**

relop → < | <= | > | >= | = | <>

id → **letter (letter | digit)***

num → **digit** $^{+}$ **(. digit** $^{+}$ **) ? (E(+ | -) ? digit** $^{+}$ **) ?**

Grammar:

stmt → |**if *expr* then *stmt***
|**if *expr* then *stmt* else *stmt***
|∈

expr → ***term* relop *term* | *term***

term → **id | num**

What Else Does Lexical Analyzer Do?

Scan away *blanks*, new lines, tabs

Can we Define Tokens For These?

blank → blank

tab → tab

newline → newline

delim → blank | tab | newline

ws → delim $^{+}$

In these cases no token is returned to parser

Overall

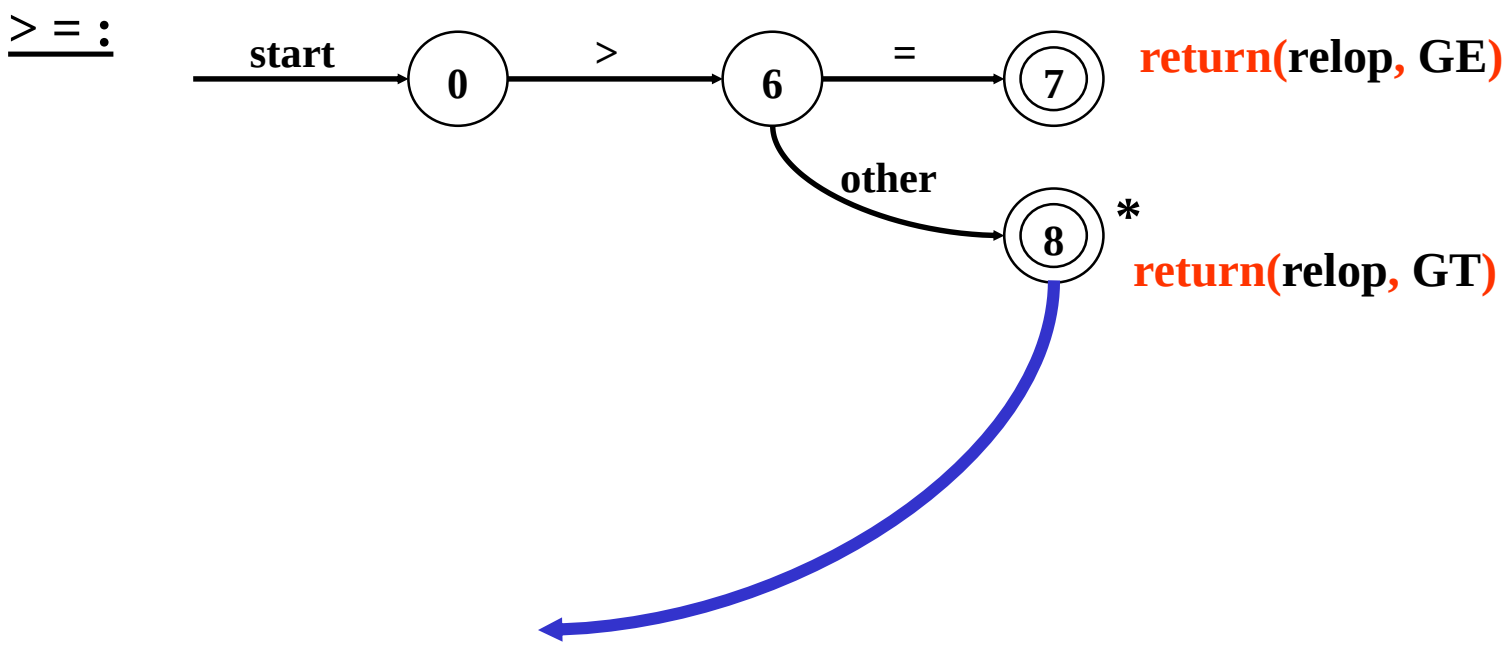
Regular Expression	Token	Attribute-Value
ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
id	**id**	**pointer to table entry**
num	**num**	**Exact value**
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
< >	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Note: Each token has a unique token identifier to define category of lexemes

Constructing Transition Diagrams for Tokens

- **Transition Diagrams (TD) are used to represent the tokens**
- **As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern**
- **Each TD has:**
  - **States : Represented by Circles**
  - **Actions : Represented by Arrows between states**
  - **Start State : Beginning of a pattern (Arrowhead)**
  - **Final State(s) : End of pattern (Concentric Circles)**
- **Each TD is Deterministic (assume) - No need to choose between 2 different actions !**

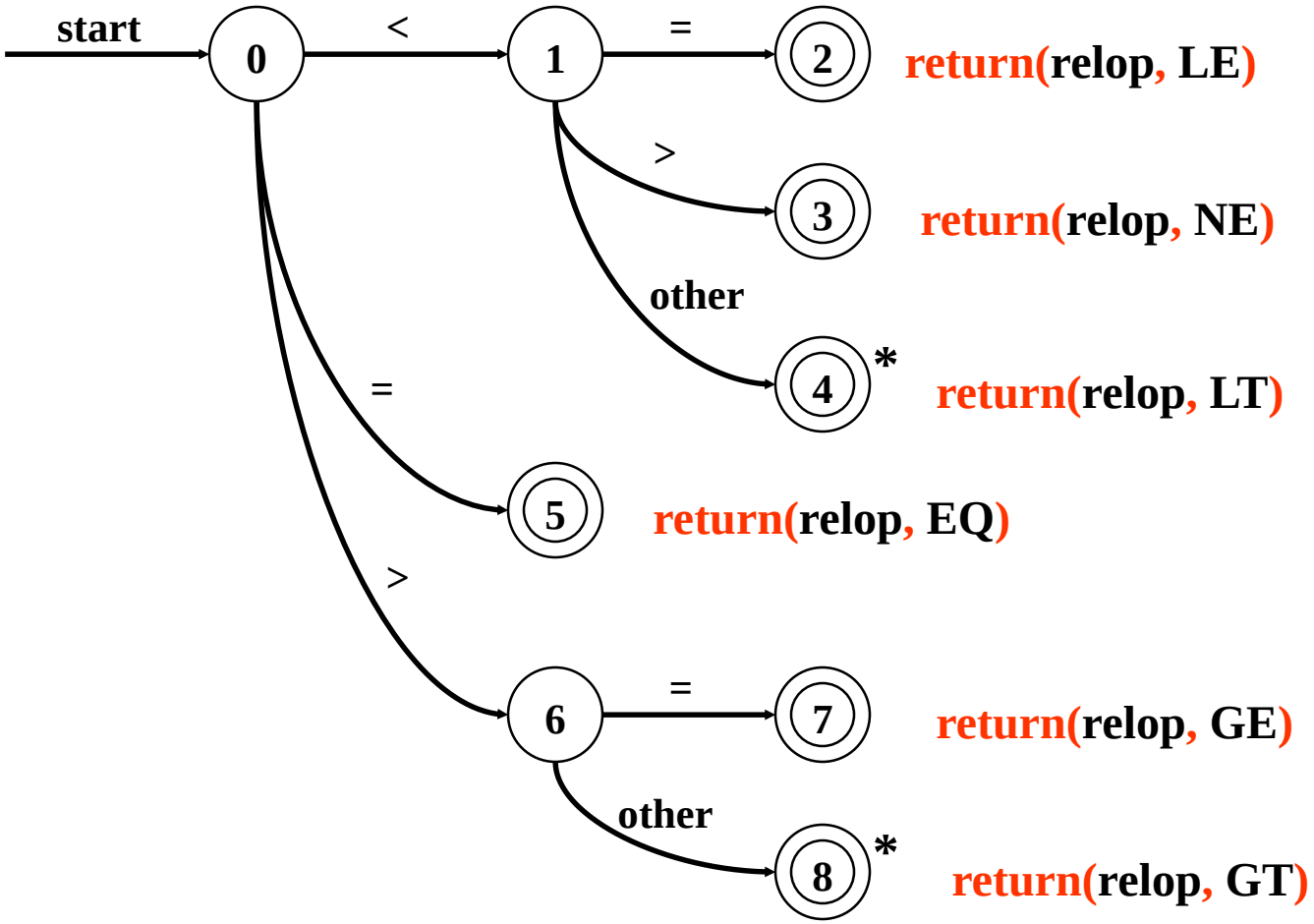
Example TDs



We've accepted ">" and have read one extra char that must be unread.

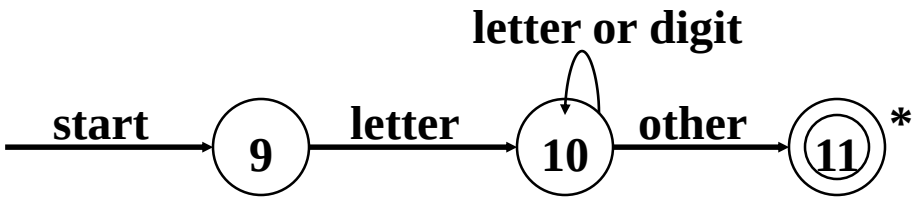
Note: The state 8 has a * to indicate that we must retract the input one position.

Example : All RELOPs



Example TDs : Identifier and Delimiter

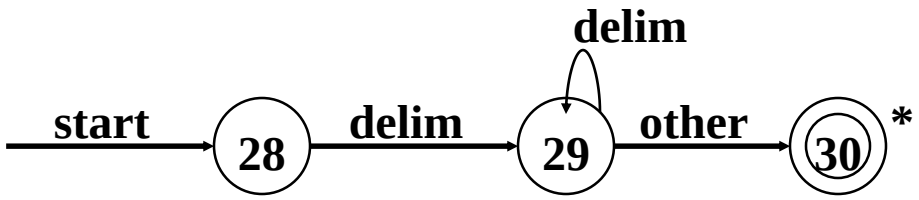
id :



return(get_token(), install_id())

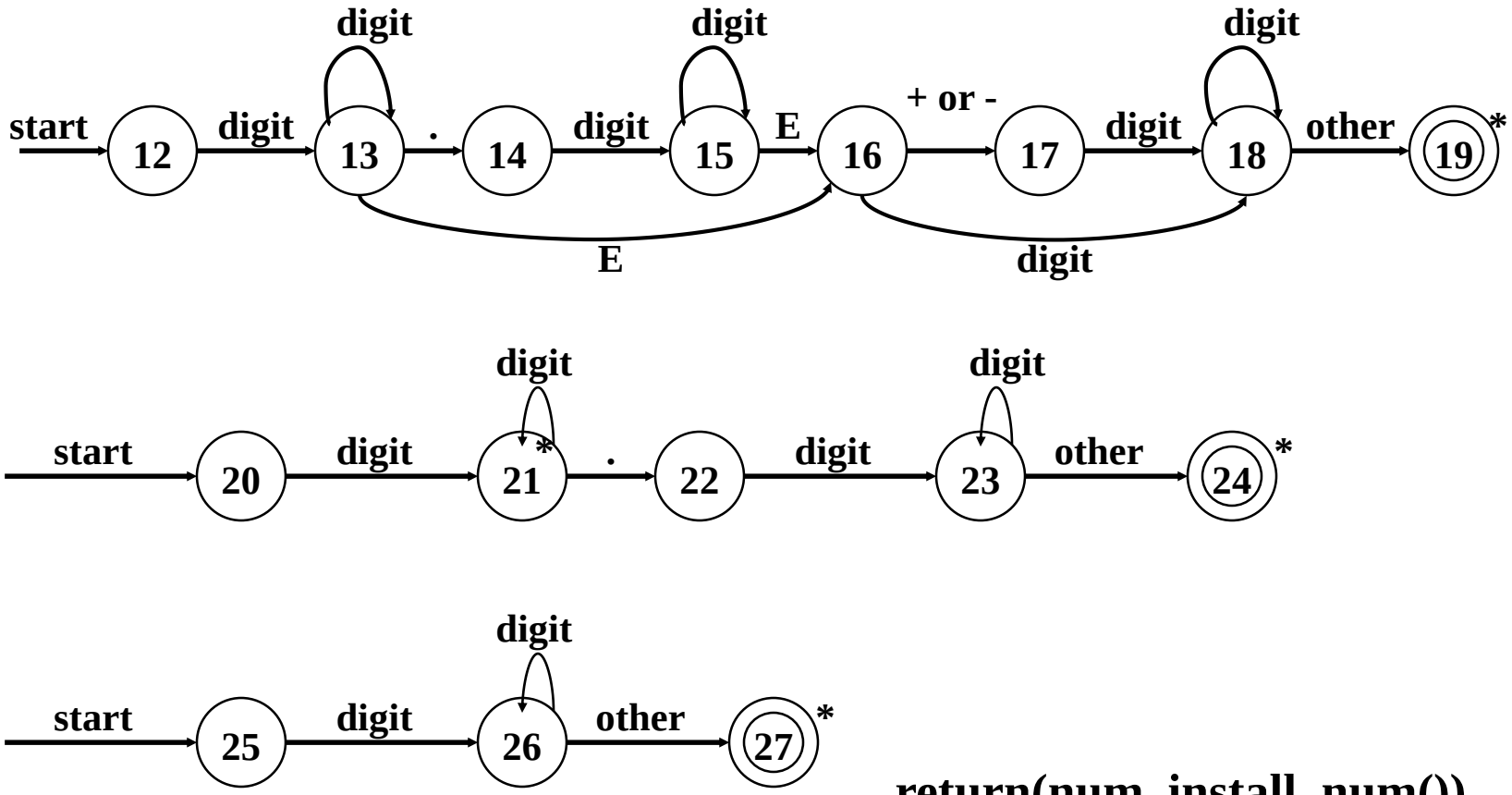
Either returns ptr or “0” if reserved

delim :



Example TDs : Unsigned Numbers

1240, 39.45, 6.33E15, or 1.578E-41

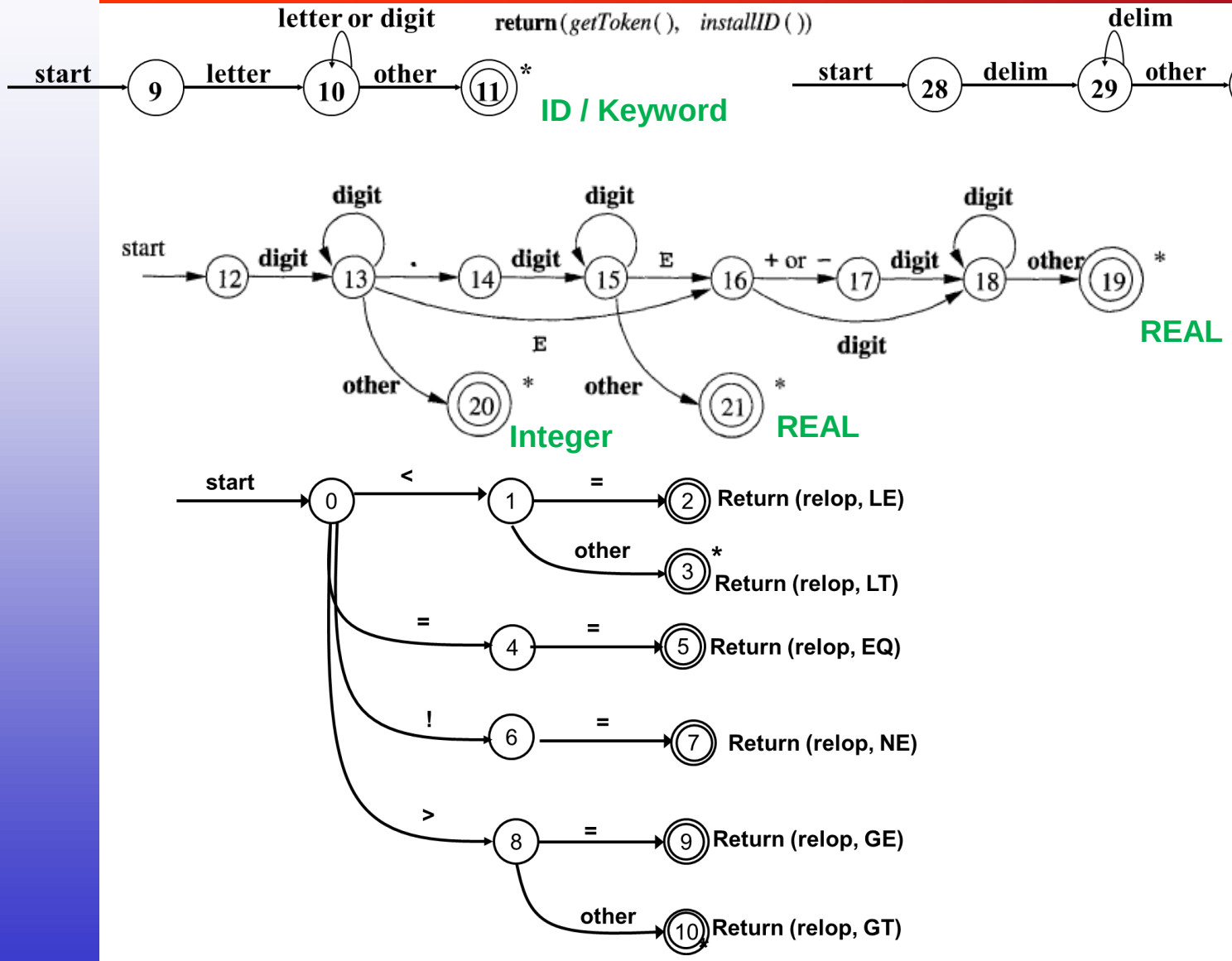


return(num, install_num())

Questions: Is ordering important for unsigned number ?

Why are there no TDs for `then, else, if` ?

Capturing Multiple Tokens



What Else Does Lexical Analyzer Do?

All Keywords / Reserved words are matched as ids

- **After the match, the symbol table or a special keyword table is consulted**
- **Keyword table contains string versions of all keywords and associated token values**

if	**15**
then	**16**
begin	**17**
...	**...**

- **When a match is found, the token is returned, along with its symbolic value, i.e., "then", 16**
- **If a match is not found, then it is assumed that an id has been discovered**

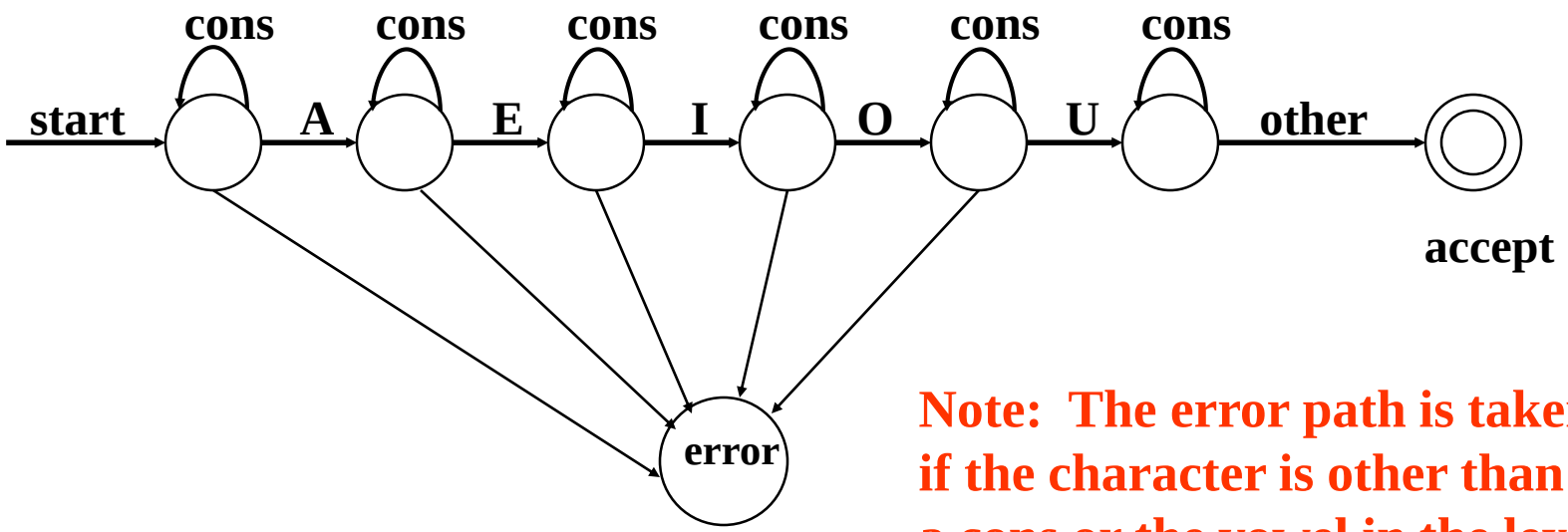
QUESTION :

What would the transition diagram (TD) for strings containing each vowel, in their strict lexicographical order, look like?

Answer

cons → B | C | D | F | G | H | J | … | N | P | … | T | V | .. | Z

string → cons* A cons* E cons* I cons* O cons* U cons*



Note: The error path is taken if the character is other than a cons or the vowel in the lex order.

Implementing a Transition Diagram

- Systematic approach for all transition diagrams
  - Program size $\propto$ number of states and edges
- We try each diagram and when we fail then we go to try the next diagram

Implementing Transition Diagrams

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the grammar

Each state gets a segment of code

FUNCTIONS USED

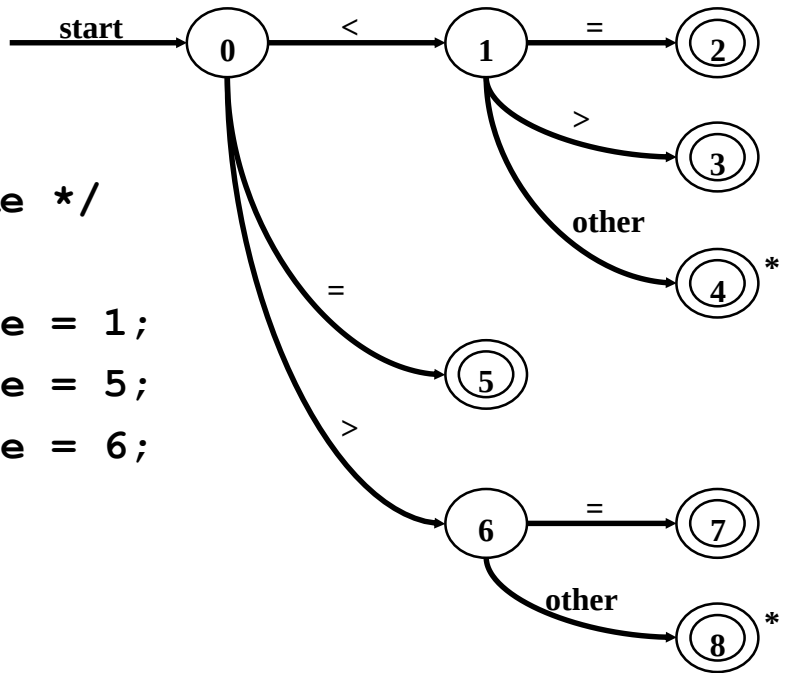
- nextchar(),
- retract(),
- install_num(),
- install_id(),
- gettoken(),
- isdigit(),
- isletter(),
- recover()

Implementing Transition Diagrams

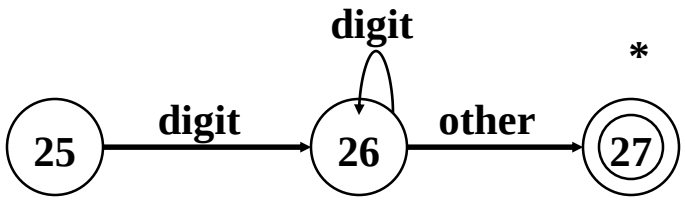
```
int state = 0, start = 0

lexeme_beginning = forward;
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
           /* c is lookahead character */
           if (c== blank || c==tab || c== newline) {
              state = 0;
              lexeme_beginning++;
                 /* advance
                   beginning of lexeme */
           }
           else if (c == '<') state = 1;
           else if (c == '=') state = 5;
           else if (c == '>') state = 6;
           else state = fail();
           break;
           … /* cases 1-8 here */
```

repeat until a "return" occurs



Implementing Transition Diagrams, II




```
.............
case 25;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = fail();
          break;
case 26;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = 27;
          break;
case 27;  retract(1); lexical_value = install_num();
          return ( NUM );
.............
```

advances forward

Case numbers correspond to transition diagram states !

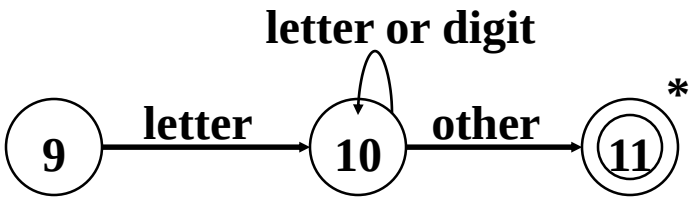
retracts forward

looks at the region lexeme_beginning ... forward

Implementing Transition Diagrams, III

```
.............
 case 9:   c = nextchar();
           if (isletter(c)) state = 10;
           else state = fail();
           break;
 case 10;  c = nextchar();
           if (isletter(c)) state = 10;
           else if (isdigit(c)) state = 10;
           else state = 11;
           break;
 case 11;  retract(1); lexical_value = install_id();
           return ( gettoken(lexical_value) );
.............
```

reads token name from ST



When Failures Occur:

```
int fail()
{
      forward = lexeme beginning;
      switch (start) {
         case 0:   start = 9;  break;
         case 9:   start = 12; break;
         case 12:  start = 20; break;
         case 20:  start = 25; break;
         case 25:  recover();  break;
         default:   /* lex error */
      }
      return start;
}
```

Switch to next transition diagram

Reading Materials

- Chapter -3 of your Text book:
  - Compilers: Principles, Techniques, and Tools
- https://www.geeksforgeeks.org/compiler-design-tutorials/

End of slide