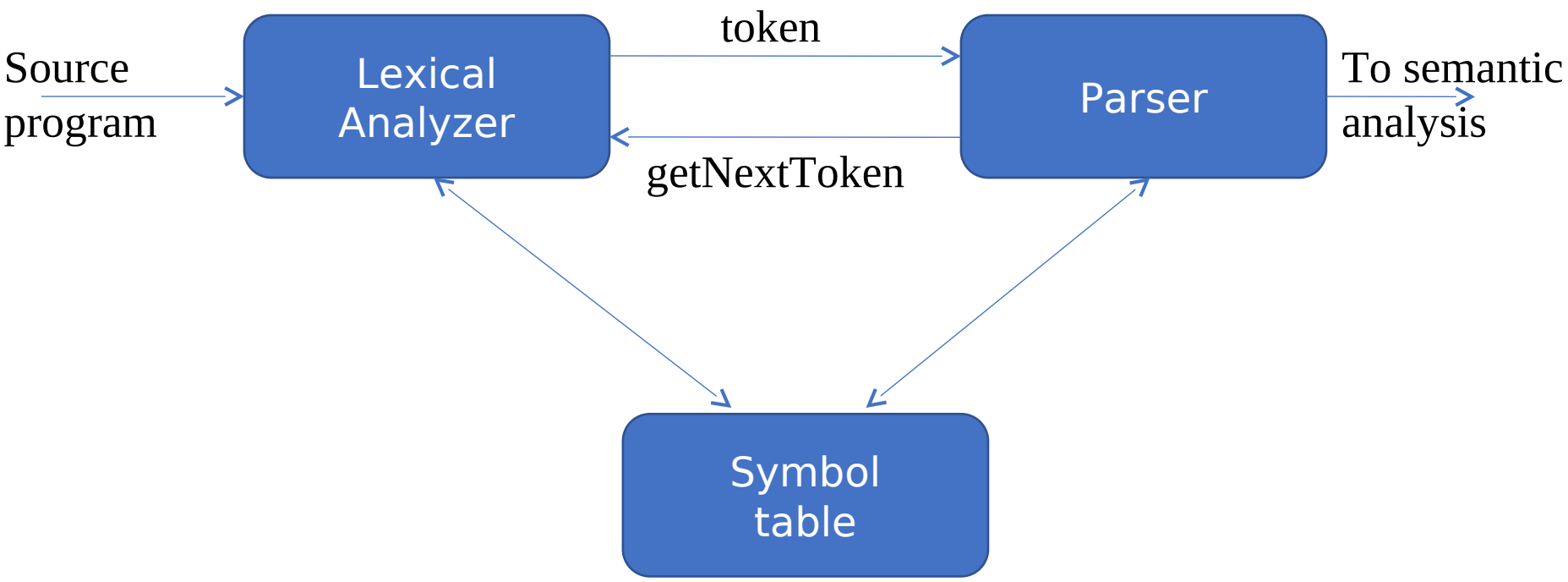
COMPILER DESIGN

Topic: Lexical Analysis

Soma Ghosh - gsn.comp@coeptech.ac.in

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

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ sorrounded by “	“core dumped”

printf(“total = %d\n”, score);

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

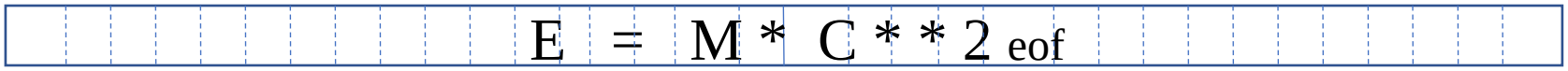
- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) ...
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

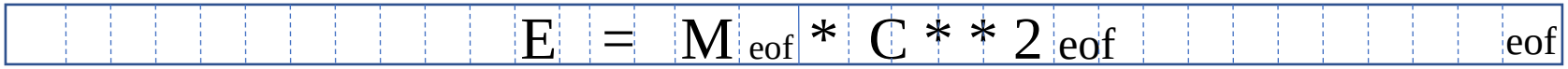
- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely



Sentinels




```
Switch (*forward++) {
  case eof:
        if (forward is at end of first buffer) {
              reload second buffer;
              forward = beginning of second buffer;
        }
        else if {forward is at end of second buffer) {
              reload first buffer;\
              forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
              terminate lexical analysis;
        break;
  cases for the other characters;
}
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in $\sum$ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

...

dn -> rn

- Example:

  letter_ -> A | B | ... | Z | a | b | ... | Z | _

  digit -> 0 | 1 | ... | 9

  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
  | ε

  expr -> term **relop** term
  | term

  term -> **id**
  | **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

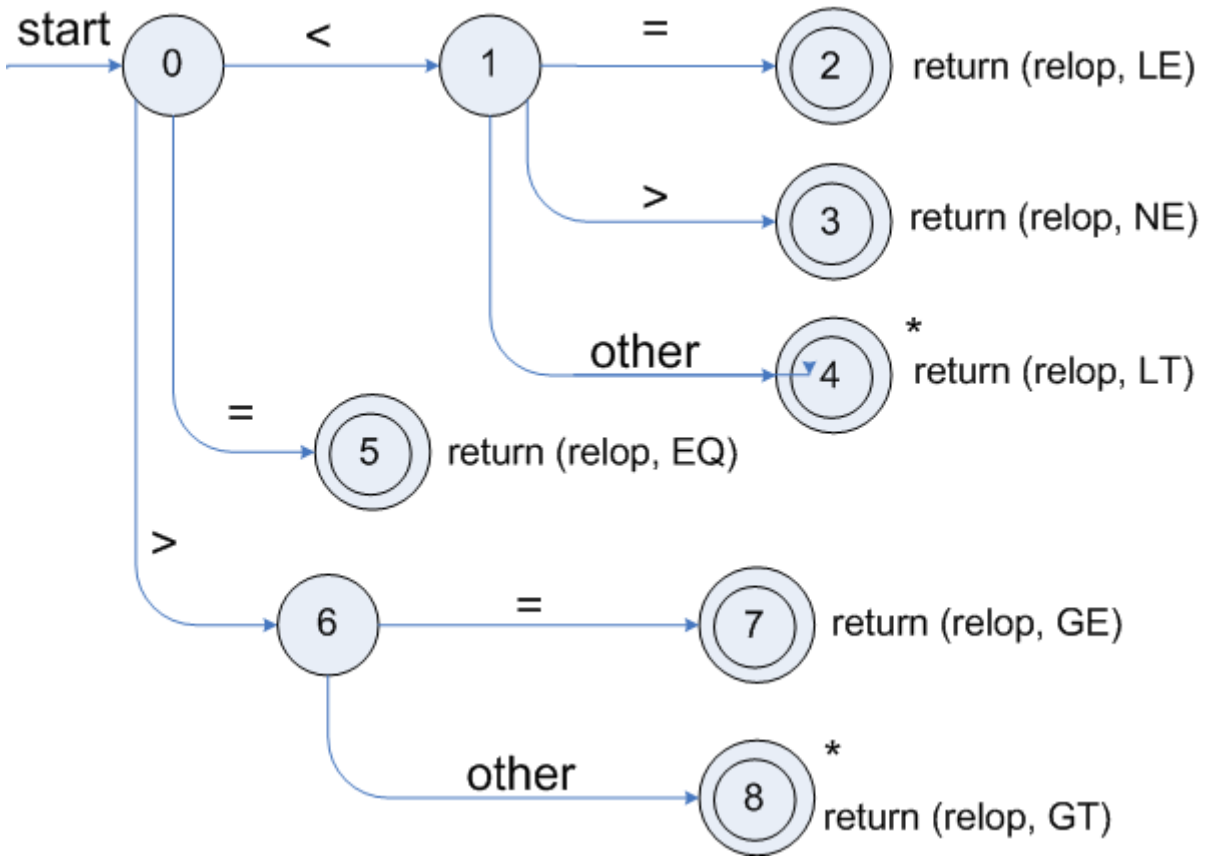
  digit -> [0-9]
  Digits -> digit+
  number -> digit(.digits)? (E[+-]? Digit)?
  letter -> [A-Za-z_]
  id -> letter (letter|digit)*
  If -> if
  Then -> then
  Else -> else
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

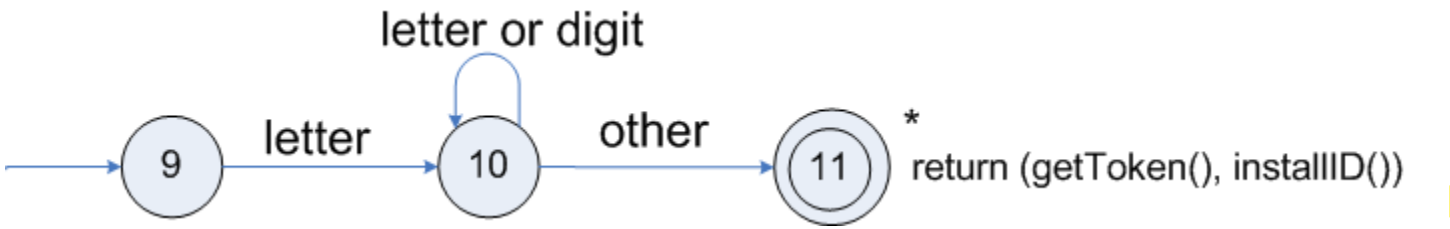
Transition diagrams

- Transition diagram for relop



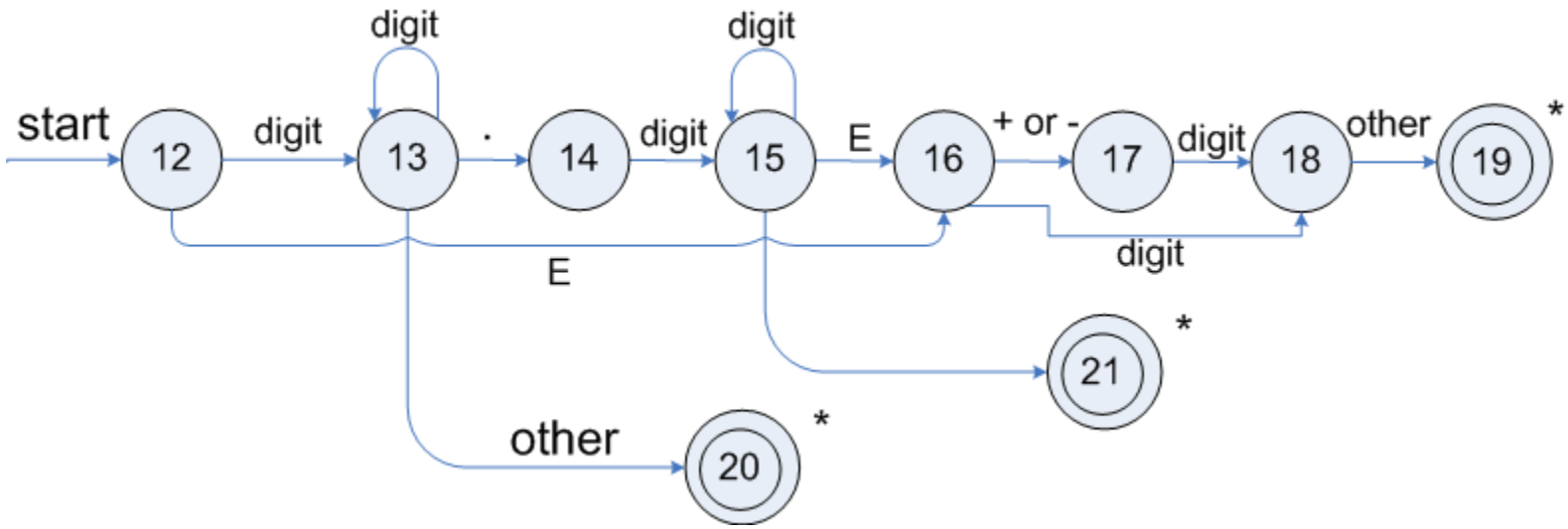
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



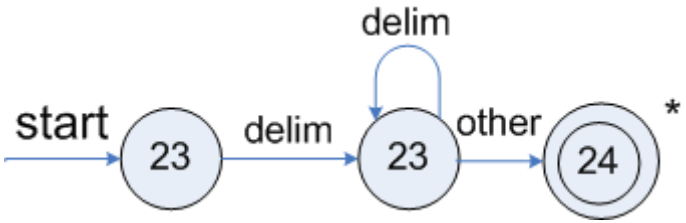
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

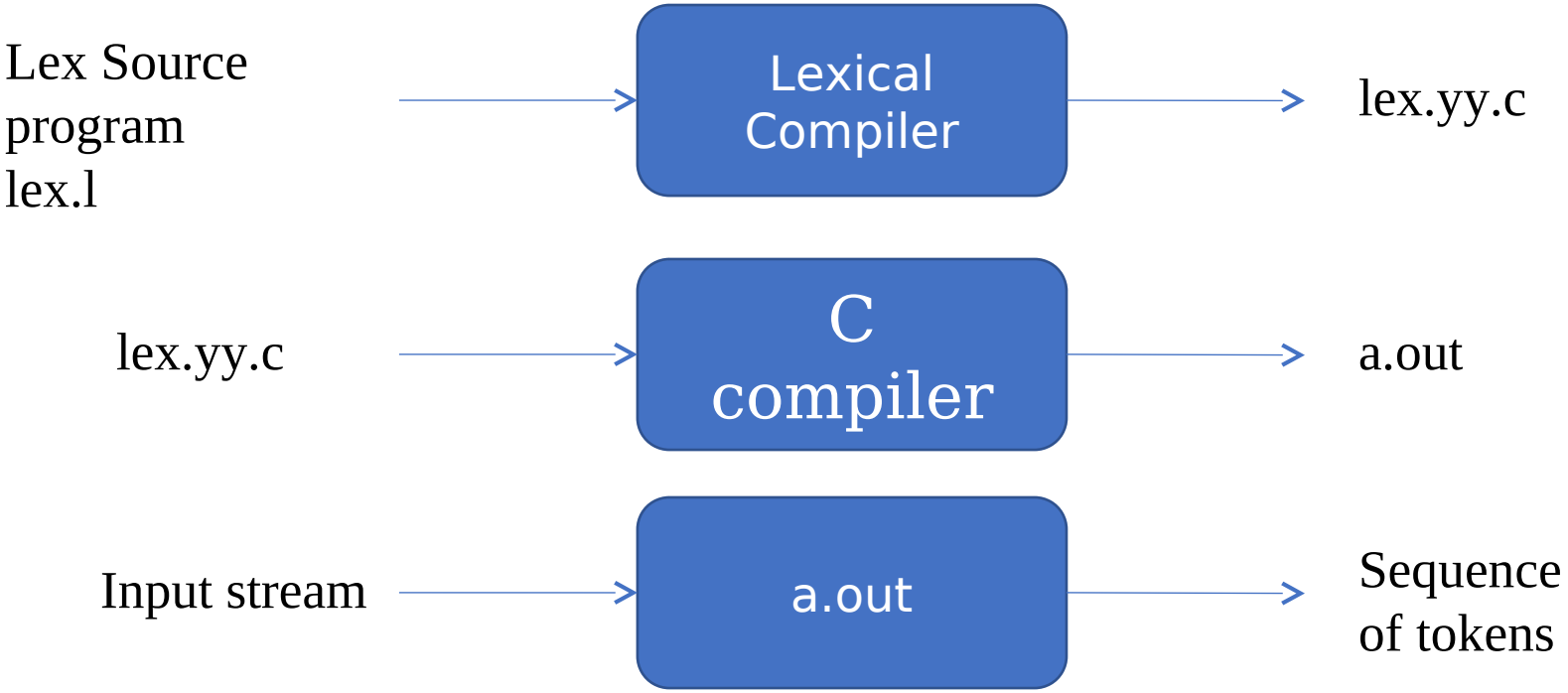
- Transition diagram for whitespace



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);
  }
```

Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules
%%
auxiliary functions

Pattern {Action}

Thank You