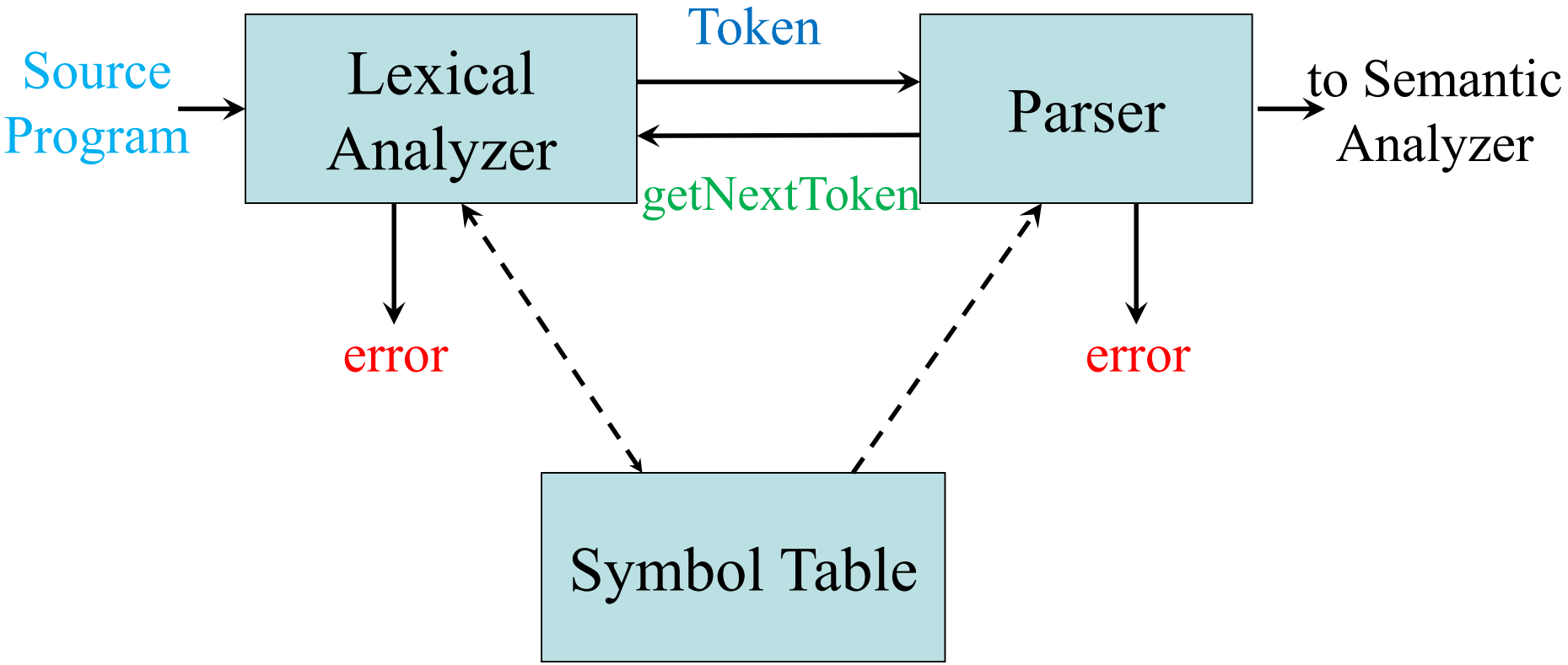
Chapter 3

Lexical Analysis

Overview

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

Role of Lexical Analyzer



Lexical Analysis as a Separate Phase

- Simplifies design of compiler
  - A parser dealing (Moameleh) with comments and whitespace is more complex than parser assumes comments and whitespace are removed by lexical analyzer
- Improves 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 compiler
- Enhancing (Taqviat konandeh) portability of compiler
  - Input-device-specific peculiarities (Vizhegi haye khas) can be restricted (Mahdod shodeh ast) to lexical analyzer

Tokens, Patterns and Lexemes

- 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 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 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: ( ) , ;

Noqte gozari

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

Lexical analyzer returns to the parser:

1. Token name
   - Influences parsing decisions

2. Attribute value describing 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

- 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-rel>,
  <id, pointer to symbol-table entry for f>, <(>,
  <id, pointer to symbol-table entry for x>, <)>, <)>, …

Lexical Errors

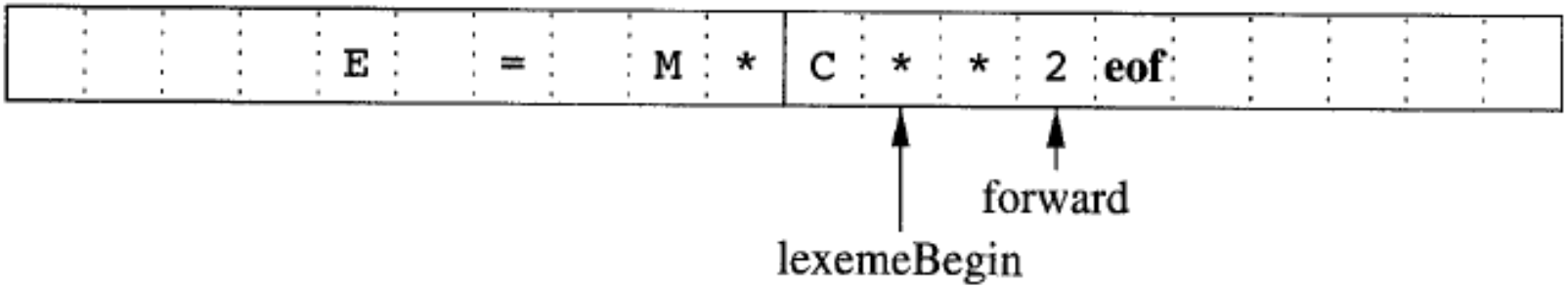
- None of patterns for tokens matches any prefix of remaining input
- The simplest recovery strategy: panic mode
  - Delete successive characters from remaining input until lexical analyzer can find a well-formed token at beginning of what input is left
- Other error-recovery actions:
  - Delete one character from remaining input
  - Insert a missing character into remaining input
  - Replace a character by another character
  - Transpose two adjacent characters

Input Buffering

1. To speed-up task of reading source program
2. Lexical analyzer has to look some characters beyond next lexeme before it can detect 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 lexeme for id)
   - In C, single-character operators like -, =, or < could also be beginning of a two-character operator like ->, ==, or <=

Buffer Pairs

- 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 end of file
- Two pointers to input are maintained:
  - lexemeBegin: marks beginning of current token's lexeme
  - forward: scans ahead until a pattern match is found



Specification of Patterns for Tokens: Definitions

- An alphabet Σ is a finite set of symbols (characters)

- A string s is a finite sequence of symbols from Σ
  - $|s|$ denotes length of string s
  - ε denotes 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

- 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

- 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

- Regular definitions introduce a naming convention:

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  …
  $d_n \rightarrow r_n$

  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

Alternated ....

Specification of Patterns for Tokens: Regular Definitions

- Example:

letter → A | B | … | Z | a | b | … | z
digit → 0 | 1 | … | 9
id → letter (letter | digit)*

- Regular definitions are not recursive:

digits → *digit digits* | *digit* wrong!

Specification of Patterns for Tokens: Notational Shorthand

- Following shorthands are often used:

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

- Example:

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

Regular Definitions and Grammars

mansoori@shirazu.ac.ir

Grammar:

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number

Regular definition:

if → if

else → else

relop → < | <= | <> | > | >= | =

id → letter (letter | digit)*

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

Recognition of Tokens:

Token Examples

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

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number

Recognition of Tokens: Regular Definitions

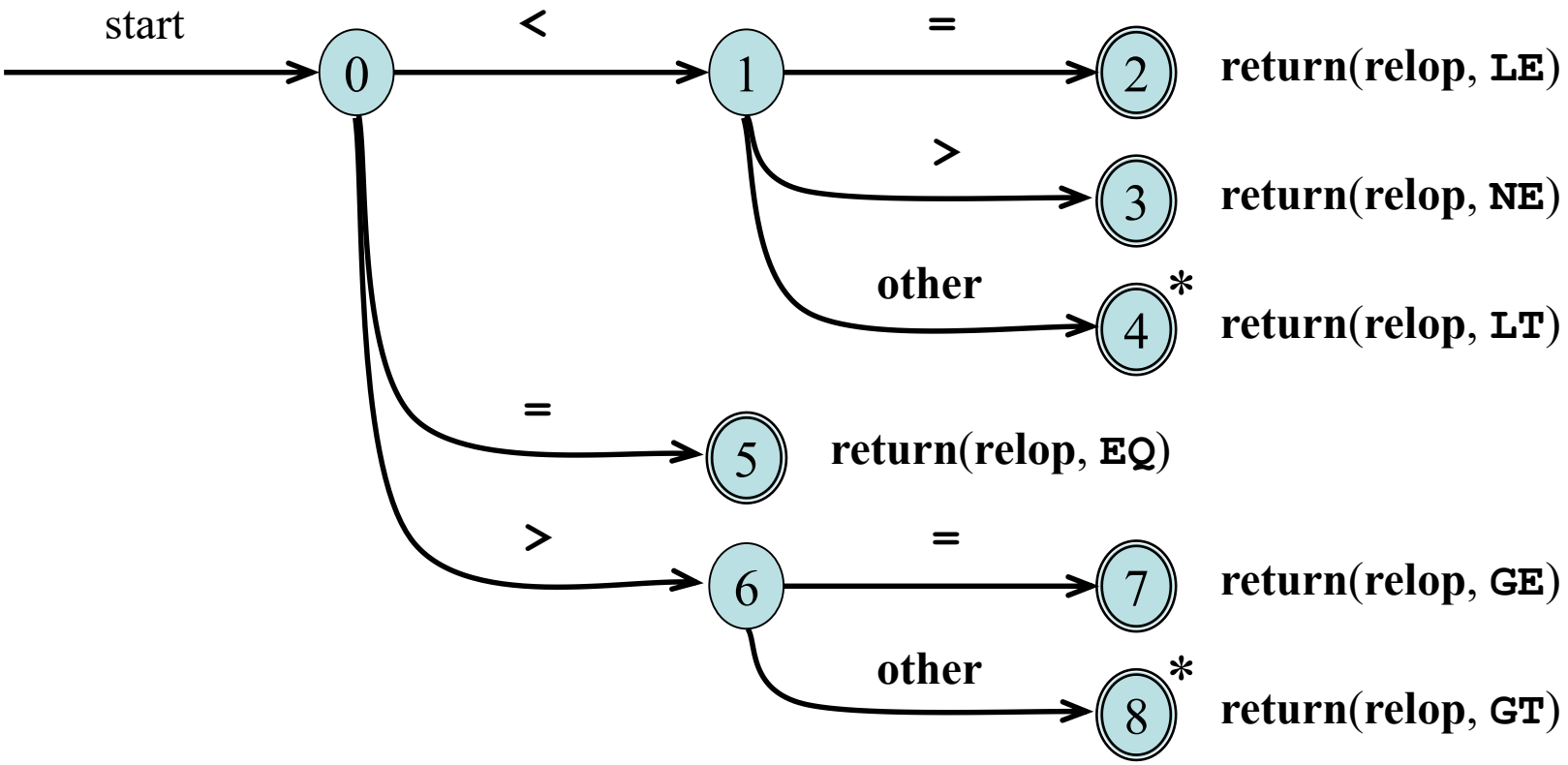
$$
\begin{aligned}
\text{digit} &\rightarrow [0\text{-}9] \\
\text{digits} &\rightarrow \text{digit}^{+} \\
\text{number} &\rightarrow \text{digits}\ (.\ \text{digits})?\ (\ \text{E}\ [+\text{-}]?\ \text{digits}\ )? \\
\text{letter} &\rightarrow [\text{A-Za-z}] \\
\text{id} &\rightarrow \text{letter}\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

Recognition of Tokens: Tokens Specification

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

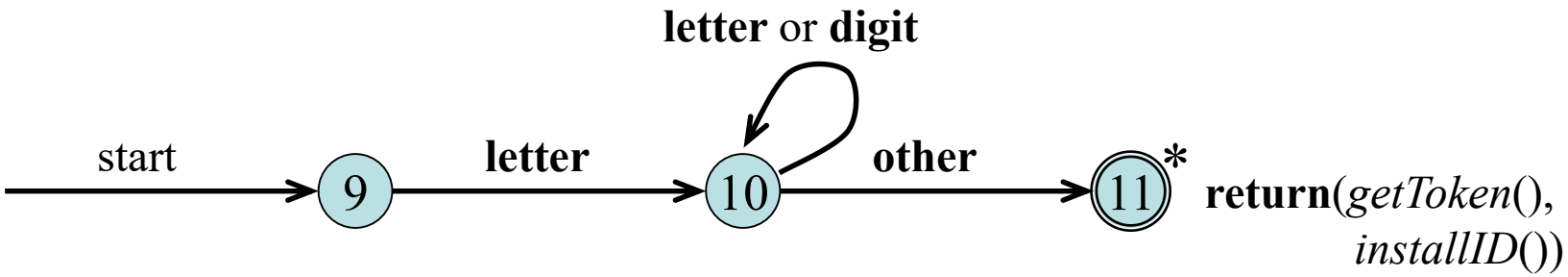
Recognition of **relop**: Transition Diagram

relop → < | <= | <> | > | >= | =



Recognition of **id**: Transition Diagram

id → letter (letter | digit)*

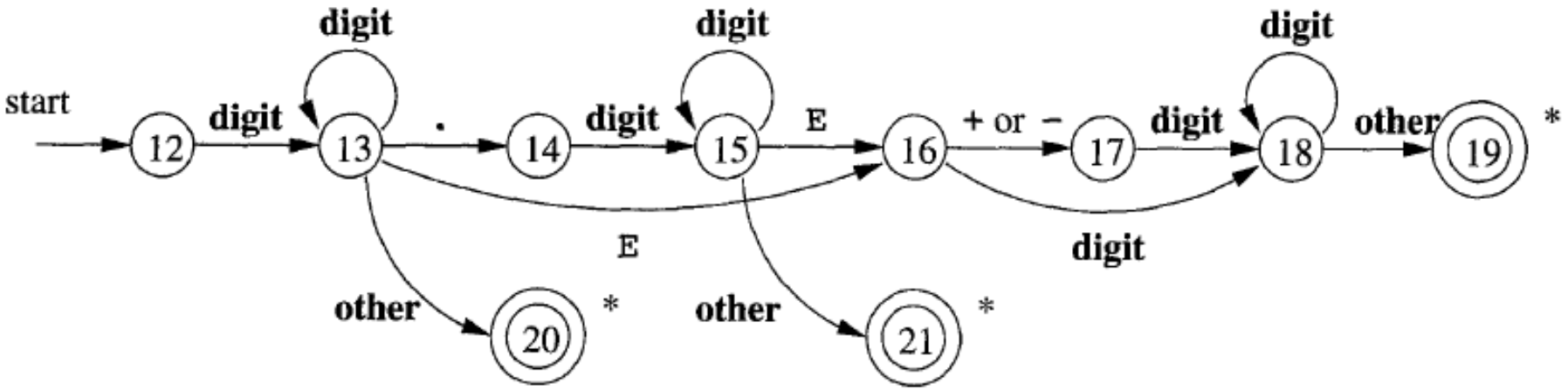


- How to handle reserved words that look like identifiers:

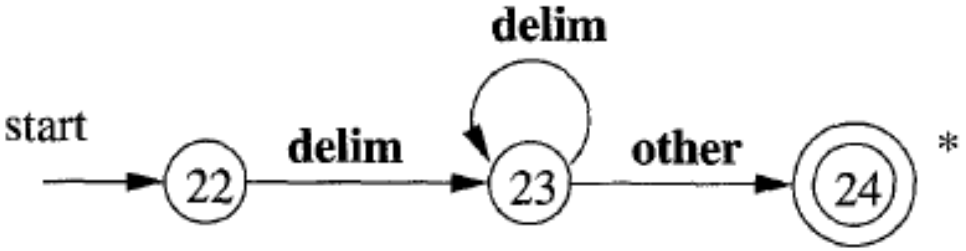
1. Install reserved words in symbol table initially as non-id
   - installId: place in symbol table if new, return pointer to its entry
   - getToken: examine symbol table for lexeme and return token type
2. Create separate transition diagram for each keyword

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

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 Token(Relop);
       retTkn.attribute=LE;
       lexemeBegin=forward;
       return(retTkn);

     case 3: /* as 2 for NE */

     case 4: forward--;
       Token retTkn=new Token(Relop);
       retTkn.attribute=GT;
       lexemeBegin=forward;
       return(retTkn);
     …
```

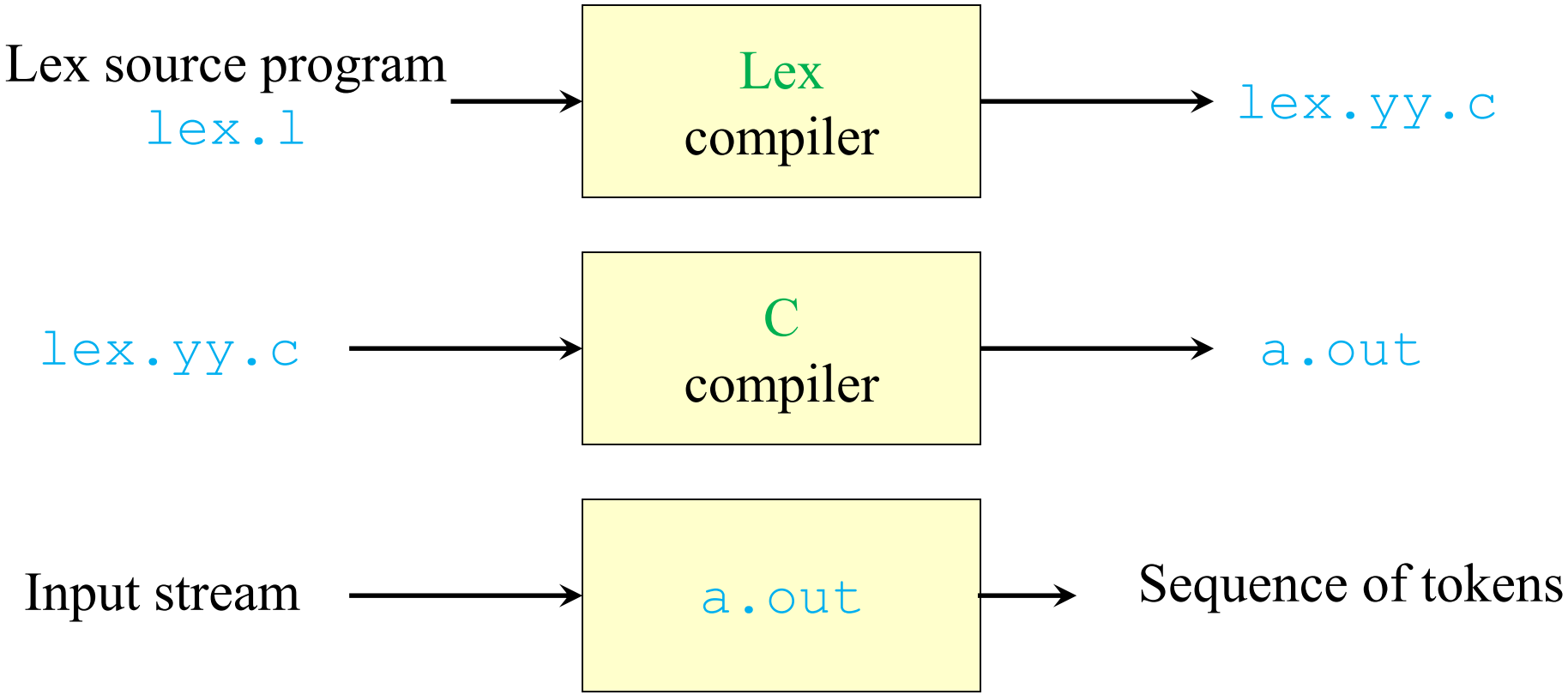
```
     case 9: c=inpBuf[forward++];
       if (isletter(c)) state=10;
       else state=fail(); break;

     case 10: c=inpBuf[forward++];
       if (isletter(c) ||
           isdigit(c)) state=10;
       else state = 11;  break;

     case 11: forward--;
       Token retTkn=new Token(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

- 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



Lex Specification

- A Lex specification consists of three parts:

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

- Translation rules are of form:

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

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

Translation rules

Contains the matching lexeme

```
%{
#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

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


```
%{
#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();
}
```

Regular definition

Translation rules

Lex Specification: Example 4

```
%{ /* 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