MODULE-3

Lexical Analysis

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer

Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

- **Attributes for tokens**

  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>

  <number, integer value 2>

- **Lexical errors**

  Some errors are out of power of lexical analyzer to recognize:
  - **fi (a == f(x)) …**

  However it may be able to recognize errors like:
  - **d = 2r**

  Such errors are recognized when no pattern for tokens matches a character sequence

- **Error recovery**

1. Panic mode: successive characters are ignored until we reach to a well formed token
2. Delete one character from the remaining input
3. Insert a missing character into the remaining input

4. Replace a character by another character
5. Transpose two adjacent characters

➢ Input buffering

Sentinels

➢ Specification of tokens

1. In theory of compilation regular expressions are used to formalize the specification of tokens
2. Regular expressions are means for specifying regular languages
3. Example:

   i. Letter_(letter_ | digit)*

4. Each regular expression is a pattern specifying the form of strings

➢ Regular expressions

1. Ɛ is a regular expression, L(Ɛ) = {Ɛ}
2. If a is a symbol in ∑then a is a regular expression, L(a) = {a}
3. (r) | (s) is a regular expression denoting the language L(r)   L(s)
4. (r)(s) is a regular expression denoting the language L(r)L(s)
5. (r)* is a regular expression denoting (L(r))*

6. (r) is a regular expression denoting L(r)

➢ Regular definitions

1. d1 -> r1
2. d2 -> r2
3. …
4. dn -> rn

5. Example:

6. letter_ -> A | B | … | Z | a | b | … | Z | _
7. digit -> 0 | 1 | … | 9
8. id -> letter_ (letter_ | digit)*

➢ Extensions

One or more instances: (r)+

Zero of one instances: r?

Character classes: [abc]

Example:

letter_ -> [A-Za-z_]

digit -> [0-9]

id -> letter_(letter|digit)*

➢ Recognition of tokens

Starting point is the language grammar to understand the tokens:

```
stmt -> if expr then stmt
        |  if expr then stmt else stmt
        | Ɛ
expr -> term relop term
        |  term
term -> id
        |  number
```

➢ Recognition of tokens (cont.)

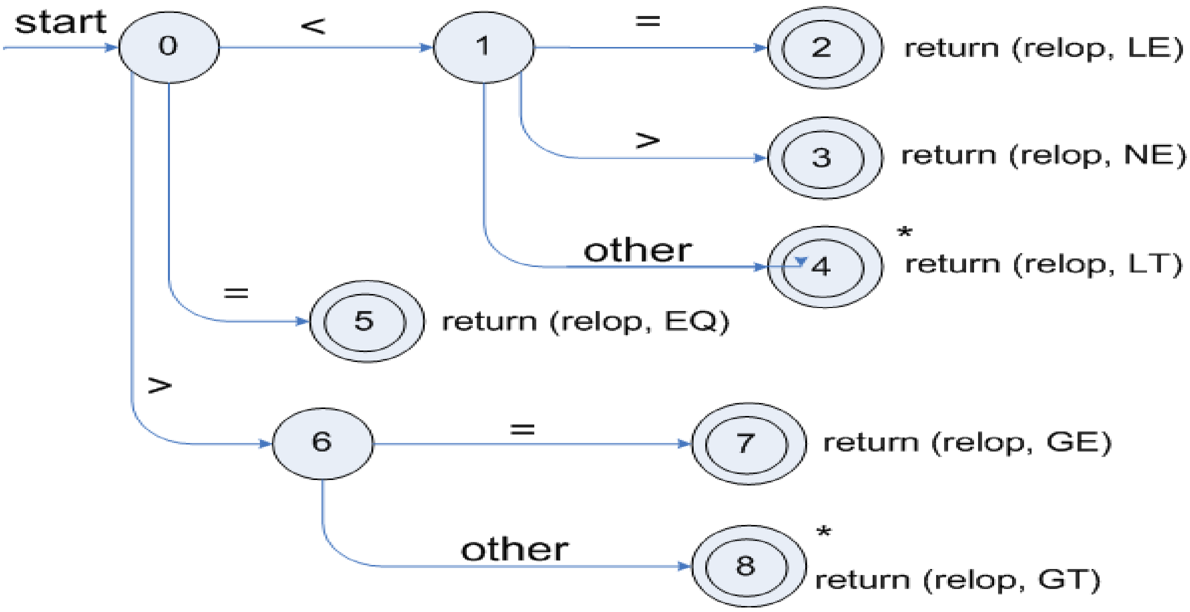
The next step is to formalize the patterns:

digit **-> [0-9]**
Digits **-> digit+**
number **-> digit(.digits)? (E[+-]? Digit)?**
letter **-> [A-Za-z_]**
id **-> letter (letter|digit)***
If **-> if**
Then **-> then**
Else **-> else**
Relop **-> < | > | <= | >= | = | <>**

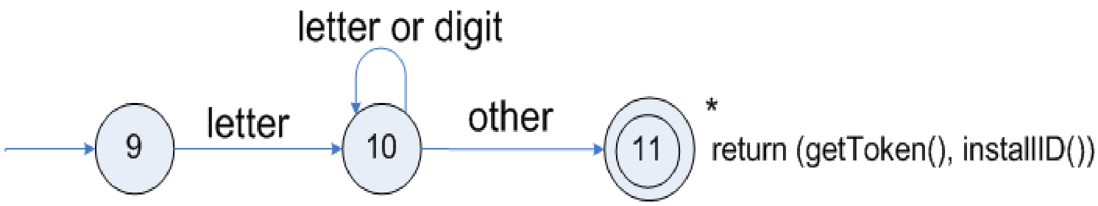
We also need to handle whitespaces:

ws **-> (blank | tab | newline)+**

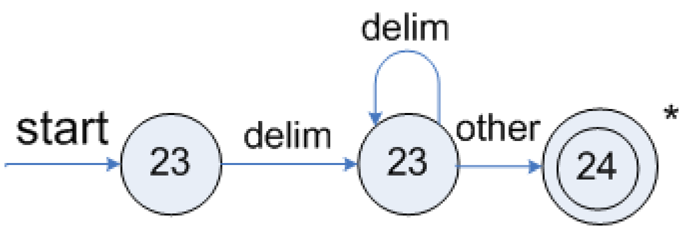
- **Transition diagrams**



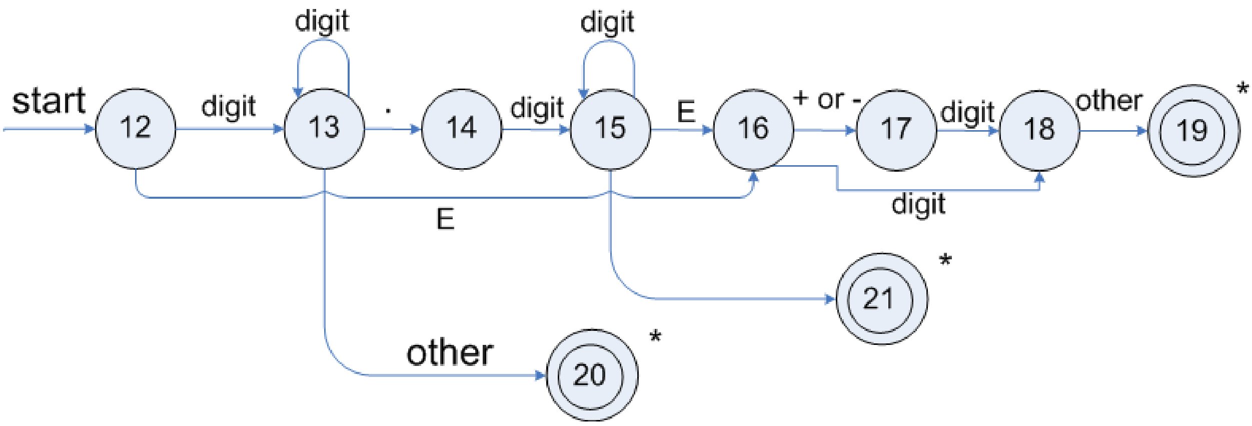
- **Transition diagrams (cont.)**



- **Transition diagram for whitespace**



- **Transition diagram for unsigned numbers**



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
        TOKEN retToken = new (RELOP)
        while (1) {      /* repeat character processing until a
                            return or failure occurs      */
        switch(state) {
          case 0: c= nextchar();
                  if (c == '<') state = 1;
                  else if (c == '=') state = 5;
                  else if (c == '>') state = 6;
                  else fail();    /* lexeme is not a relop */
                  break;
          case 1: ...
          ...
          case 8: retract();
                  retToken.attribute = GT;
                  return(retToken);
        }
```

- **Finite Automata**
- Regular expressions = specification

- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet
  - A set of states S
  - A start state n
  - A set of accepting states F   S
  - A set of transitions state   input state

- Transition

$s_1 \quad ^a s_2$

- Is read

In state s_1 on input "a" go to state s_2

- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Example

- Alphabet still { 0, 1 }

  The operation of the automaton is not completely defined by the input

  On input "11" the automaton could be in either state

MODULE-4