NEW YORK UNIVERSITY

CSCI-GA.2130-001

Compiler Construction

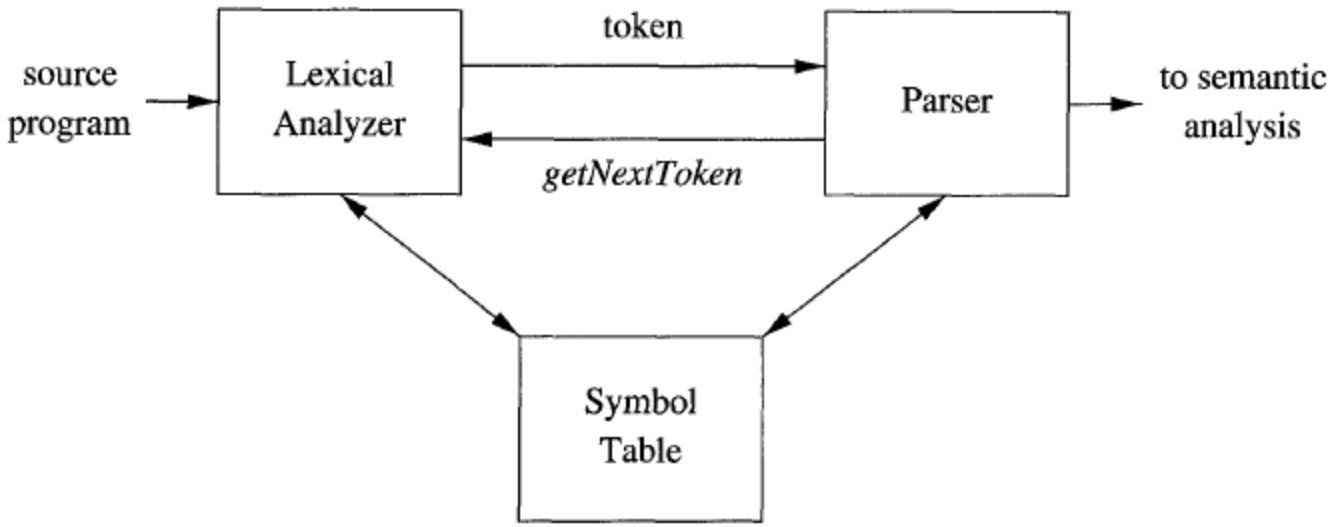
Lecture 4: Lexical Analysis I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu

Role of the Lexical Analyzer

- Remove comments and white spaces (aka *scanning*)
- Macros expansion
- Read input characters from the source program
- Group them into lexemes
- Produce as output a sequence of tokens
- Interact with the symbol table
- Correlate error messages generated by the compiler with the source program

Scanner-Parser Interaction



Why Separating Lexical and Syntactic?

- Simplicity of design
- Improved compiler efficiency
  - allows us to use specialized technique for lexer, not suitable for parser
- Higher portability
  - Input-device-specific peculiarities restricted to lexer

Some Definitions

- Token: a pair consisting of
  - Token name: abstract symbol representing lexical unit [affects parsing decision]
  - Optional attribute value [influences translations after parsing]
- Pattern: a description of the form that different lexemes take
- Lexeme: sequence of characters in source program matching a pattern

Pattern

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 letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Token classes

- One token per keyword
- Tokens for the operators
- One token representing all identifiers
- Tokens representing constants (e.g. numbers)
- Tokens for punctuation symbols

Example

```
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>

Dealing With Errors

Lexical analyzer unable to proceed: no pattern matches

- Panic mode recovery: delete successive characters from remaining input until token found
- Insert missing character
- Delete a character
- Replace character by another
- Transpose two adjacent characters

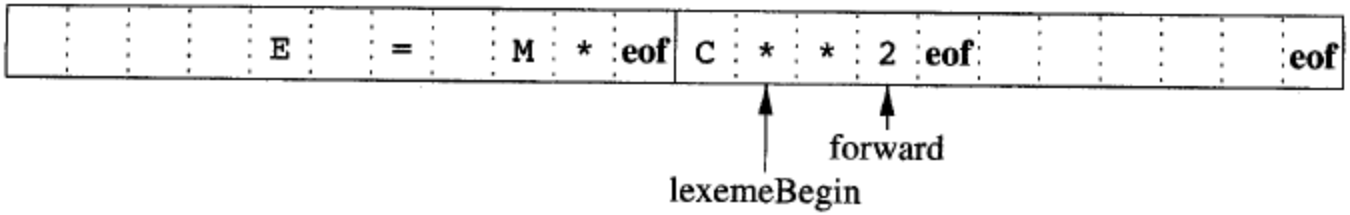
Example

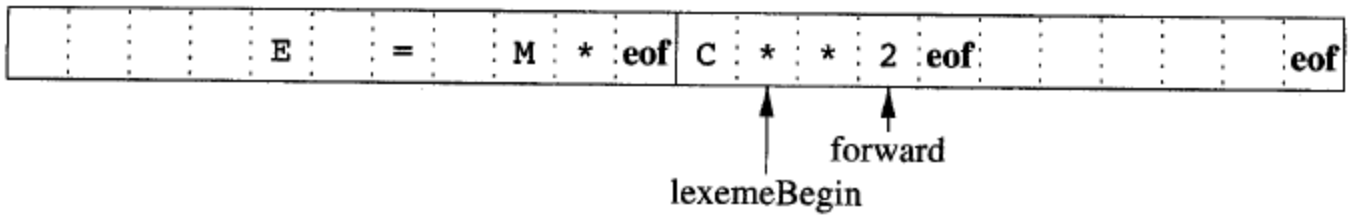
```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

What tokens will be generated from the above C++ program?

Buffering Issue

- Lexical analyzer may need to look at least a character ahead to make a token decision.
- Buffering: to reduce overhead required to process a single character






```
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
}
```

Tokens Specification

- We need a *formal* way to specify patterns: regular expressions
- Alphabet: any finite set of symbols
- String over alphabet: finite sequence of symbols drawn from that alphabet
- Language: countable set of strings over some fixed alphabet

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

Operations

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Zero or one instance: r? is equivalent to r|ε

Character class: a|b|c|…|z can be replaced by [a-z]
a|c|d|h can be replaced by [acdh]

Examples

Which language is generated by:

- (a|b)(a|b)
- a^*
- $(a|b)^*$
- $a|a^*b$

Example

How can we present number that can be integer with option floating point and exponential parts?

Examples

Write regular definition of all strings of lowercase letters in which the letters are in ascending order

Tokens Recognition

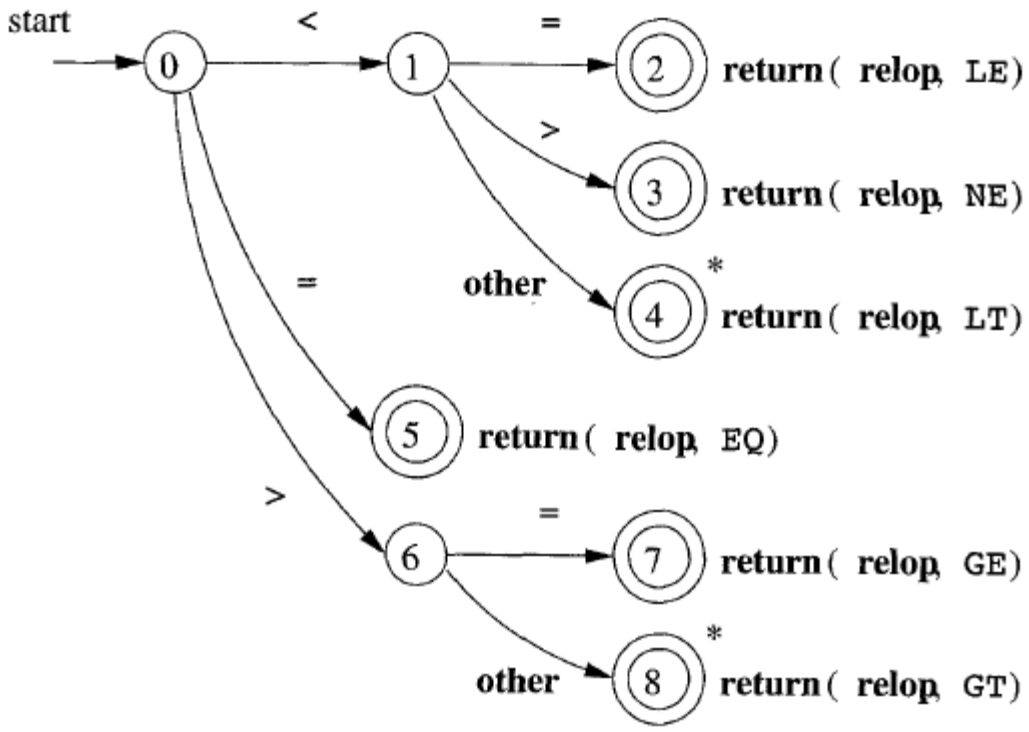
$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}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

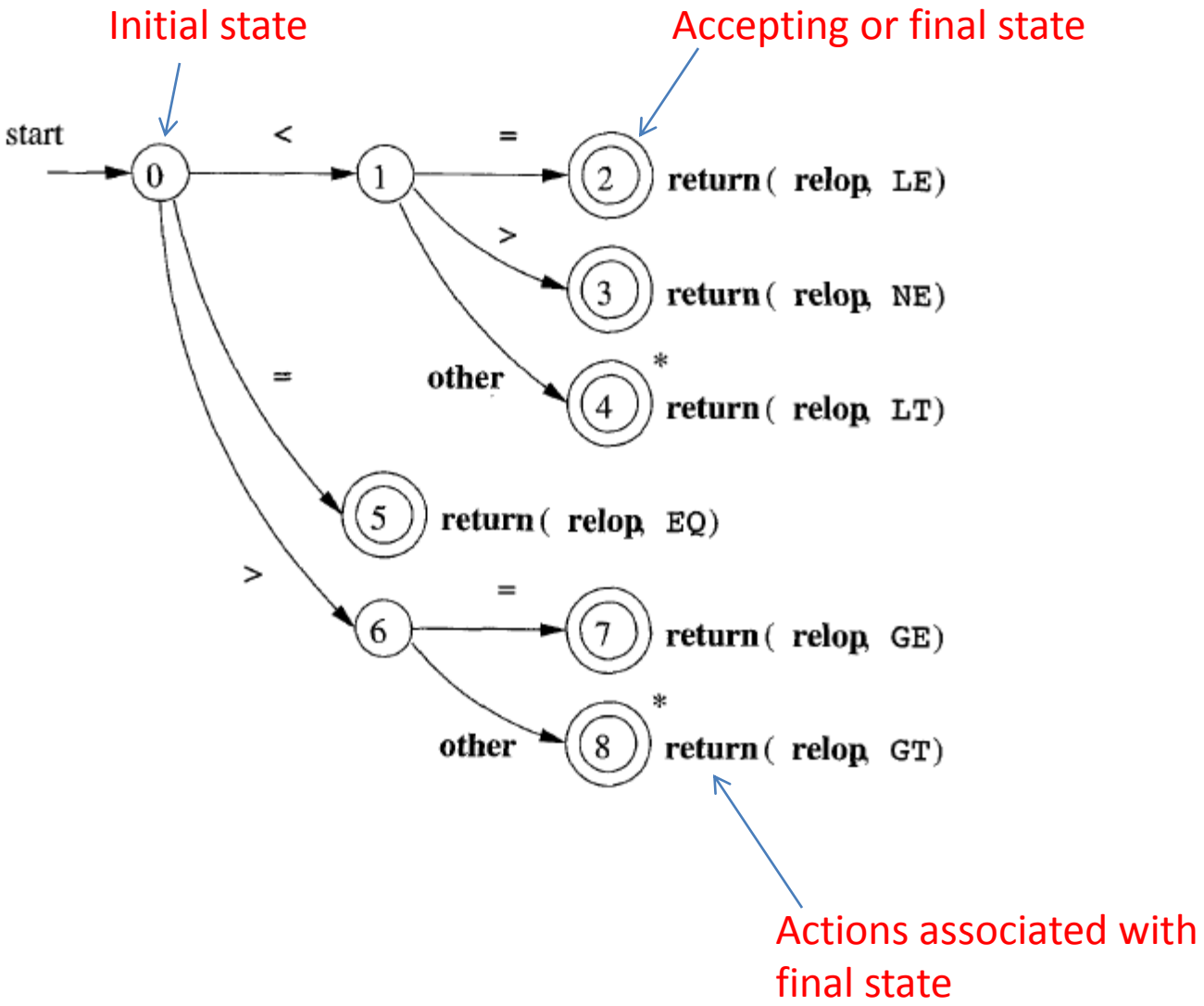
$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits \ (\texttt{.} \ digits)? \ (\ \texttt{E} \ \texttt{[+-]}? \ digits \ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter \ (\ letter \mid digit \ )^{*}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

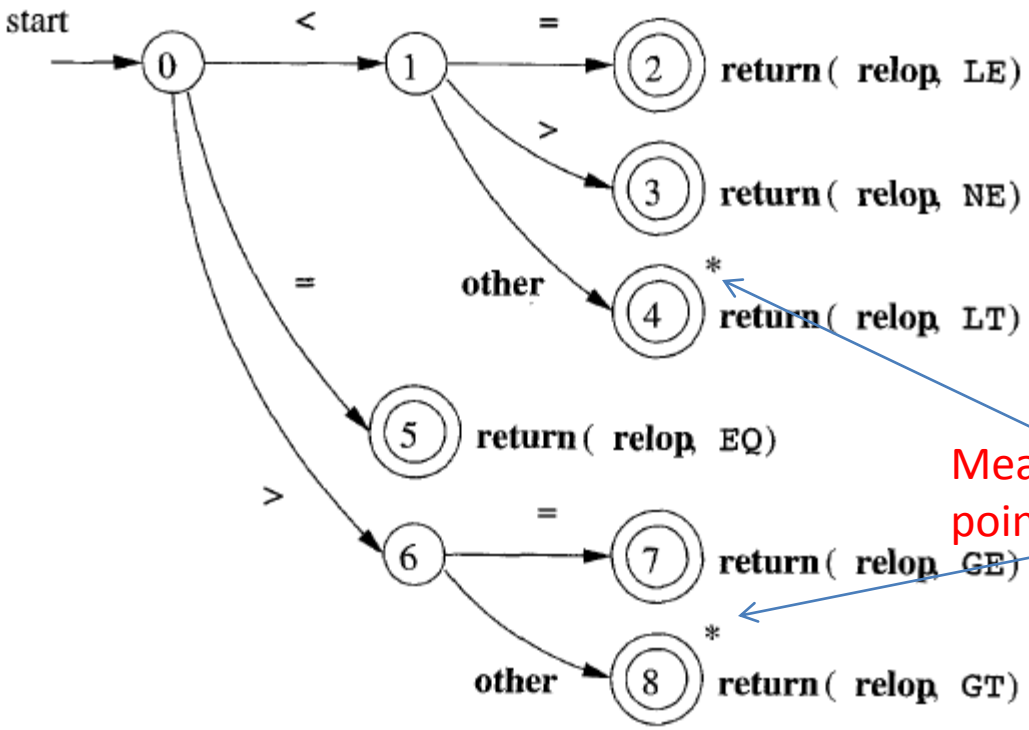
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Implementation: Transition Diagrams

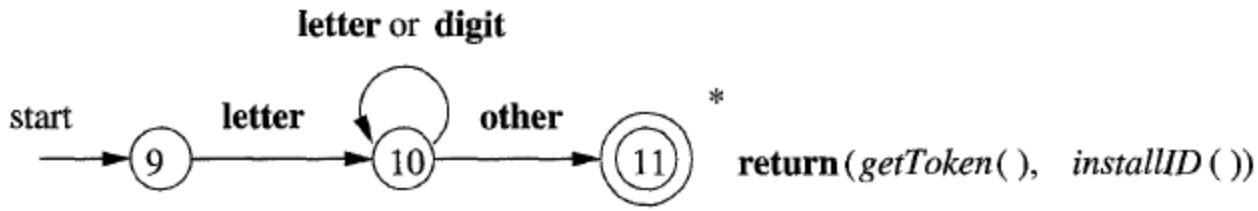
- Intermediate step in constructing lexical analyzer
- Convert patterns into flowcharts called transition diagrams
  - nodes or circles: called states
  - Edges: directed from state to another, labeled by symbols


start
0
<
1
=
2
return(relop, LE)
>
3
return(relop, NE)
other
*
4
return(relop, LT)
=
5
return(relop, EQ)
>
6
=
7
return(relop, GE)
other
*
8
return(relop, GT)

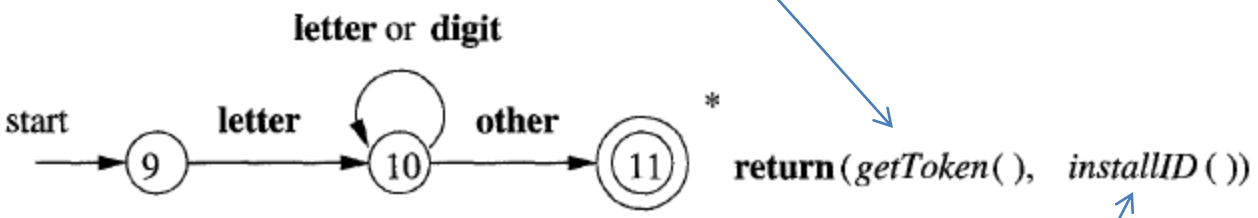

Initial state
Accepting or final state
start
0
<
1
=
2
return(relop, LE)
>
3
return(relop, NE)
other
*
4
return(relop, LT)
=
5
return(relop, EQ)
>
6
=
7
return(relop, GE)
other
*
8
return(relop, GT)
Actions associated with final state



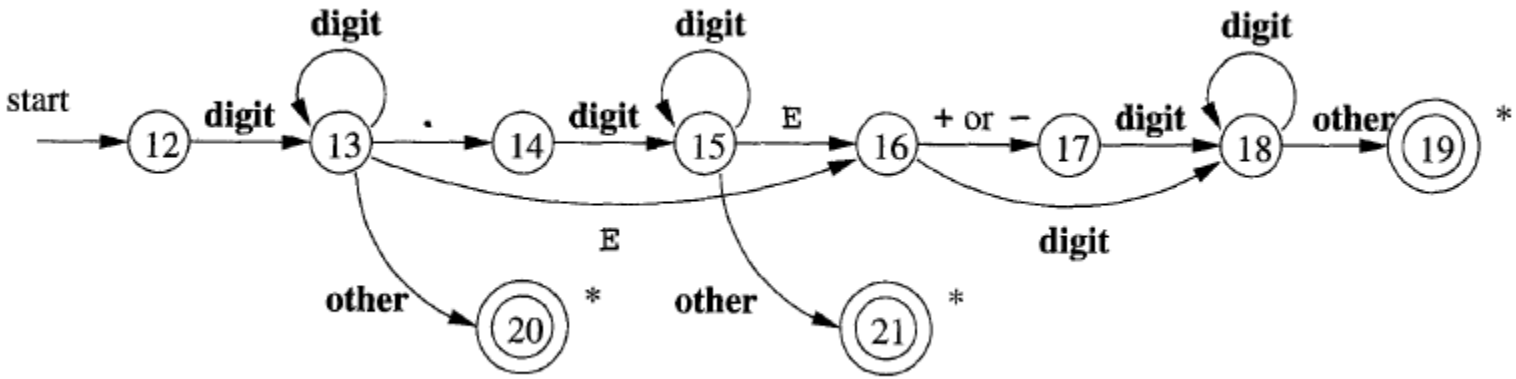
Means retract the forward pointer


letter or digit
start
letter
other
9
10
11
*
return (getToken( ), installID( ))

- Examine symbol table for the lexeme found
- Returns whatever token name is there



- Places ID in symbol table if not there.
- Returns a pointer to symbol table entry


start
12
digit
13
digit
.
14
digit
15
digit
E
16
+ or −
17
digit
18
digit
other
19
*
E
digit
other
20
*
other
21
*

Reserved Words and Identifiers

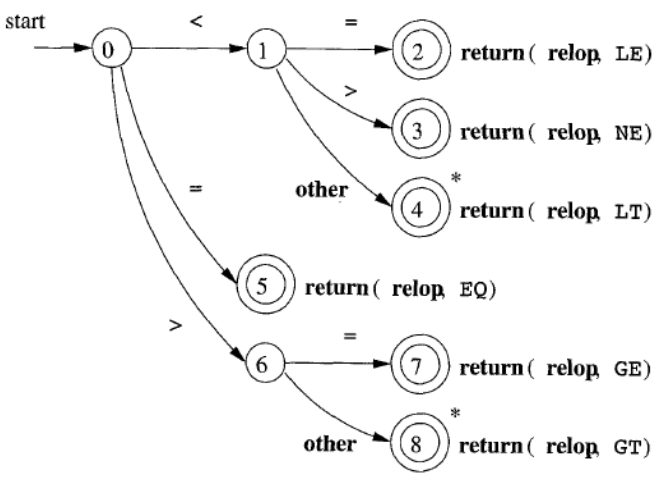
- Install reserved words in symbol table initially

OR

- Create transition diagram for each keyword, then prioritize the tokens so that keywords have higher preference

Implementation of Transition Diagram

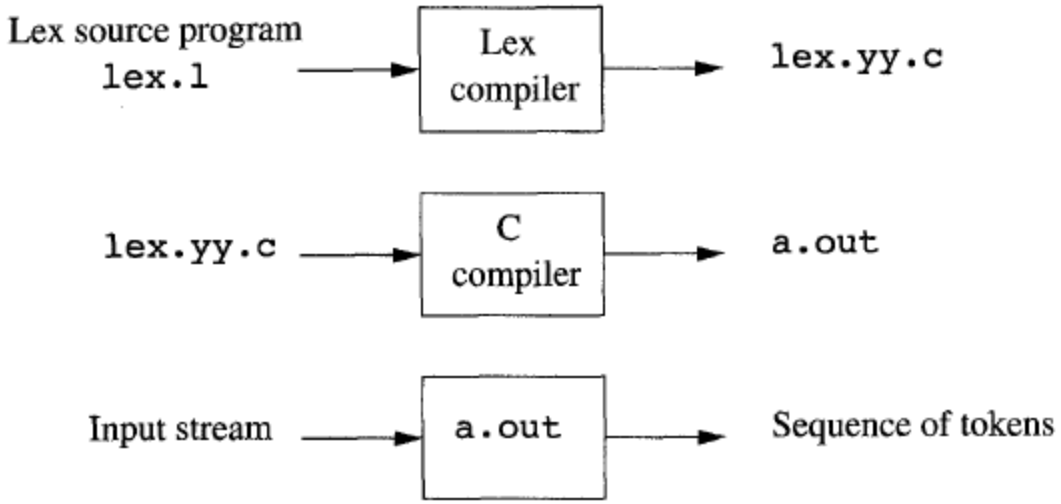
```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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



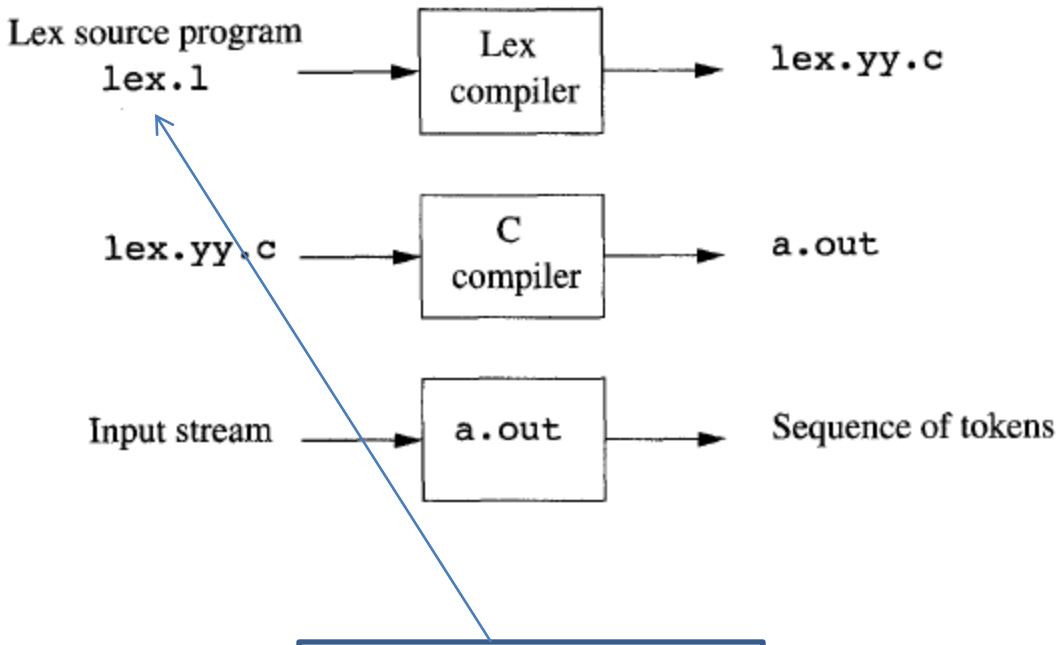
Using All Transition Diagrams: The Big Picture

- Arrange for the transition diagrams for each token to be tried sequentially
- Run transition diagrams in parallel
- Combine all transition diagrams into one

The First Part of the Project



The First Part of the Project



declarations
%%
translation rules
%%
auxiliary functions

declarations

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

translation rules

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
```

auxiliary functions

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Anything between these 2 marks is copied as it is in lex.yy.c

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

braces means the pattern is defined somewhere

pattern

Actions

Lecture of Today

- Sections 3.1 to 3.5
- First part of the project assigned