Acknowledgements

The slides for this lecture are a modified versions of the offering by **Prof. Sanjeev K Aggarwal**

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

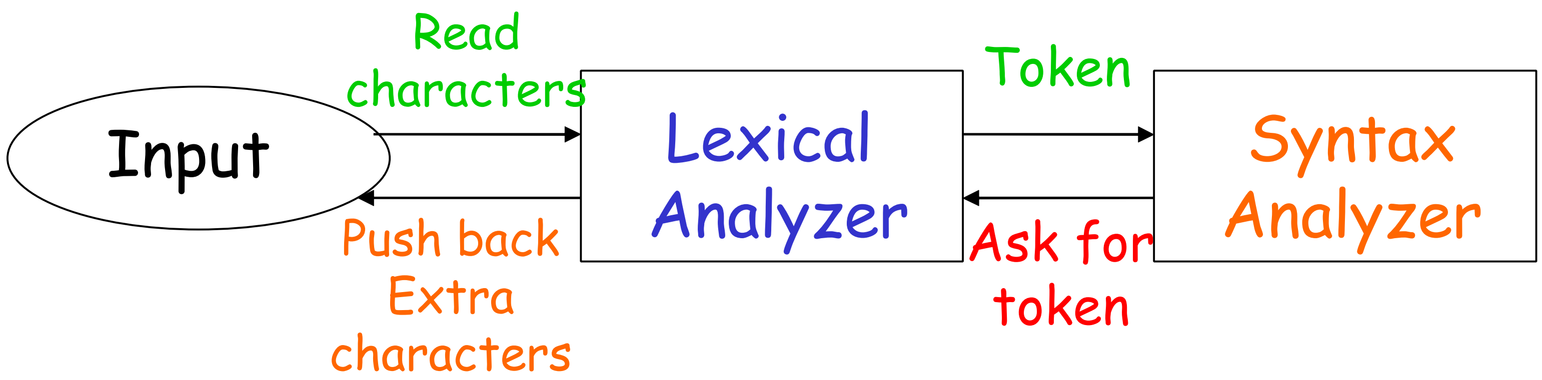
- Model using regular expressions

- 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)*
- Discard whatever does not contribute to parsing like white spaces (blanks, tabs, newlines) and comments
- construct constants: convert numbers to token num and pass number as its attribute, for example, integer 31 becomes <num, 31>
- recognize keyword and identifiers
  for example counter = counter + increment
  becomes id = id + id /*check if id is a keyword*/

Interface to other phases



- Push back is required due to lookahead for example > = and >

- It is 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

Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression

- Return tokens and attributes to the syntax analyzer

- A global variable **tokenval** is set to the value of the number

- Design requires that
  - A finite set of tokens be defined
  - Describe strings belonging to each token

```
#include <stdio.h>
#include <ctype.h>
int  lineno = 1;
int tokenval = NONE;
int lex() {
        int t;
        while (1) {
        t = getchar ();
        if (t ==‘ ‘ || t == ‘\t’);
                else if (t == ‘\n’) lineno = lineno + 1;
                else if (isdigit (t) ) {
                                tokenval = t - ‘0’ ;
                                t = getchar ();
                                while (isdigit(t)) {
                                    tokenval = tokenval * 10 + t - ‘0’ ;
                                    t = getchar();
                                }
                                ungetc(t,stdin);
                                return num;
                        }
                        else { tokenval = NONE; return t; }
        }
}
```

Problems

- Scans text character by character
- Look ahead character determines what kind of token to read and when the current token ends
- First character cannot determine what kind of token we are going to read

Difficulties in design of lexical analyzers

- Is it as simple as it sounds?

- Lexemes in a fixed position. Fix format vs. free format languages

- Handling of blanks
  - in Pascal, blanks separate identifiers
  - in Fortran, blanks are important only in literal strings for example variable counter is same as count er
  - Another example

    DO 10 I = 1.25   DO10I=1.25

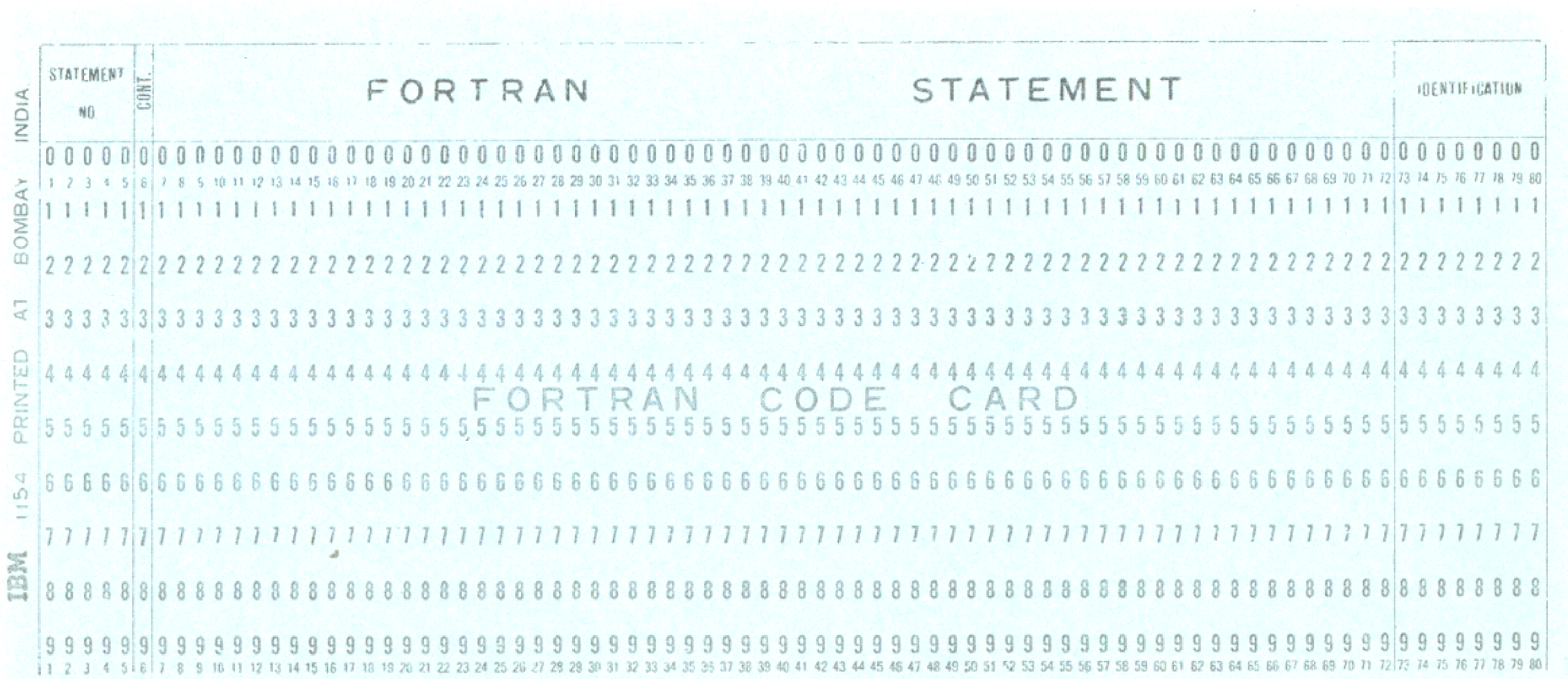
    DO 10 I = 1,25   DO10I=1,25

- The first line is a variable assignment
  DO10I=1.25

- 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


STATEMENT NO
CONT.
FORTRAN STATEMENT
IDENTIFICATION
FORTRAN CODE CARD
IBM 1154 PRINTED AT BOMBAY INDIA

```
DO 10 I=1,3
```

STATEMENT NO
CONT.
FORTRAN STATEMENT
IDENTIFICATION
FORTRAN CODE CARD
IBM 1154 PRINTED AT BOMBAY INDIA

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?

Building a lexer...

Need to answer the following questions:

- How to specify tokens (patterns)?
- How to recognize tokens (efficiently)?
  - Not just a language acceptance question...
- How to resolve conflicts?
  - Priority
  - Maximal munch

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;

  iff (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

Notation

- Let Σ be a set of characters. A language over Σ is a set of strings of characters belonging to Σ

- A regular expression r denotes a language L(r)

- Rules that define the regular expressions over Σ
  - ϵ is a regular expression that denotes $\{\epsilon\}$ the set containing the empty string
  - If a is a symbol in Σ then a is a regular expression that denotes $\{a\}$

- If r and s are regular expressions denoting the languages L(r) and L(s) then
- (r)|(s) is a regular expression denoting L(r) U L(s)
- (r)(s) is a regular expression denoting L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

- Let $\Sigma = \{a, b\}$

- The regular expression $a|b$ denotes the set $\{a, b\}$

- The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$

- The regular expression a^* denotes the set of all strings $\{\epsilon, a, aa, aaa, \ldots\}$

- The regular expression $(a|b)^*$ denotes the set of all strings containing ϵ and all strings of a's and b's

- The regular expression $a|a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's followed by a character b

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 -> r_1

    d_2 -> r_2

    …..

    d_n -> 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 = digits U {-, (,) }

- Country -> $digit^+$ $digit^2$

- Area -> '(' $digit^+$ ')' $digit^3$

- Exchange -> $digit^+$ $digit^3$

- Phone -> $digit^+$ $digit^4$

- Number -> country '-' area '-' exchange '-' phone

Examples …

- Email address
  abc@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 Pascal

  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 important

- Goal: Partition the input into tokens

1. Write a regular expression for lexemes of each token
    - number $\rightarrow$ digit$^+$
    - identifier $\rightarrow$ letter(letter|digit)$^+$

2. Construct R matching all lexemes of all tokens
    - R = R1 + R2 + R3 + ...

3. Let input be $x_1 \ldots x_n$
    - for $1 \leq i \leq n$ 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

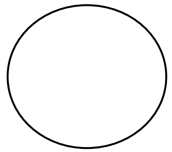
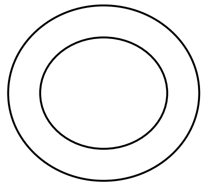
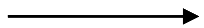
- A transition diagram consists of
  - An input alphabet belonging to Σ
  - A set of states S
  - A set of transitions $state_i \xrightarrow{input} state_j$
  - A set of final states F
  - A start state n

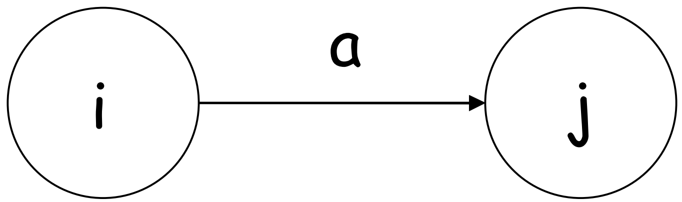
- Transition $s1 \xrightarrow{a} s2$ is read:
  in state s1 on input a go to state s2

- 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)*

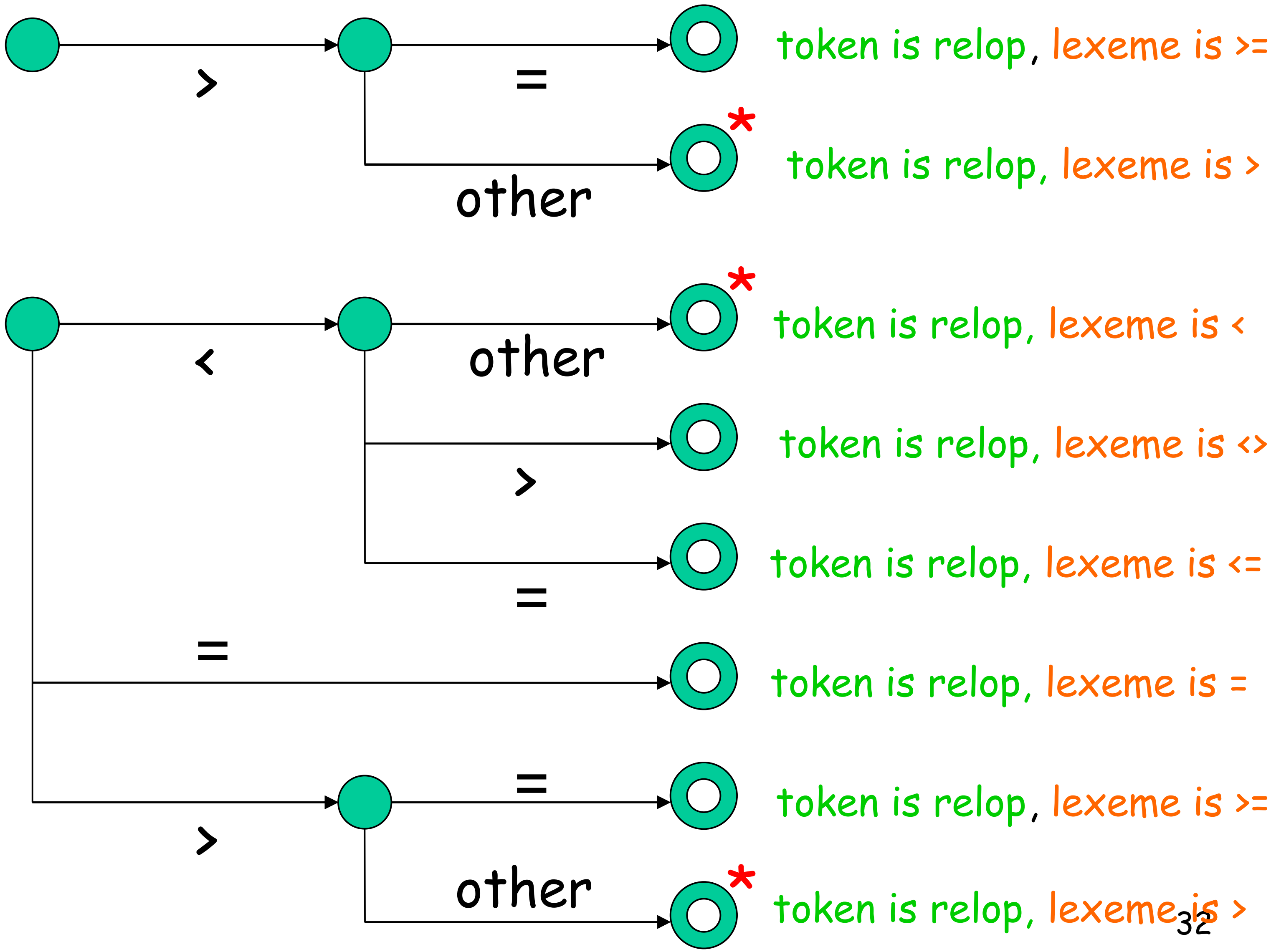
  num → digit$^+$ ('.' digit$^+$)? (E('+'|'-')? digit$^+$)?

  delim → blank | tab | newline

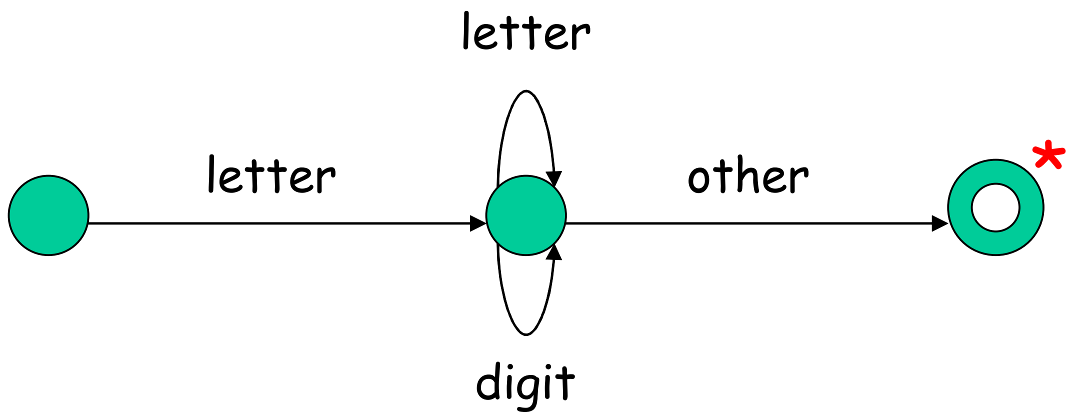
  ws → delim$^+$

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

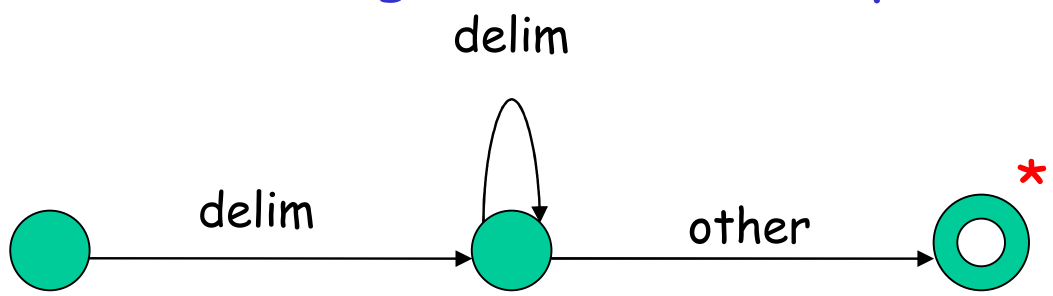
Transition diagram for relops



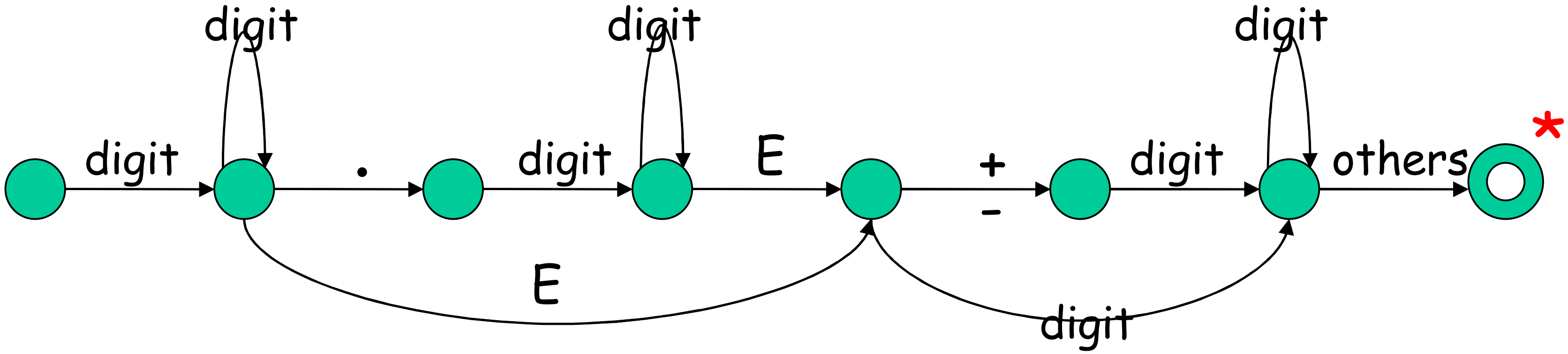
Transition diagram for identifier

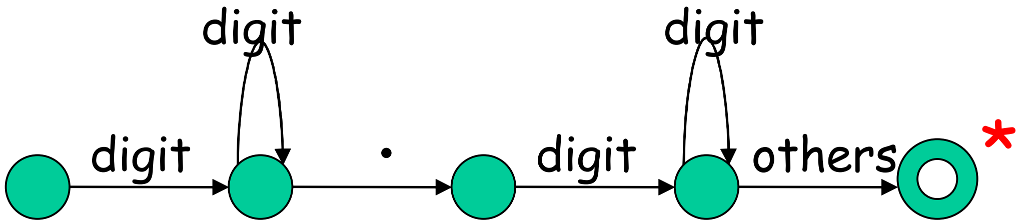


Transition diagram for white spaces

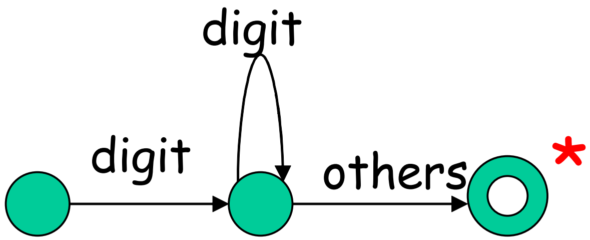


Transition diagram for unsigned numbers





Real numbers



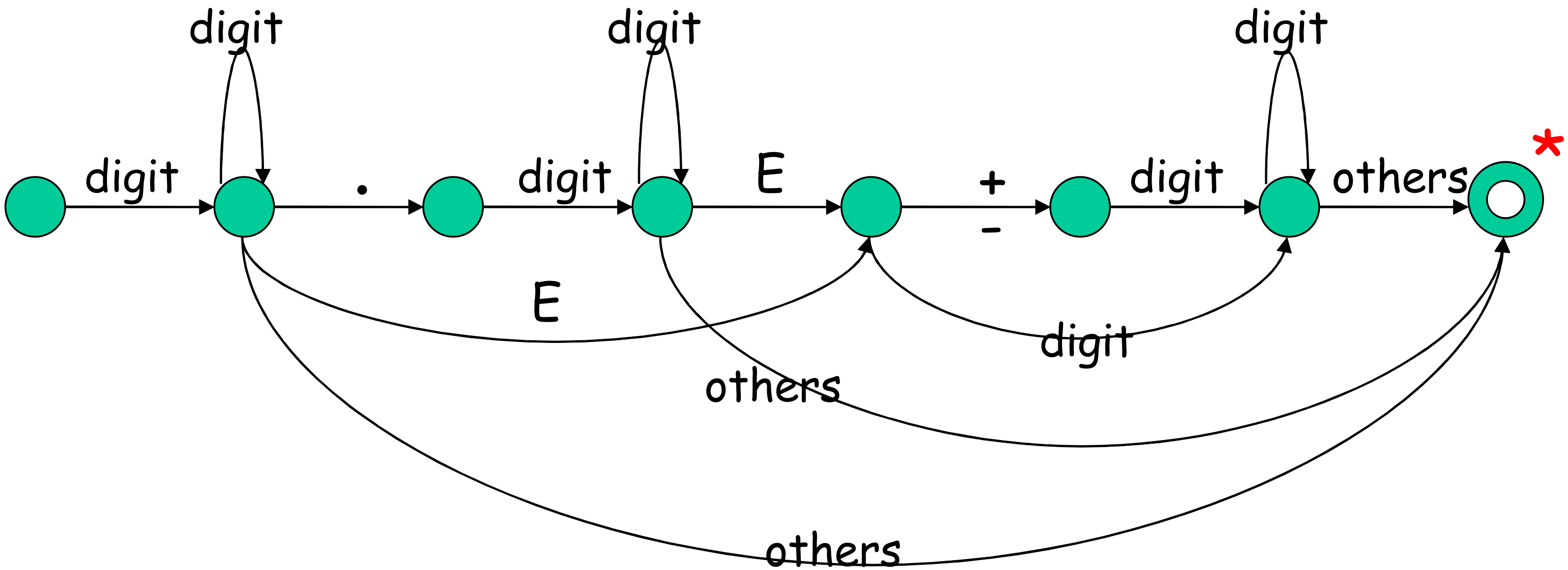
Integer number

- 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



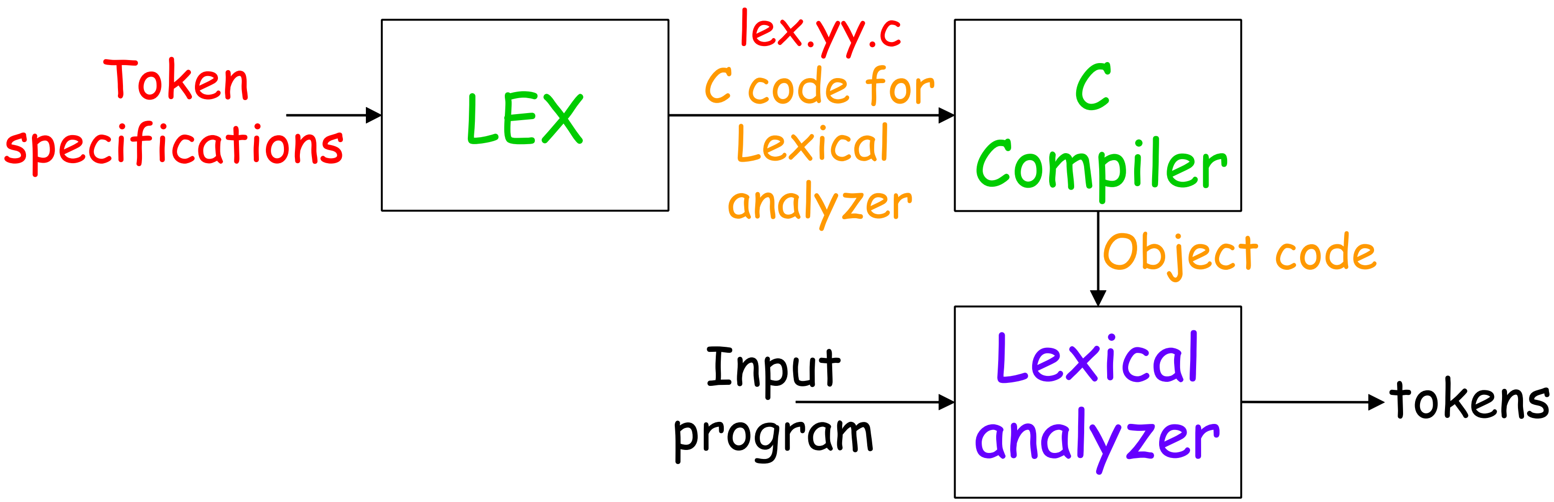
A more complex transition diagram
is difficult to implement and
may give rise to errors during coding,
however,
there are ways to better implementation

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