Lexical Analysis

- Recognize tokens and ignore white spaces, comments

i	f		(	x	1		*	x	2	<	1	.	0	)	{

Generates token stream

if	(	x1	*	x2	<	1.0	)	{

- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata

Lexical Analysis

- Sentences consist of string of tokens (a syntactic category)

  For example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme

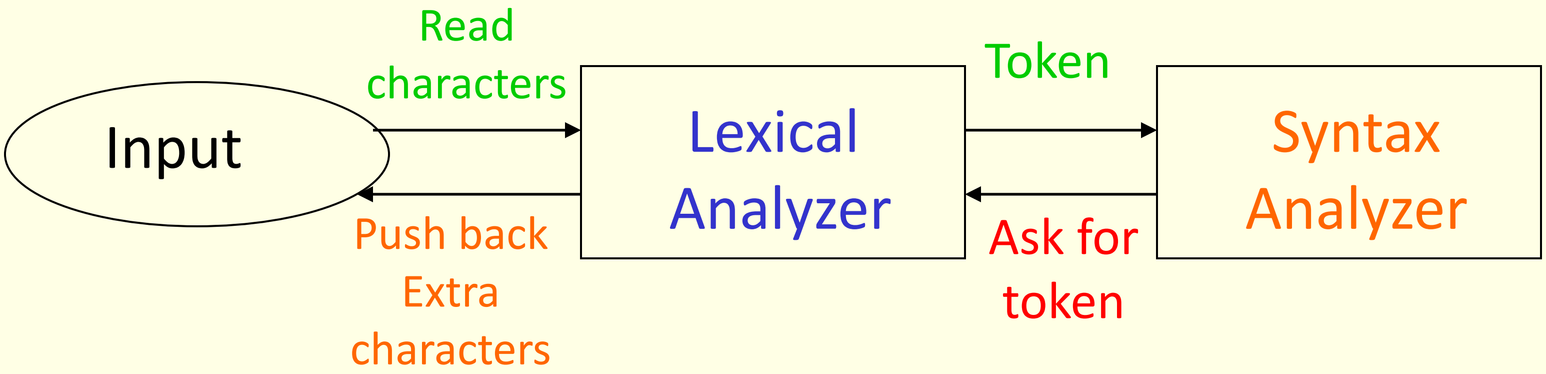
  for example, 100.01, counter, const, “How are you?”
- Rule of description is a pattern

  for example, letter (letter | digit)*
- Task: Identify Tokens and corresponding Lexemes

Lexical Analysis

- Examples
- Construct constants: for example, convert a number to token num and pass the value as its attribute,
  - 31 becomes <num, 31>
- Recognize keyword and identifiers
  - counter = counter + increment becomes id = id + id
  - check that id here is not a keyword
- Discard whatever does not contribute to parsing
  - white spaces (blanks, tabs, newlines) and comments

Interface to other phases



- Why do we need Push back?
- Required due to look-ahead
  for example, to recognize >= and >
- Typically implemented through a buffer
  - Keep input in a buffer
  - Move pointers over the input

Approaches to implementation

- Use assembly language

  Most efficient but most difficult to implement

- Use high level languages like C

  Efficient but difficult to implement

- Use tools like lex, flex

  Easy to implement but not as efficient as the first two cases

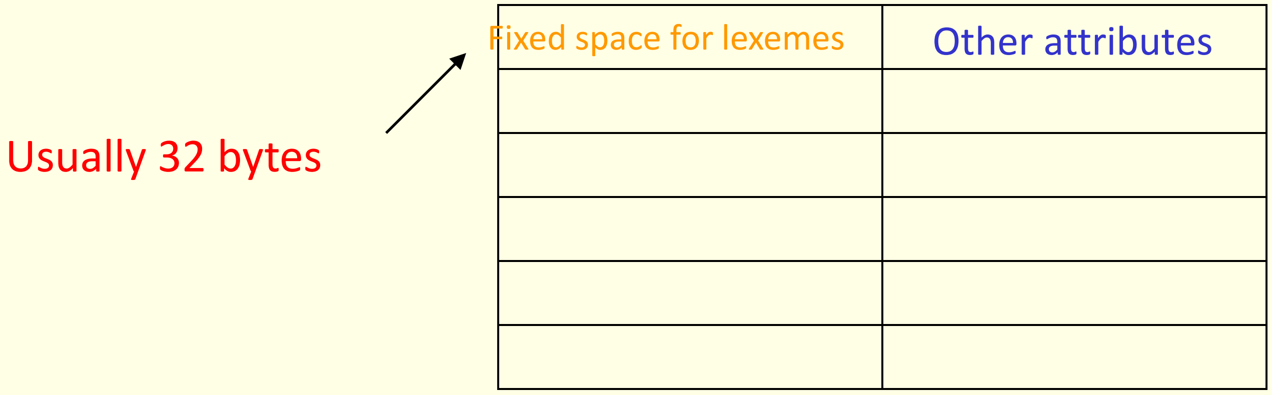
Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - Insert(s,t): save lexeme s and token t and return pointer
  - Lookup(s): return index of entry for lexeme s or 0 if s is not found

Implementation of Symbol Table

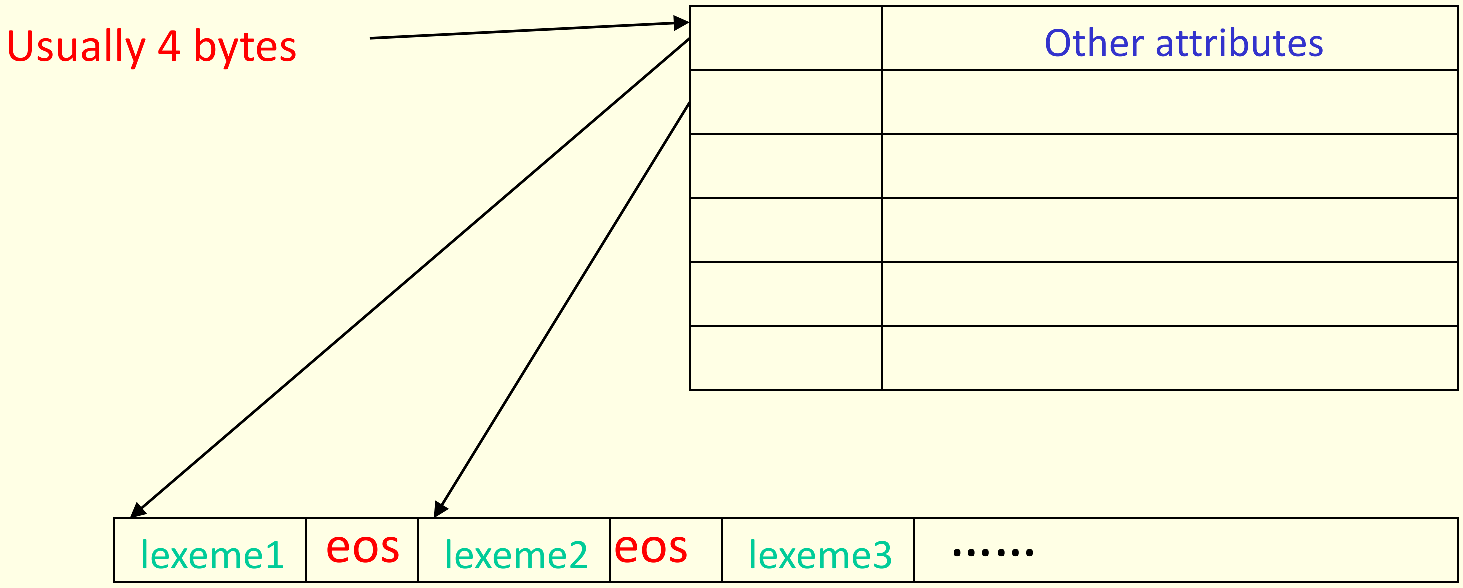
- Fixed amount of space to store lexemes.
  - Not advisable as it waste space.

- Store lexemes in a separate array.
  - Each lexeme is separated by eos.
  - Symbol table has pointers to lexemes.



Usually 32 bytes

Fixed space for lexemes	Other attributes



Usually 4 bytes

	Other attributes

lexeme1	eos	lexeme2	eos	lexeme3	……

How to handle keywords?

- Consider token DIV and MOD with lexemes div and mod.
- Initialize symbol table with insert(“div” , DIV) and insert(“mod” , MOD).
- Any subsequent insert fails (unguarded insert)
- Any subsequent lookup returns the *keyword* value, therefore, these cannot be used as an identifier.

Difficulties in the design of lexical analyzers

Is it as simple as it sounds?

Lexical analyzer: Challenges

- Lexemes in a fixed position. Fixed format vs. free format languages
- FORTRAN Fixed Format
  - 80 columns per line
  - Column 1-5 for the statement number/label column
  - Column 6 for continuation mark (?)
  - Column 7-72 for the program statements
  - Column 73-80 Ignored (Used for other purpose)
  - Letter C in Column 1 meant the current line is a comment

Lexical analyzer: Challenges

- Handling of blanks
  - in C, blanks separate identifiers
  - in FORTRAN, blanks are important *only* in literal strings
  - variable counter is same as count er
  - Another example

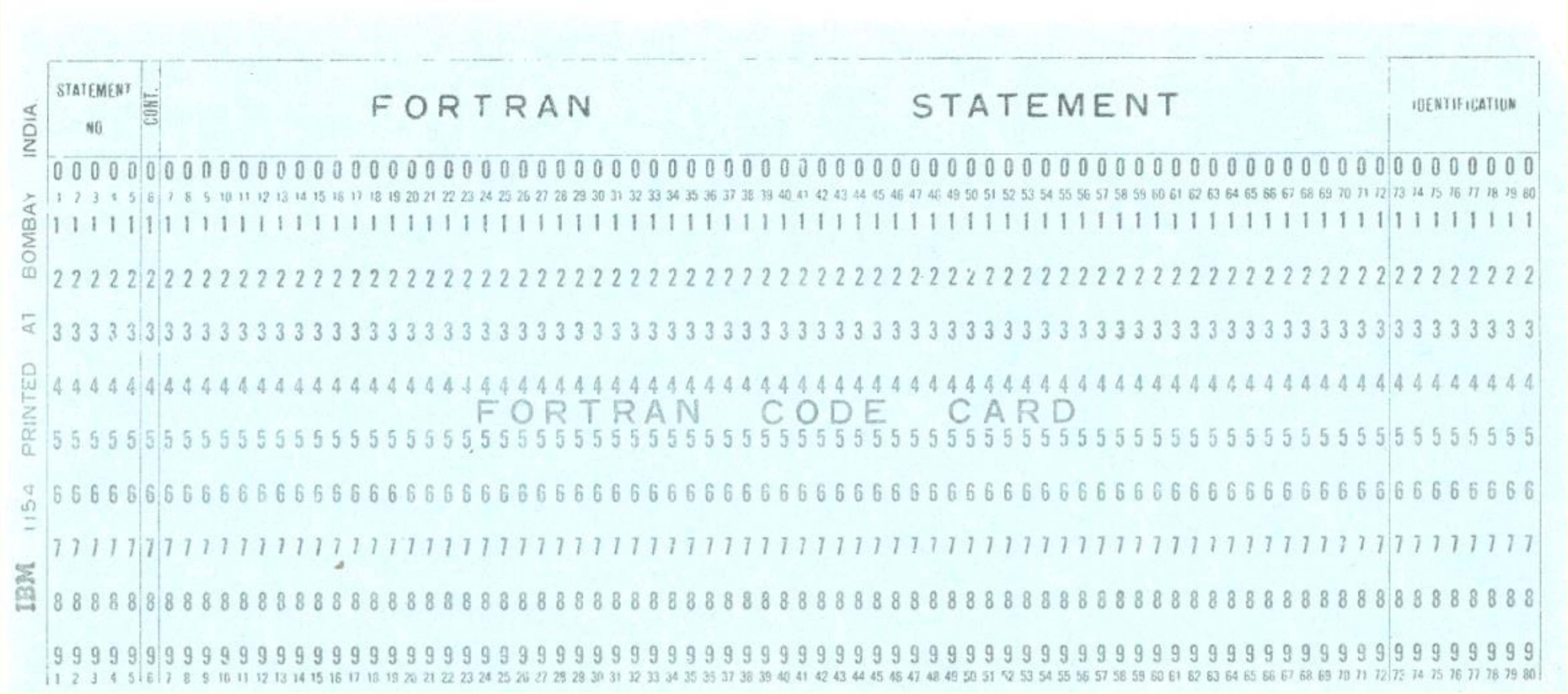
```
DO 10  I = 1.25          DO10I=1.25
DO 10  I = 1,25          DO10I=1,25
```

- The first line is a variable assignment
  DO10I=1.25

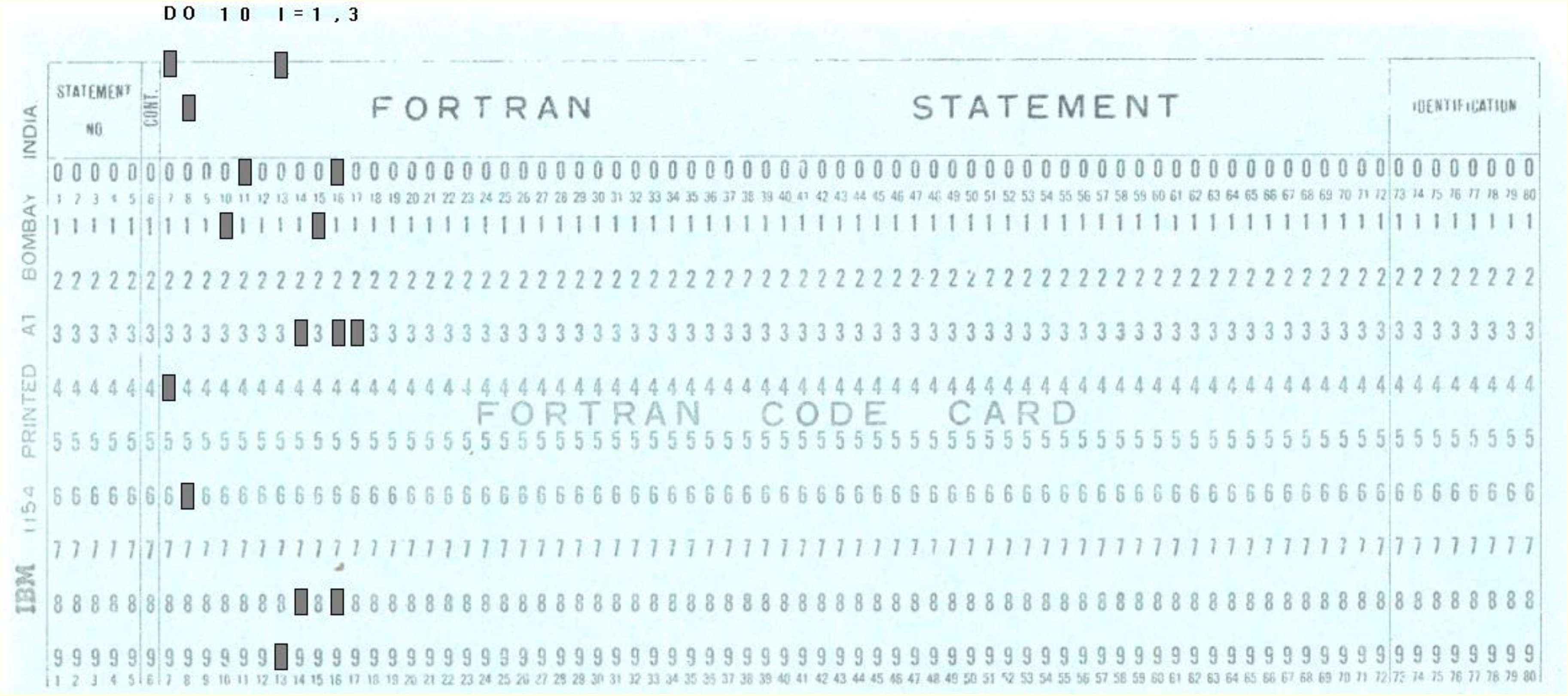
- The second line is beginning of a
  Do loop

- Reading from left to right one can not distinguish between the two until the “;” or “.” is reached

Fortran white space and fixed format rules came into force due to punch cards and errors in punching



Fortran white space and fixed format rules came into force due to punch cards and errors in punching



PL/1 Problems

- Keywords are not reserved in PL/1

  if then then then = else else else = then

  if if then then = then + 1

- PL/1 declarations

  Declare(arg$_1$,arg$_2$,arg$_3$,…….,arg$_n$)

- Cannot tell whether Declare is a keyword or array reference until after “)”
- Requires arbitrary lookahead and very large buffers.
  - Worse, the buffers may have to be reloaded.

Problem continues even today!!

- C++ template syntax: Foo<Bar>
- C++ stream syntax: cin >> var;
- Nested templates:

  Foo<Bar<Bazz>>
- Can these problems be resolved by lexical analyzers alone?

How to specify tokens?

- How to describe tokens

  2.e0   20.e-01   2.000

- How to break text into token

  if (x==0) a = x << 1;

  if (x==0) a = x < 1;

- How to break input into tokens efficiently
  - Tokens may have similar prefixes
  - Each character should be looked at only once

How to describe tokens?

- Programming language tokens can be described by regular languages
- Regular languages
  - Are easy to understand
  - There is a well understood and useful theory
  - They have efficient implementation
- Regular languages have been discussed in great detail in the “Theory of Computation” course

How to specify tokens

- Regular definitions
  - Let r_i be a regular expression and d_i be a distinct name
  - Regular definition is a sequence of definitions of the form

    $d_1 \rightarrow r_1$

    $d_2 \rightarrow r_2$

    …..

    $d_n \rightarrow r_n$
  - Where each r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Examples

- My fax number 91-(512)-259-7586
- Σ = digit U {-, (,) }
- Country → digit$^+$ digit^2
- Area → ‘(‘ digit$^+$ ‘)’ digit^3
- Exchange → digit$^+$ digit^3
- Phone → digit$^+$ digit^4
- Number → country ‘-’ area ‘-’ exchange ‘-’ phone

Examples …

- My email address karkare@iitk.ac.in
- Σ = letter U {@, . }
- letter → a| b| …| z| A| B| …| Z
- name → letter$^+$
- address → name ‘@’ name ‘.’ name ‘.’ name

Examples …

- Identifier

  letter → a| b| …|z| A| B| …| Z

  digit → 0| 1| …| 9

  identifier → letter(letter|digit)*

- Unsigned number in C

  digit → 0| 1| …|9

  digits → digit^{+}

  fraction → ’.’ digits | ϵ

  exponent → (E (‘+’ | ‘-’ | ϵ) digits) | ϵ

  number → digits fraction exponent

Regular expressions in specifications

- Regular expressions describe many useful languages
- Regular expressions are only specifications; implementation is still required
- Given a string s and a regular expression R, does s ∈ L(R) ?
- Solution to this problem is the basis of the lexical analyzers
- However, just the yes/no answer is not sufficient
- Goal: Partition the input into tokens

1. Write a regular expression for lexemes of each token
    - number → digit^+
    - identifier → $\text{letter}(\text{letter}|\text{digit})^+$
2. Construct R matching all lexemes of all tokens
    - R = R1 + R2 + R3 + …..
3. Let input be $x_1 \ldots x_n$
    - for $1 \le i \le n$ check $x_1 \ldots x_i \in L(R)$
4. $x_1 \ldots x_i \in L(R) \Rightarrow x_1 \ldots x_i \in L(Rj)$ for some j
    - smallest such j is token class of $x_1 \ldots x_i$
5. Remove $x_1 \ldots x_i$ from input; go to (3)

- The algorithm gives priority to tokens listed earlier
  - Treats “if” as keyword and not identifier
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$
  - Pick up the longest possible string in L(R)
  - The principle of “maximal munch”
- Regular expressions provide a concise and useful notation for string patterns
- Good algorithms require a single pass over the input

How to break up text

- Elsex=0

else	x	=	0

elsex	=	0

- Regular expressions alone are not enough
- Normally the longest match wins
- Ties are resolved by prioritizing tokens
- Lexical definitions consist of regular definitions, priority rules and maximal munch principle

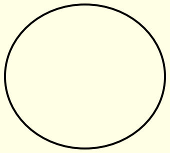
Transition Diagrams

- Regular expression are declarative specifications
- Transition diagram is an implementation
- A transition diagram consists of
  - An input alphabet belonging to Σ
  - A set of states S
  - A set of transitions $\text{state}_i \rightarrow^{input} \text{state}_j$
  - A set of final states F
  - A start state n
- Transition s1 $\rightarrow^{a}$ s2 is read:

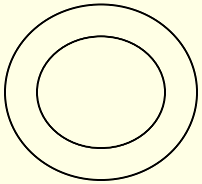
  in state s1 on input a go to state s2
- If end of input is reached in a final state then accept
- Otherwise, reject

Pictorial notation

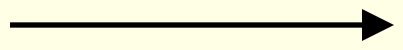
- A state



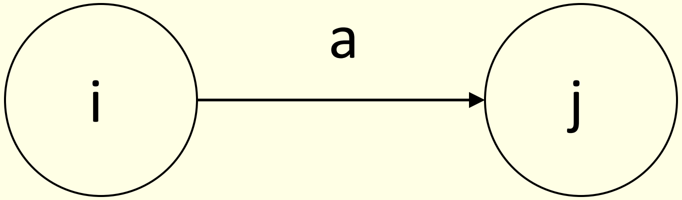
- A final state



- Transition



- Transition from state i to state j on an input a



How to recognize tokens

- Consider

  relop → < | <= | = | <> | >= | >

  id → letter(letter|digit)*

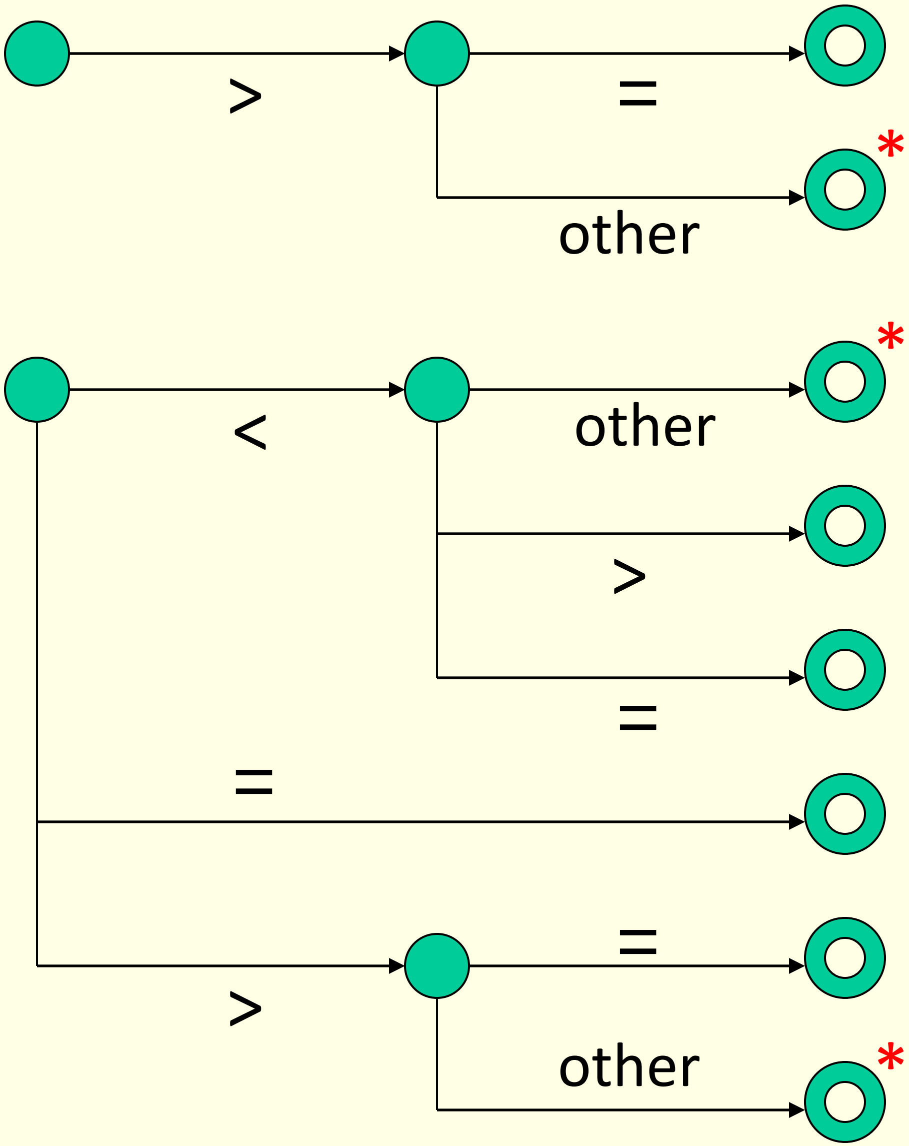
  num → digit$^+$ (‘.’ digit$^+$)? (E(‘+’|’-’)? digit$^+$)?

  delim → blank | tab | newline

  ws → delim$^+$

- Construct an analyzer that will return <token, attribute> pairs

Transition diagram for relops



token is relop, lexeme is >=

token is relop, lexeme is >

token is relop, lexeme is <

token is relop, lexeme is <>

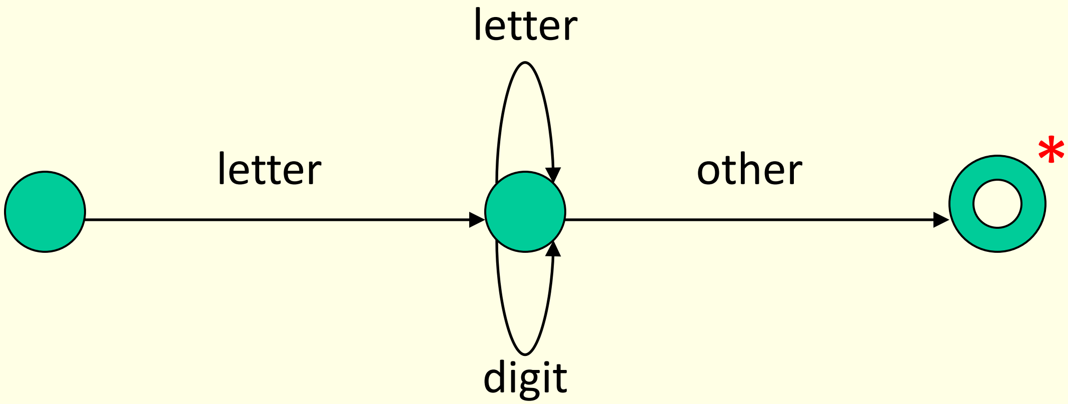
token is relop, lexeme is <=

token is relop, lexeme is =

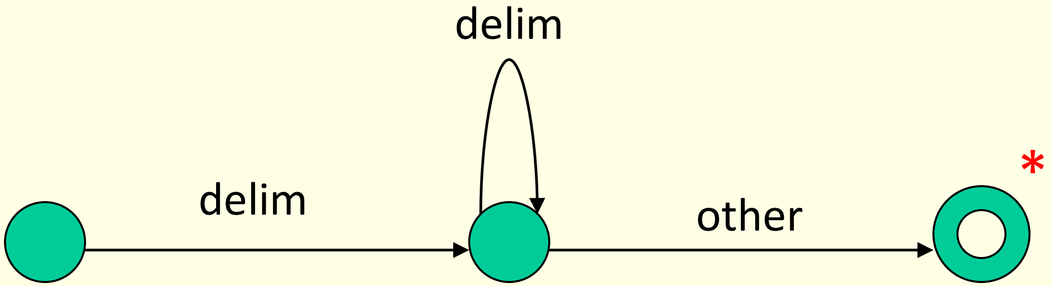
token is relop, lexeme is >=

token is relop, lexeme is >

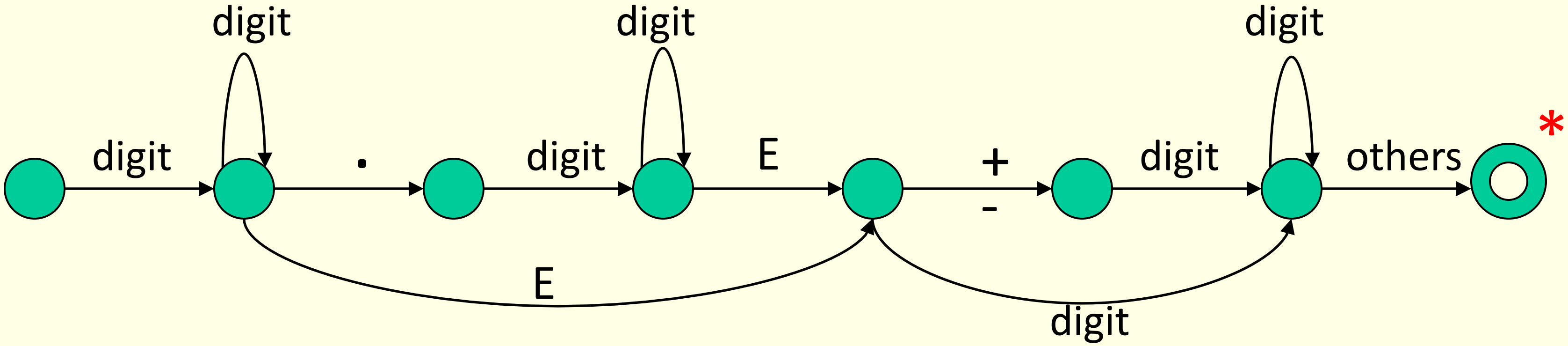
Transition diagram for identifier

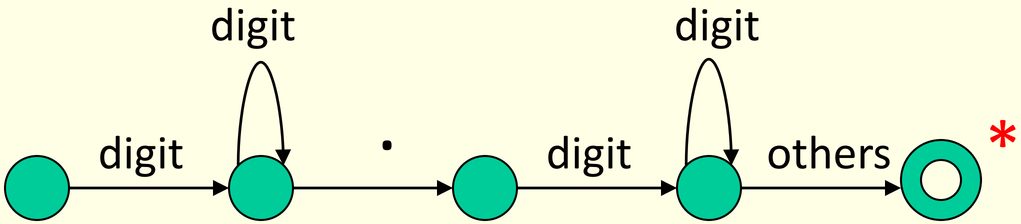


Transition diagram for white spaces

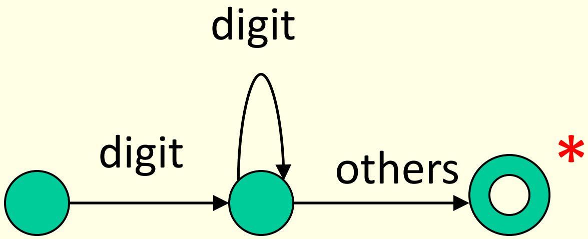


Transition diagram for unsigned numbers





Real numbers



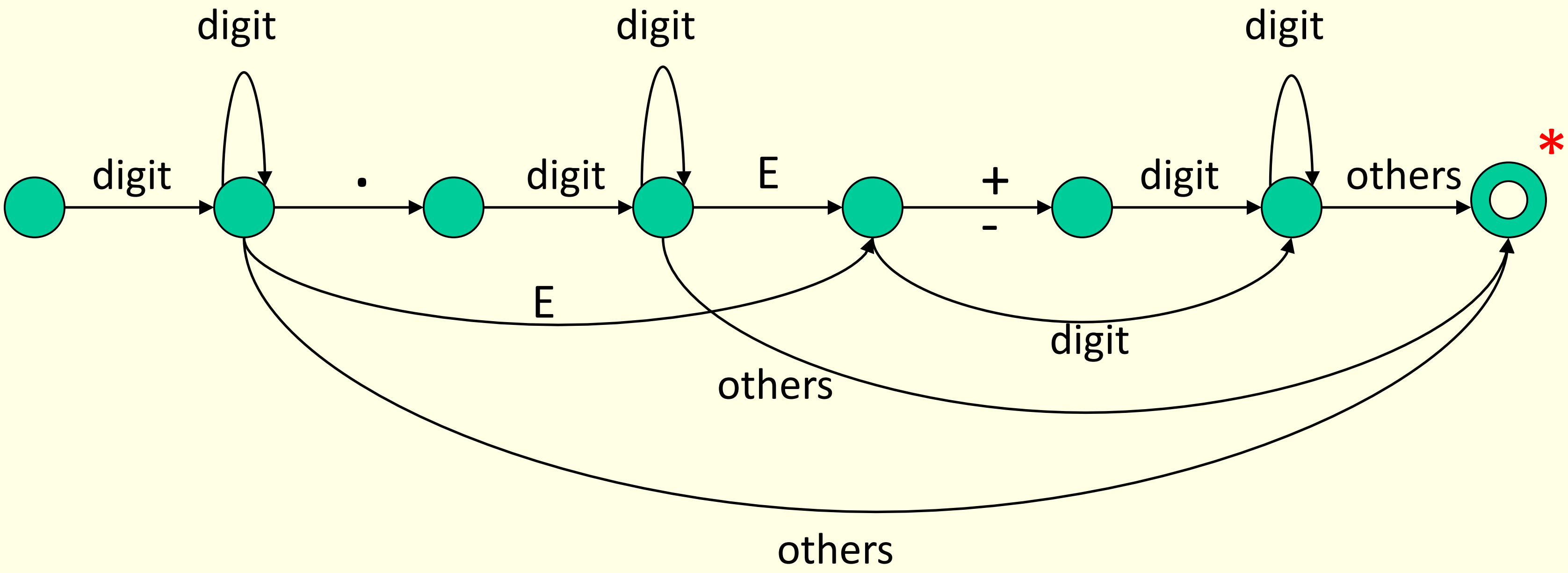
Integer number

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- Starting in the third diagram the accept state will be reached after 12
- Therefore, the matching should always start with the first transition diagram
- If failure occurs in one transition diagram then retract the forward pointer to the start state and activate the next diagram
- If failure occurs in all diagrams then a lexical error has occurred

Implementation of transition diagrams

```
Token nexttoken() {
    while(1) {
          switch (state) {
                    ……
                    case 10: c=nextchar();
                      if(isletter(c)) state=10;
                      elseif (isdigit(c)) state=10;
                      else state=11;
                      break;
                    ……
                    }
          }
}
```

Another transition diagram for unsigned numbers



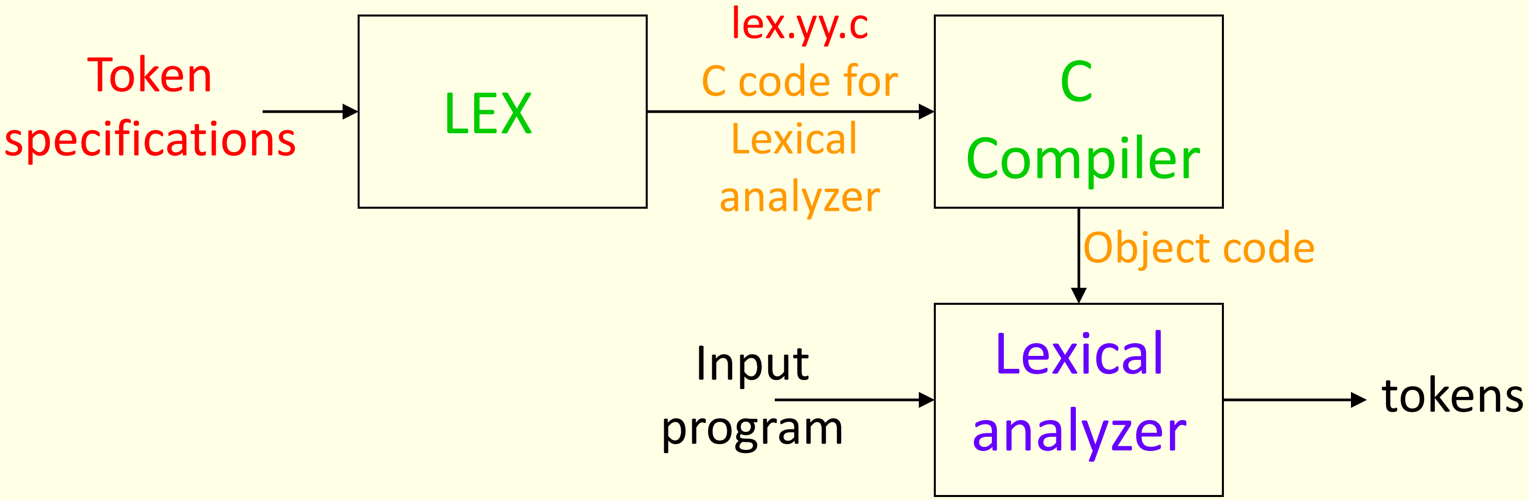
A more complex transition diagram
is difficult to implement and
may give rise to errors during coding, however,
there are ways to better implementation

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other book keeping information)

- Output of the generator
  - Program that reads input character stream and breaks that into tokens
  - Reports lexical errors (unexpected characters), if any

LEX: A lexical analyzer generator



Refer to LEX User's Manual

How does LEX work?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non deterministic finite automaton (NFA)
- Convert the NFA into an equivalent DFA
- Minimize the DFA to reduce number of states
- Emit code driven by the DFA tables