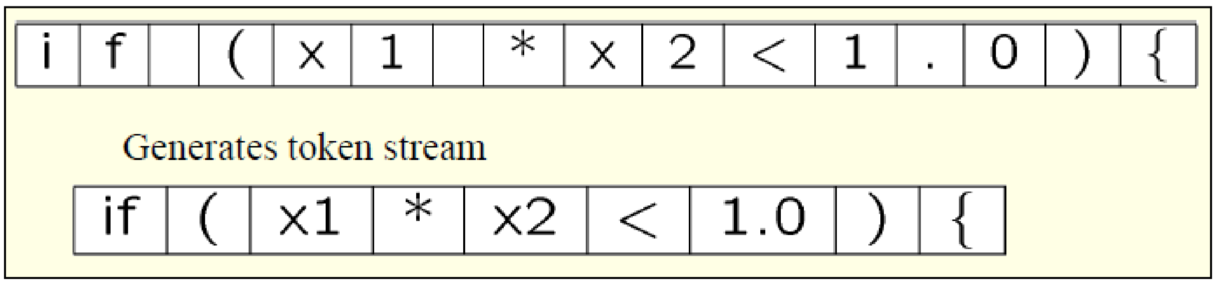
UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Compiler Design

Prof. Sankhadeep Chatterjee

Lexical Analysis

- Recognize tokens and ignore white spaces, comments



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

Lexical Analysis

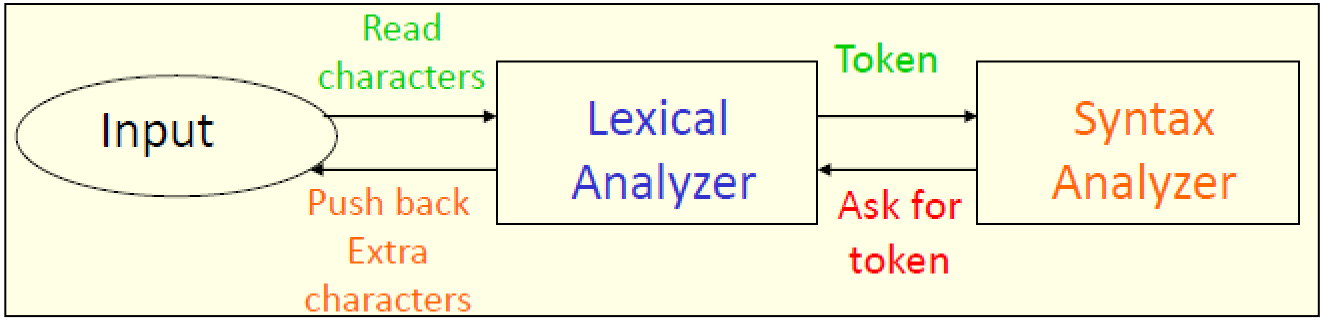
- Sentences consist of string of tokens (a syntactic category)
  - For example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme
  - for example, 100.01, counter, const, “How are you?”
- Rule of description is a pattern for example, letter (letter | digit)*
- Task: 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

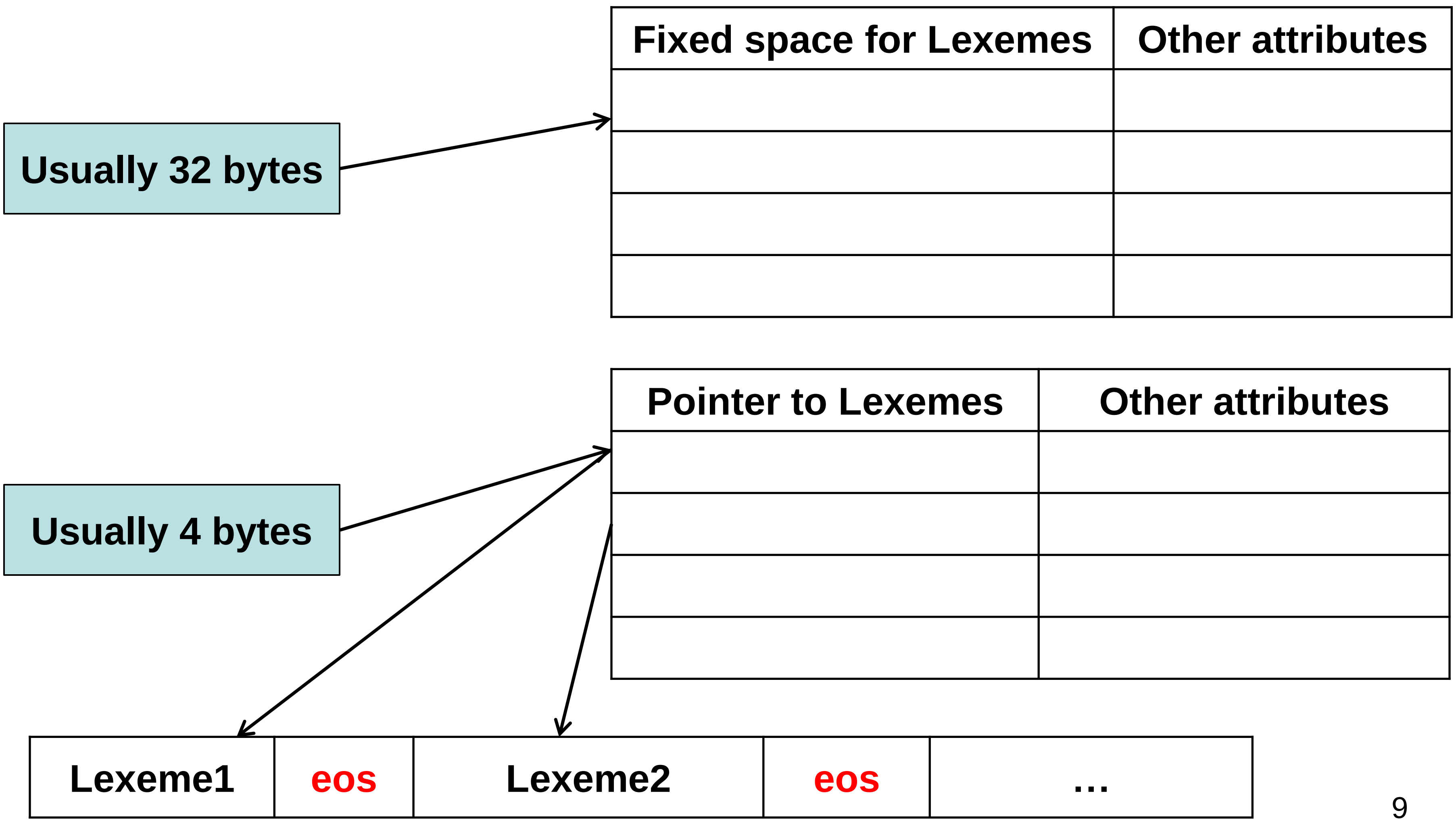
Symbol Table

- 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.

Lexical Analysis



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.

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

$$\mathbf{d_1 \rightarrow r_1}$$
$$\mathbf{d_2 \rightarrow r_2}$$
$$\mathbf{\ldots}$$
$$\mathbf{d_n \rightarrow r_n}$$

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

Examples

- Fax number
  - 91-(123)-456-7890
  - Σ = digit U {-, (,) }
  - Country → digit$^+$
  - Area → ‘(‘ digit$^+$ ‘)’
  - Exchange → digit$^+$
  - Phone → digit$^+$
  - Number → country ‘-’ area ‘-’exchange ‘-’ phone

Examples

- Email address
  - xyz.official@company.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 $s \in L(R)$?
- 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

Algorithm

- Write a regular expression for lexemes of each token e.g.
  - number $\rightarrow$ digit$^+$
  - identifier $\rightarrow$ letter(letter|digit)$^+$
- Construct R matching all lexemes of all tokens

  R = R1 + R2 + R3 + …..

  1. Let input be $x_1 \ldots x_n$

     for $1 \leq i \leq n$ check $x_1 \ldots x_i \in L(R)$
  2. $x_1 \ldots x_i \in L(R) \rightarrow x_1 \ldots x_i \in L(R_j)$ for some j

     smallest such j is token class of $x_1 \ldots x_i$
  3. Remove $x_1 \ldots x_i$ from input; go to (1)

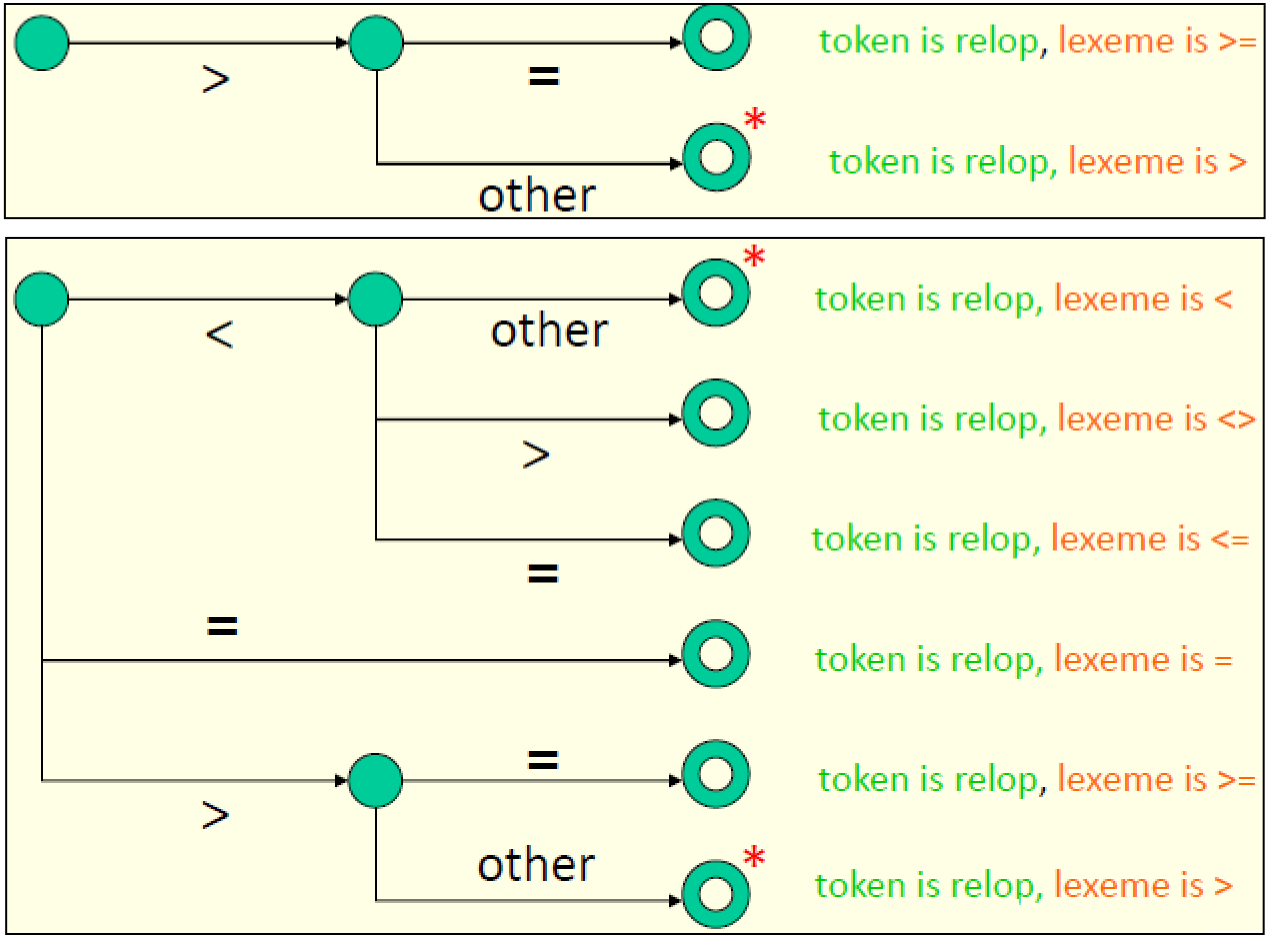
Algorithm contd.

- 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”

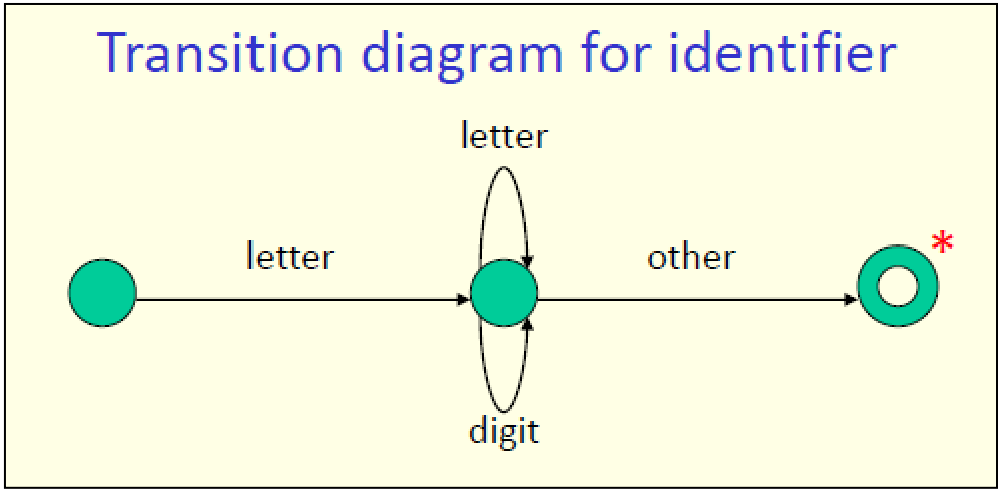
Examples

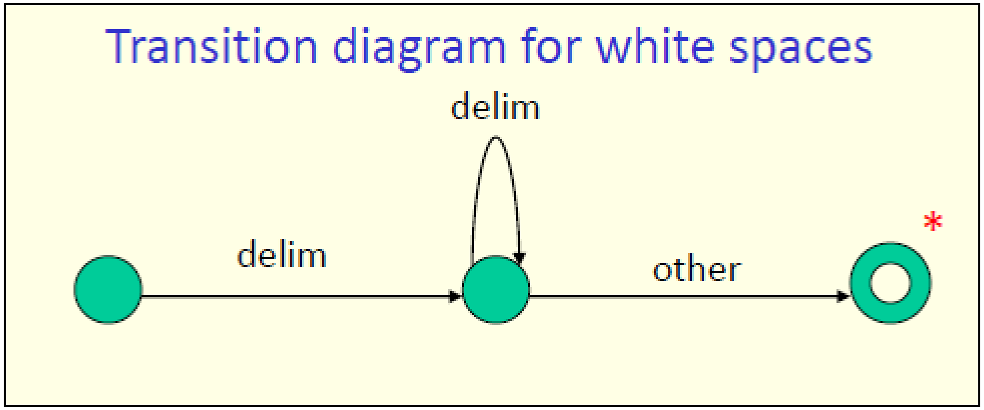
- Consider
  - relop → < | <= | = | <> | >= | >
  - id → letter(letter|digit)*
  - num → $digit^+$('.' $digit^+$)?(E('+'|'-')?$digit^+$)?
  - delim → blank | tab | newline
  - ws → $delim^+$
- Construct an analyzer that will return <token, attribute> pairs

Transition Diagram for relops

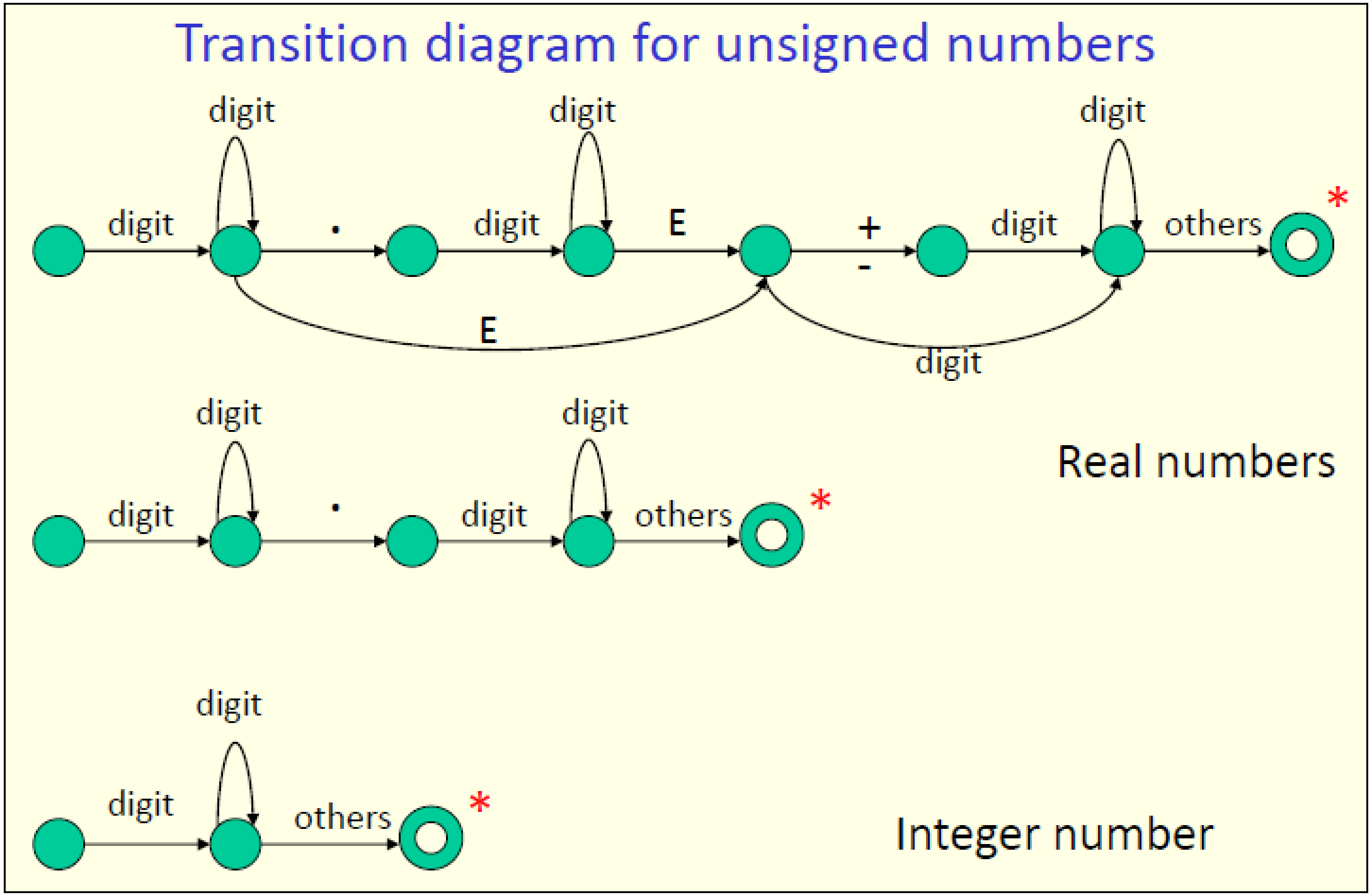


Transition Diagram contd.





Transition Diagram for Unsigned numbers



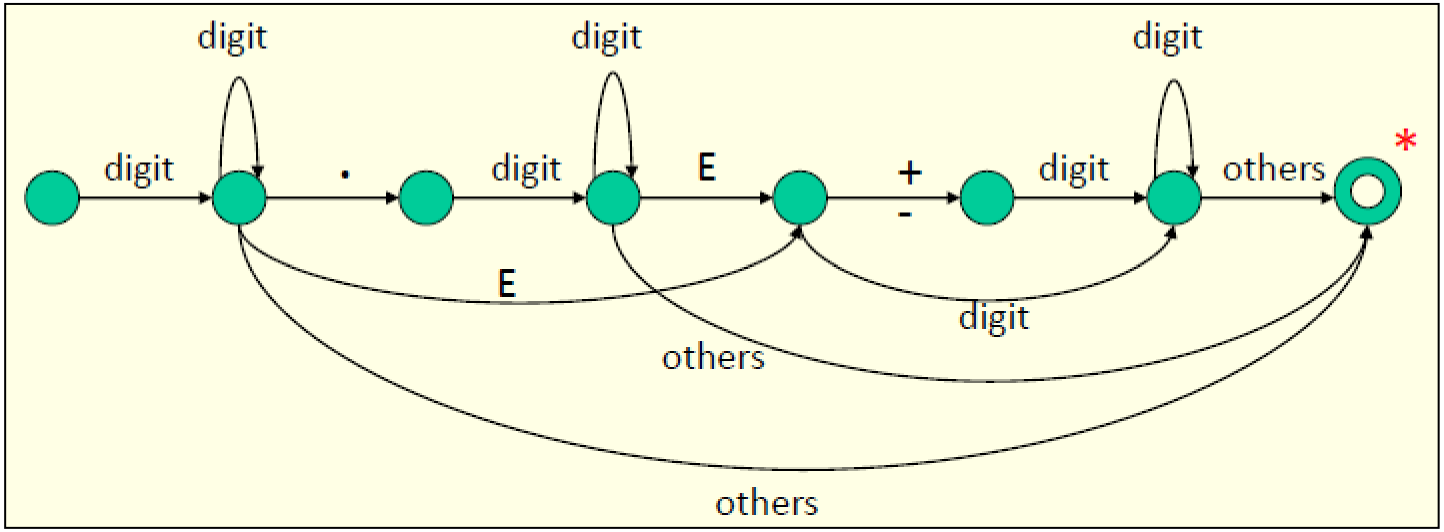
Transition Diagram for Unsigned numbers

- 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 Diagram

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

Transition diagram for unsigned numbers



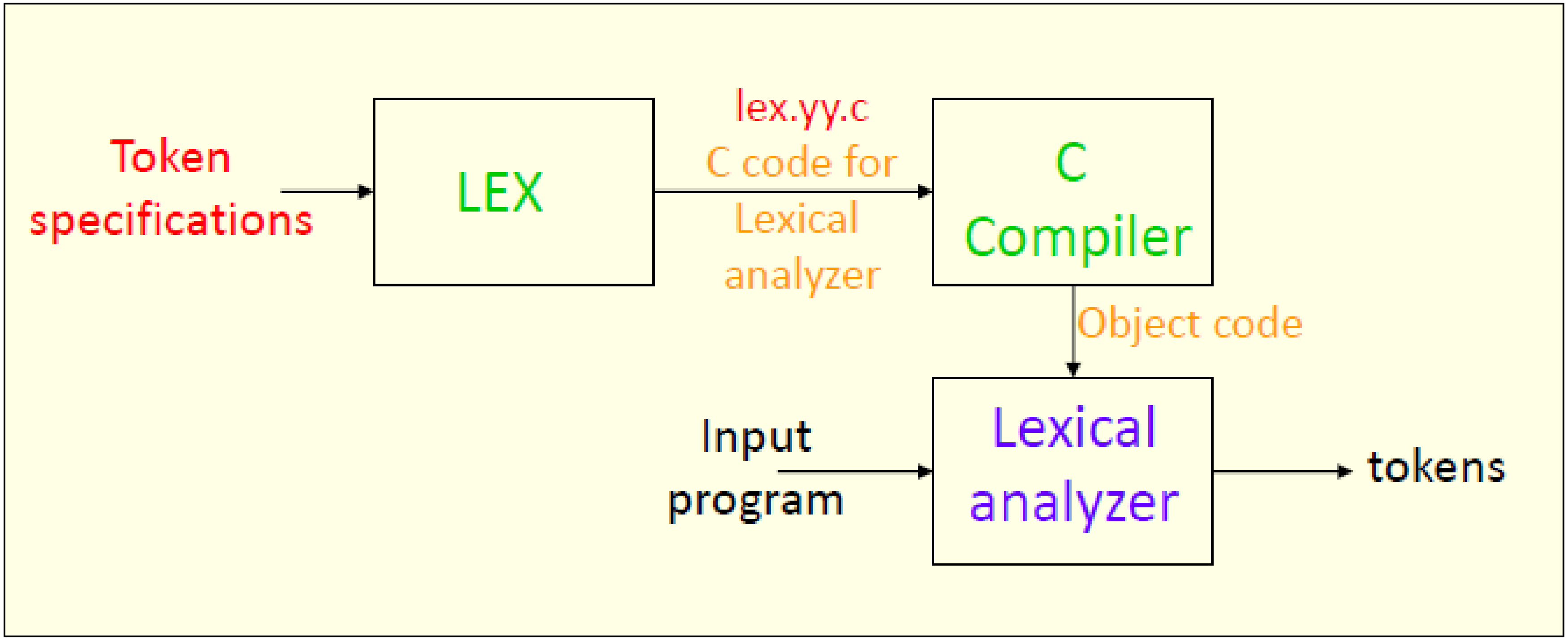
- A more complex transition diagram is difficult to implement and may give rise to errors during coding, however, there are ways to better implementation

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



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

Thank You