Chapter 3

Lexical Analysis

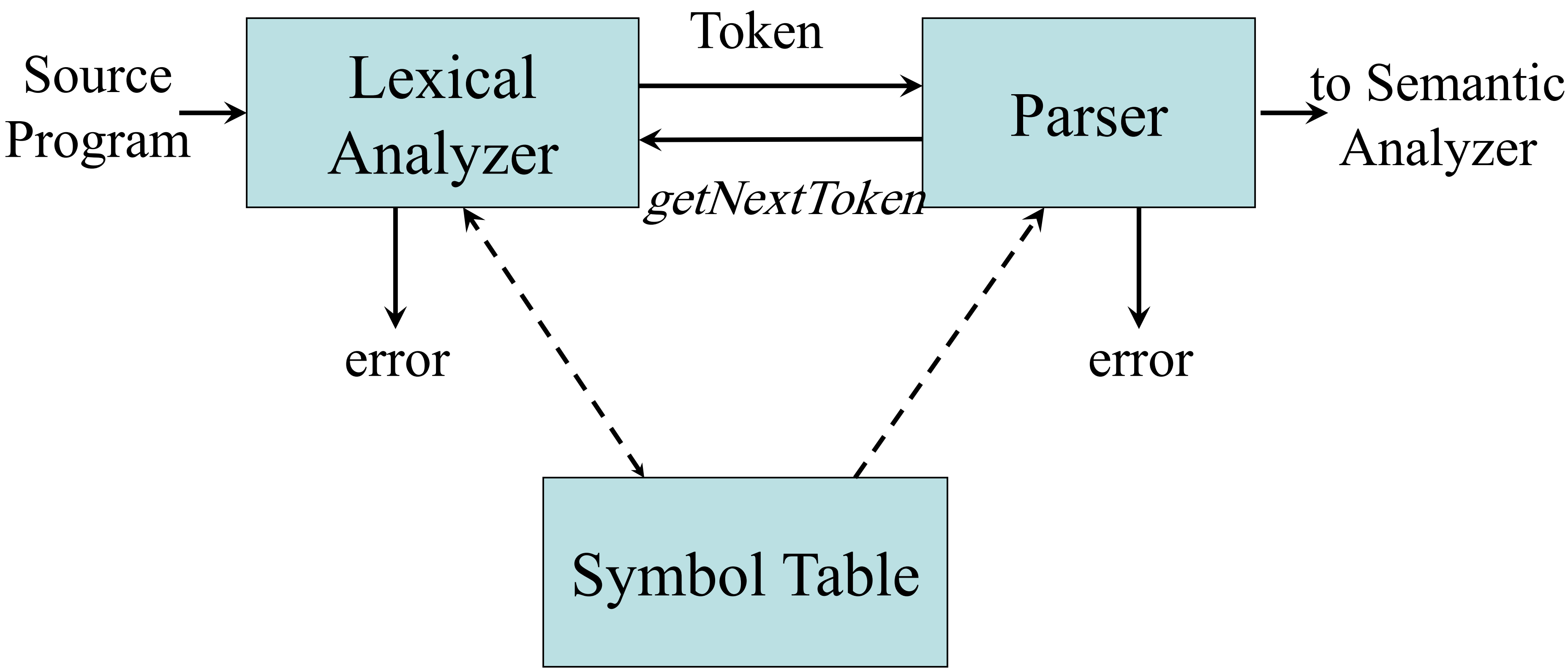
Overview

mansoori@shirazu.ac.ir

- How to construct a *lexical analyzer* by hand
  - Start with a diagram or regular expression for lexemes of each token
  - Then write code to identify the occurrence of each lexeme on the input
  - And return information about the token identified
- How to produce a *lexical analyzer* automatically
  - Specify the lexeme patterns to a lexical-analyzer generator
  - Compile those patterns into code that functions as a lexical analyzer

Role of Lexical Analyzer

mansoori@shirazu.ac.ir



Lexical Analysis as a Separate Phase

mansoori@shirazu.ac.ir

- Simplifies the design of compiler
  - A parser dealing with comments and whitespace is more complex than parser assumes comments and whitespace are removed by lexical analyzer
- Improves the efficiency of compiler
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Specialized buffering techniques for reading input characters to speed up the compiler
- Enhancing the portability of compiler
  - Input-device-specific peculiarities can be restricted to the lexical analyzer

Tokens, Patterns and Lexemes

mansoori@shirazu.ac.ir

- Token: <*token name*, optional *attribute value*>
  - Token name: abstract name representing a kind of lexical unit
    - **id**, **num**, **if**
  - Attribute value: depends on token
    - Pointer to a row of symbol table, 125, ‘if’

- Pattern: rules describing the set of lexemes belonging to a token
  - **id**: *letter followed by letters and digits*
  - **num**: *non-empty sequence of digits*

- Lexeme: a character string that matches the pattern for a token
  - **id**: x, test, a25, 3b4, b@2

Some Classes of Tokens

mansoori@shirazu.ac.ir

- One token for each keyword
- Tokens for operators: either individually or in classes
- One token for all identifiers
- One token for each constants types: numbers, literals
- Tokens for punctuation symbols: ( ) , ;

Token	Pattern (informal)	Sample lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Attributes for Tokens

mansoori@shirazu.ac.ir

Lexical analyzer returns to the parser:

1. Token name
   - Influences parsing decisions

2. Attribute value describing the lexeme represented by token
   - Influences translation of tokens after parsing

- Token: identifier
  - Token name: **id**
  - Attribute value: pointer to symbol-table entry for identifier
    - Information in symbol-table entry: its lexeme, its type, its firstly-found location

Example Attributes for Tokens

mansoori@shirazu.ac.ir

- **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**>,

  <**num**, 2>

- **fi** (**a** == **f**(**x**)) …

  <**id**, pointer to symbol-table entry for **fi**>, <**(**>,

  <**id**, pointer to symbol-table entry for **a**>, <**eq-op**>,

  <**id**, pointer to symbol-table entry for **f**>, <**(**>,

  <**id**, pointer to symbol-table entry for **x**>, <**)**>, <**)**>, …

mansoori@shirazu.ac.ir

Lexical Errors

- None of the patterns for tokens matches any prefix of the remaining input
- The simplest recovery strategy: panic mode
  - Delete successive characters from the remaining input until the lexical analyzer can find a well-formed token at the beginning of what input is left
- Other error-recovery actions:
  - 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

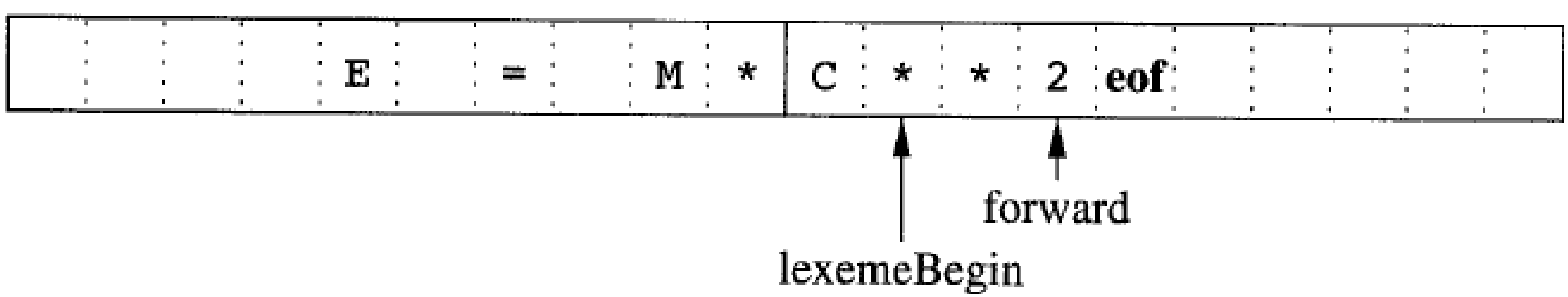
mansoori@shirazu.ac.ir

1. To speed-up the task of reading the source program
2. Lexical analyzer has to look some characters beyond the next lexeme before it can detect the right lexeme
   - In Fortran:
     - Input: DO 5 I = 1.25 → Tokens: DO5I, =, 1.25
     - Input: DO 5 I = 1,25 → Tokens: DO, 5, I, =, 1, ,, 25
   - In most programming languages, for an identifier:
     - Lexical analyzer should read characters until it sees a character that is not a letter or digit (not part of the lexeme for **id**)
   - In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=

Buffer Pairs

mansoori@shirazu.ac.ir

- Two buffers that are alternately reloaded
  - Each buffer of size N (size of a disk block, e.g., 4096 bytes)
  - Each read command reads N characters into a buffer
  - If <N characters remain in input, **eof** marks the end of file
- Two pointers to the input are maintained:
  - lexemeBegin: marks the beginning of the current token's lexeme
  - forward: scans ahead until a pattern match is found



Specification of Patterns for Tokens: Definitions

mansoori@shirazu.ac.ir

- An *alphabet* Σ is a finite set of symbols (characters)
- A *string s* is a finite sequence of symbols from Σ
  - $|s|$ denotes the length of string *s*
  - ε denotes the empty string, thus $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet Σ
  - {}, {ε}, {a, aab} are languages over Σ ={a, b}

Specification of Patterns for Tokens: *Language Operations*

mansoori@shirazu.ac.ir

- *Union*

  $$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$$

- *Concatenation*

  $$LM = \{xy \mid x \in L \text{ and } y \in M\}$$

- *Exponentiation*

  $$L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L$$

- *Kleene closure*

  $$L^* = \cup_{i=0,\ldots,\infty} L^i$$

- *Positive closure*

  $$L^+ = \cup_{i=1,\ldots,\infty} L^i$$

Specification of Patterns for Tokens: *Regular Expressions*

mansoori@shirazu.ac.ir

- Basis symbols:
  - ε is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$

- If r and s are regular expressions denoting languages $L(r)$ and $L(s)$ respectively, then
  - $r \mid s$ is a regular expression denoting $L(r) \cup L(s)$
  - rs is a regular expression denoting $L(r)L(s)$
  - r^* is a regular expression denoting $L(r)^*$
  - (r) is a regular expression denoting $L(r)$

- A language defined by a regular expression is called a *regular set* (*language*)

Specification of Patterns for Tokens: *Regular Definitions*

mansoori@shirazu.ac.ir

- Regular definitions introduce a naming convention:

$$d_1 \to r_1$$
$$d_2 \to r_2$$
$$\ldots$$
$$d_n \to r_n$$

where each r_i is a regular expression over

$$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$$

- Any d_j in r_i can be textually substituted in r_i to obtain an equivalent set of definitions

Specification of Patterns for Tokens: *Regular Definitions*

mansoori@shirazu.ac.ir

- Example:

$$
\begin{aligned}
letter &\to \mathbf{A} \mid \mathbf{B} \mid \ldots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z} \\
digit &\to \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9} \\
\mathbf{id} &\to letter\,(\,letter \mid digit\,)^{*}
\end{aligned}
$$

- Regular definitions are not recursive:

$digits \to digit\ digits \mid digit$ wrong!

Specification of Patterns for Tokens: *Notational Shorthand*

mansoori@shirazu.ac.ir

- The following shorthands are often used:

$$r^+ = r\,r^*$$
$$r? = r \mid \varepsilon$$
$$[\mathrm{a\text{-}z}] = \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \ldots \mid \mathrm{z}$$

- Example:

$$digit \rightarrow [0\text{-}9]$$
$$digits \rightarrow digit^+$$
$$\mathbf{num} \rightarrow digits\,(.\ digits)?\ (\ \mathbf{E}\ [+-]?\ digits\ )?$$

Regular Definitions and Grammars

mansoori@shirazu.ac.ir

Grammar:

$$
\begin{aligned}
stmt &\to \textbf{if} \; (\; expr \;) \; stmt \\
&\mid \; \textbf{if} \; (\; expr \;) \; stmt \; \textbf{else} \; stmt \\
&\mid \; \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
expr &\to term \; \textbf{relop} \; term \\
&\mid \; term
\end{aligned}
$$

$$
\begin{aligned}
term &\to \textbf{id} \\
&\mid \; \textbf{num}
\end{aligned}
$$

Regular definition:

$$\textbf{if} \to \text{if}$$

$$\textbf{else} \to \text{else}$$

$$\textbf{relop} \to \; < \; \mid \; <= \; \mid \; <> \; \mid \; > \; \mid \; >= \; \mid \; =$$

$$\textbf{id} \to letter \; (\; letter \mid digit \;)^{*}$$

$$\textbf{num} \to digits \; (\textbf{.} \; digits)? \; (\; \textbf{E} \; [\textbf{+-}]? \; digits)?$$

Recognition of Tokens:
Example of Tokens

mansoori@shirazu.ac.ir

- How to take patterns for all tokens
- How to build a code that examines the input to find lexemes matching the patterns

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Recognition of Tokens: *Regular Definitions*

$$
\begin{aligned}
\mathit{digit} &\rightarrow [0\text{-}9] \\
\mathit{digits} &\rightarrow \mathit{digit}^{+} \\
\textbf{number} &\rightarrow \mathit{digits}\,(.\ \mathit{digits})?\ (\ \mathbf{E}\ [+\text{-}]?\ \mathit{digits}\ )? \\
\mathit{letter} &\rightarrow [A\text{-}Za\text{-}z] \\
\textbf{id} &\rightarrow \mathit{letter}\,(\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\textbf{if} &\rightarrow \text{if} \\
\textbf{then} &\rightarrow \text{then} \\
\textbf{else} &\rightarrow \text{else} \\
\textbf{relop} &\rightarrow\ < \mid <= \mid <> \mid > \mid >= \mid = \\
\mathit{delim} &\rightarrow \mathit{blank} \mid \mathit{tab} \mid \mathit{newline} \\
\textbf{ws} &\rightarrow \mathit{delim}^{+}
\end{aligned}
$$

mansoori@shirazu.ac.ir

Recognition of Tokens: *Tokens' Specification*

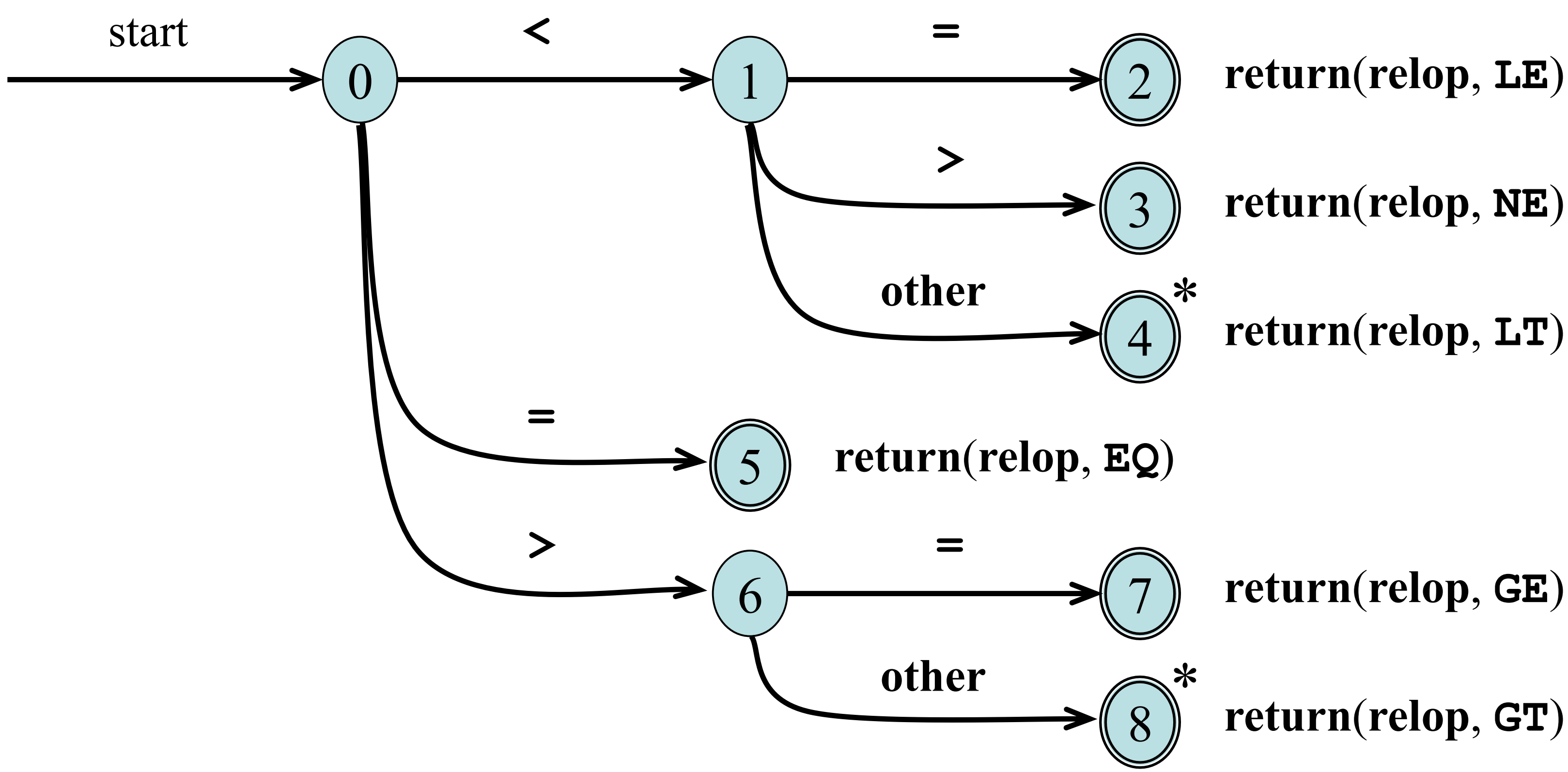
mansoori@shirazu.ac.ir

Lexemes	Token Name	Attribute Value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Recognition of **relop**: *Transition Diagram*

mansoori@shirazu.ac.ir

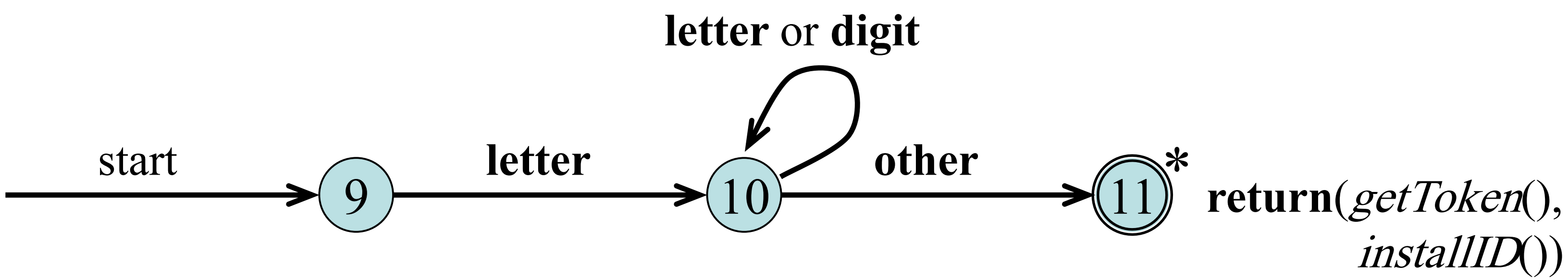
relop $\rightarrow$ $< \mid <= \mid <> \mid > \mid >= \mid =$



Recognition of **id**: *Transition Diagram*

mansoori@shirazu.ac.ir

id $\rightarrow$ *letter* (*letter* | *digit*)*



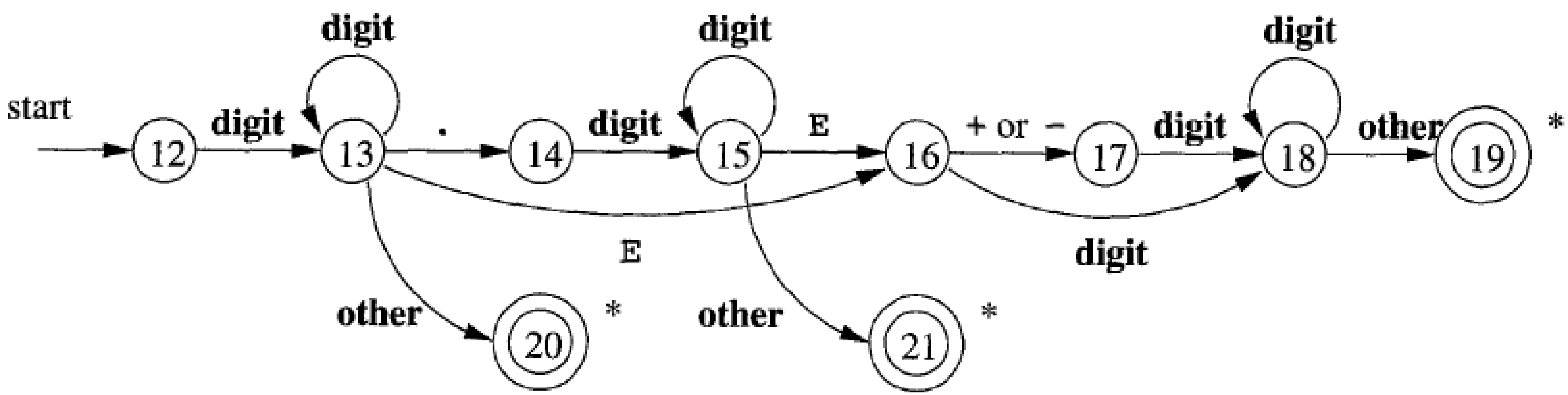
- How to handle reserved words that look like identifiers:

1. Install the reserved words in symbol table initially as non-**id**
   - *installId*: place in symbol table if new, return pointer to its entry
   - *getToken*: examine the symbol table for lexeme and return token type
2. Create separate transition diagram for each keyword

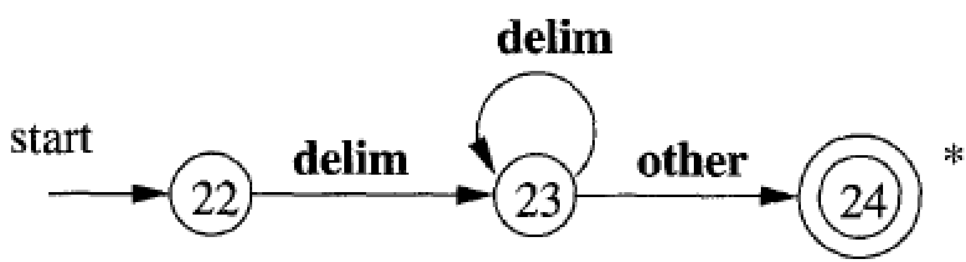
Recognition of **number** and **ws**: *Transition Diagram*

mansoori@shirazu.ac.ir

number → *digits* (. *digits*)? (**E** [+-]? *digits*)?



ws → *delim*$^+$



Recognition of Tokens: *Code*

```
state=0; lexemeBegin=0; forward=0;

Token nextToken() {
  while (1) {
    switch (state) {
     case 0: c=inpBuf[forward++];
       if (c=='<') state=1;
       else if (c=='=') state=5;
       else if (c=='>') state=6;
       else state=fail(); break;
     case 1: c=inpBuf[forward++];
       if (c=='=') state=2;
       else if (c=='>') state=3;
       else state = 4;  break;
     case 2: token retTkn=new(Relop);
       retTkn.attribute=LE;
       lexemeBegin=forward;
       return(retTkn);
     case 3: /* as 2 for NE */
     case 4: forward--;
       token retTkn=new(Relop);
       retTkn.attribute=GT;
       lexemeBegin=forward;
       return(retTkn);
     …
```

```
 …
 case 9: c=inpBuf[forward++];
   if (isletter(c)) state=10;
   else state=fail(); break;
 case 10: c=inpBuf[forward++];
   if (isletter(c) ||
        isdigit(c)) state=10;
   else state = 11;  break;
case 11: forward--;
   token retTkn=new(Id);
   lexeme=inpBuf[lexemeBegin:forward];
   retTkn.attribute=installId(lexeme);
   retTkn.name=getToken(lexeme);
   lexemeBegin = forward;
   return(retTkn);
```

```
int fail() {
  forward=lexemeBegin;
  switch (state) {
   case 0: state=9; break;
   case 9: state=12; break;
   case 12: state=22; break;
   case 22: error_recover(); break;
   default: /* error */
  } return state;
}
```

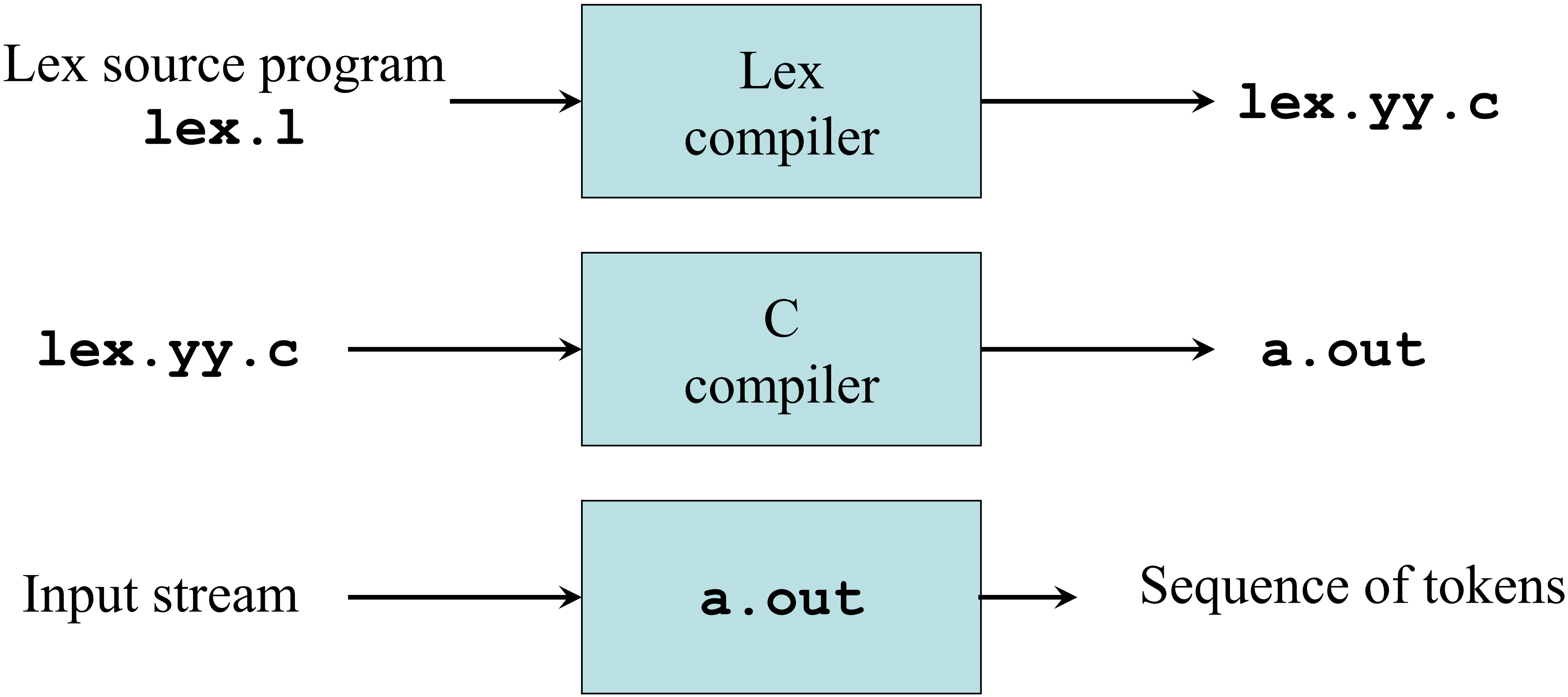
Lexical-Analyzer Generator: *Lex* and *Flex*

mansoori@shirazu.ac.ir

- *Lex* and its newer cousin *Flex* are lexical-analyzer generators
- Translate regular definitions into C source code for efficient lexical analysis
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with *Lex*

mansoori@shirazu.ac.ir



Lex Specification

mansoori@shirazu.ac.ir

- A *Lex specification* consists of three parts:

  Regular definitions, C declarations in `%{ %}`
  `%%`
  Translation rules
  `%%`
  User-defined *auxiliary procedures*

- The *translation rules* are of the form:

  $p_1 \quad \{ \ action_1 \ \}$
  $p_2 \quad \{ \ action_2 \ \}$
  …
  $p_n \quad \{ \ action_n \ \}$

mansoori@shirazu.ac.ir

Regular Expressions in *Lex*

x	match the character **x**
\\.	match the character **.**
"*string*"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
r*****	closure (match zero or more occurrences)
r**+**	positive closure (match one or more occurrences)
r**?**	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
$r_1 \mid r_2$	match r_1 or r_2 (union)
(r **)**	grouping
$r_1 \backslash r_2$	match r_1 when followed by r_2
{ d **}**	match the regular expression defined by d

Lex Specification: Example 1

mansoori@shirazu.ac.ir

Contains the matching lexeme

Translation rules

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    { }
%%
main() {
  yylex();
}
```

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll ./a.out < spec.l
```

Lex Specification: Example 2

mansoori@shirazu.ac.ir

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim       [ \t]+
%%
\n          { ch++; wd++; nl++; }
^{delim}    { ch+=yyleng; }
{delim}     { ch+=yyleng; wd++; }
.           { ch++; }
%%
main() {
  yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Lex Specification: Example 3

m
a
n
s
o
o
r
i @shirazu.ac.ir

Regular definition

Translation rules

```
%{
#include <stdio.h>
%}
digit       [0-9]
letter      [A-Za-z]
id          {letter}({letter}|{digit})*
%%
{digit}+    { printf("number: %s\n", yytext); }
{id}        { printf("ident: %s\n", yytext); }
.           { printf("other: %s\n", yytext); }
%%
main() {
  yylex();
}
```

Lex Specification: Example 4

mansoori@shirazu.ac.ir

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}        { }
if          {return IF;}
then        {return THEN;}
else        {return ELSE;}
{id}        {yylval = install_id(); return ID;}
{number}    {yylval = install_num(); return NUMBER;}
"<"         {yylval = LT; return RELOP;}
"<="        {yylval = LE; return RELOP;}
"="         {yylval = EQ; return RELOP;}
"<>"        {yylval = NE; return RELOP;}
%%
int install_id() { … }
…
```

Return token to parser

Token attribute

Install **`yytext`** as identifier in symbol table