Lexical Analysis

- *lexeme:* a sequence of characters in the src pgm
- *token:* atomic unit of parsing
  - there can be different lexemes for same token
- *Pattern:* set of rules applied to lexemes to yield token

Lexeme vs Token

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<`, `<=`, `=`, `<>`, `>`, `>=`	`<` or `<=` or `=` or `<>` or `>=` or `>`
id	`pi`, `count`, `D2`	letter followed by letters and digits
num	`3.1416`, `0`, `6.02E23`	any numeric constant
literal	`"core dumped"`	any characters between " and " except "

Atributes of a Token

- Lexical Analyzer (LA) also collects info abt tokens
- Tokens influence parsing decisions
- Attributes influence translation of tokens
- *Token* has *attribute* – a *pointer to symbol table entry*
- For an identifier, some important *attr*
  - Lexeme,
  - line no. of its occurrences

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, >**
- <**num,** integer value 2>
-

Lexical Errors

- LA has very localized view of a src pgm

  `fi ( a == f (x) ) ….`
- fi is misspelling or undeclared identifier?
- “panic mode” recovery:

  - delete all chars until a well-formed one

  - sometimes confusing
-

Confusions for LA

In Fortran 77 or Fortran 90 statement

DO 5 I = 1.25

whether first lexeme is D05I?

OR

DO 5 I = 1,25

first lexeme is keyword DO here.

Buffering

- Switch (*forward++)[

```
case eof :
if (forward is at end of first buffer){
  reload second half of buff;
  fwd = begnning of sec half
  }
  else if (forward is at end of second buffer){
  reload first half of buff;
  fwd = begnning of 1st half
  }
  else /*terminate LA */
  break;
  cases for other chars
  }
```

Specification of Tokens

- Strings and Languages
  - alphabets
  - empty string
  - empty language

Terms for different parts of a string

TERM	DEFINITION
prefix of *s*	A string obtained by removing zero or more trailing symbols of string *s*; e.g., ban is a prefix of banana.
suffix of *s*	A string formed by deleting zero or more of the leading symbols of *s*; e.g., nana is a suffix of banana.
substring of *s*	A string obtained by deleting a prefix and a suffix from *s*; e.g., nan is a substring of banana. Every prefix and every suffix of *s* is a substring of *s*, but not every substring of *s* is a prefix or a suffix of *s*. For every string *s*, both *s* and ϵ are prefixes, suffixes, and substrings of *s*.
proper prefix, suffix, or substring of *s*	Any nonempty string *x* that is, respectively, a prefix, suffix, or substring of *s* such that $s \neq x$.
subsequence of *s*	Any string formed by deleting zero or more not necessarily contiguous symbols from *s*; e.g., baaa is a subsequence of banana.

Regular Expression

- Each symbol in the alphabet set is a RE
- Operations on regular expressions generate RE

  - union

  - concatenation

  - closures
- RE *r* denotes a language *L(r)*

RE: Algebraic properties

AXIOM	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r\|\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Regular Definitions

- A sequence of definitions of the form

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

- d_i is a distinct name
  r_i is a RE over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Examples

- 5280, 0.01234, 6 . 336E4, or 1.894E-4 etc.

Soln.

$$\begin{aligned}
\mathbf{digit} &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\mathbf{digits} &\rightarrow \mathbf{digit}\ \mathbf{digit}^* \\
\mathbf{optional_fraction} &\rightarrow .\ \mathbf{digits} \mid \epsilon \\
\mathbf{optional_exponent} &\rightarrow (E (+ \mid - \mid \epsilon)\ \mathbf{digits}) \mid \epsilon \\
\mathbf{num} &\rightarrow \mathbf{digits}\ \ \mathbf{optional_fraction}\ \ \mathbf{optional_exponent}
\end{aligned}$$

Notational Shorthands

- One or more instances: *r*+
- Zero or one instance: *r?*
- Character classes: *a*|*b*|*c or [abc]*
-

Soln using shorthand

$$
\begin{aligned}
\mathbf{digit} &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\mathbf{digits} &\rightarrow \mathbf{digit}^{+} \\
\mathbf{optionalFraction} &\rightarrow (\ . \ \mathbf{digits} \ )? \\
\mathbf{optionalExponent} &\rightarrow (\ \mathbf{E} \ (\ + \mid - \ )? \ \mathbf{digits} \ )? \\
\mathbf{num} &\rightarrow \mathbf{digits} \ \ \mathbf{optionalFraction} \ \ \mathbf{optionalExponent}
\end{aligned}
$$

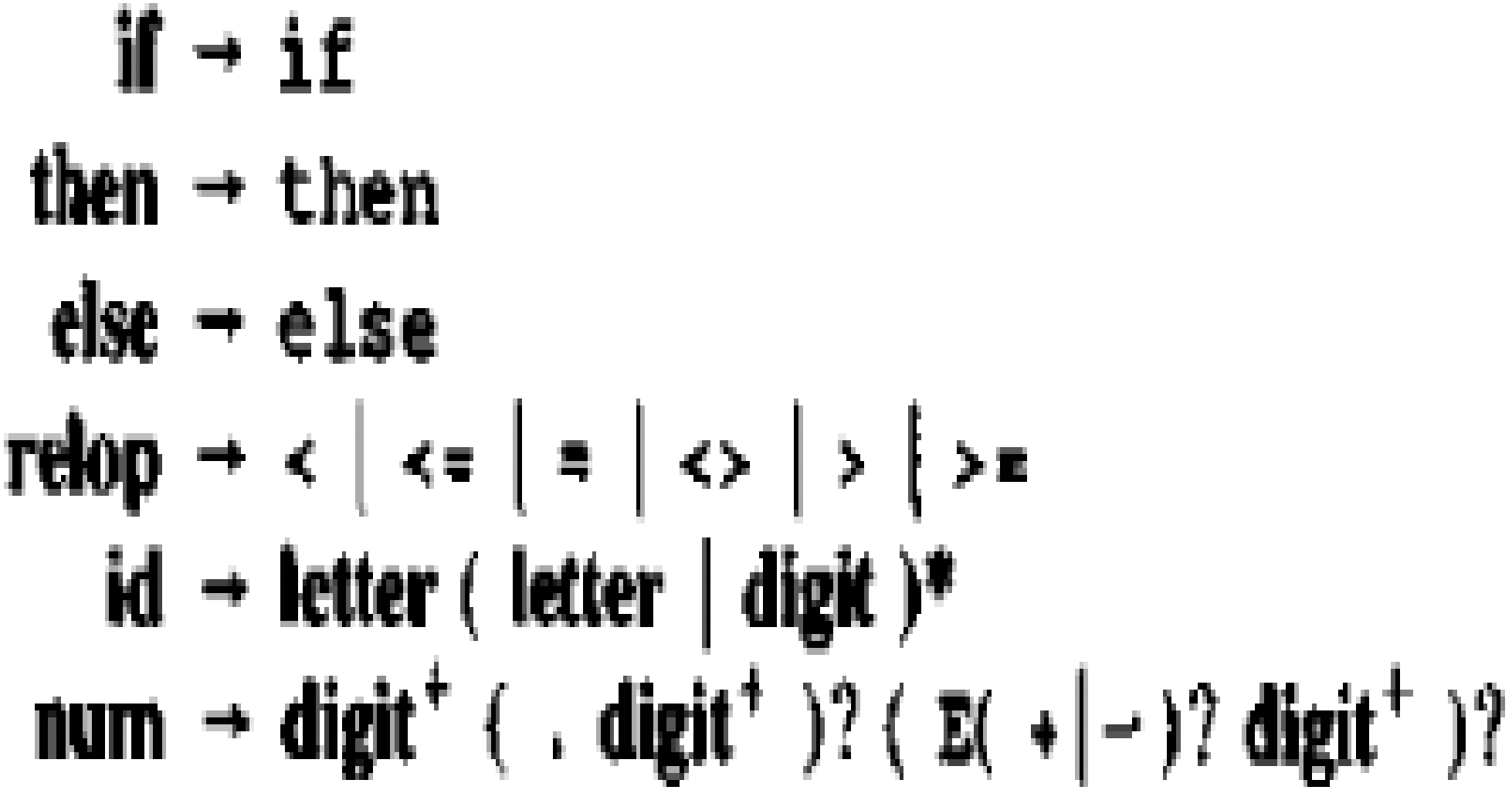
Recognition of tokens

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

expr → *term* **relop** *term*
| *term*

term → **id**
| **num**

Terrminals generated by following RDs



LA for stripping whitespaces

$$\begin{aligned} \textbf{delim} &\rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\ \textbf{ws} &\rightarrow \textbf{delim}^{+} \end{aligned}$$

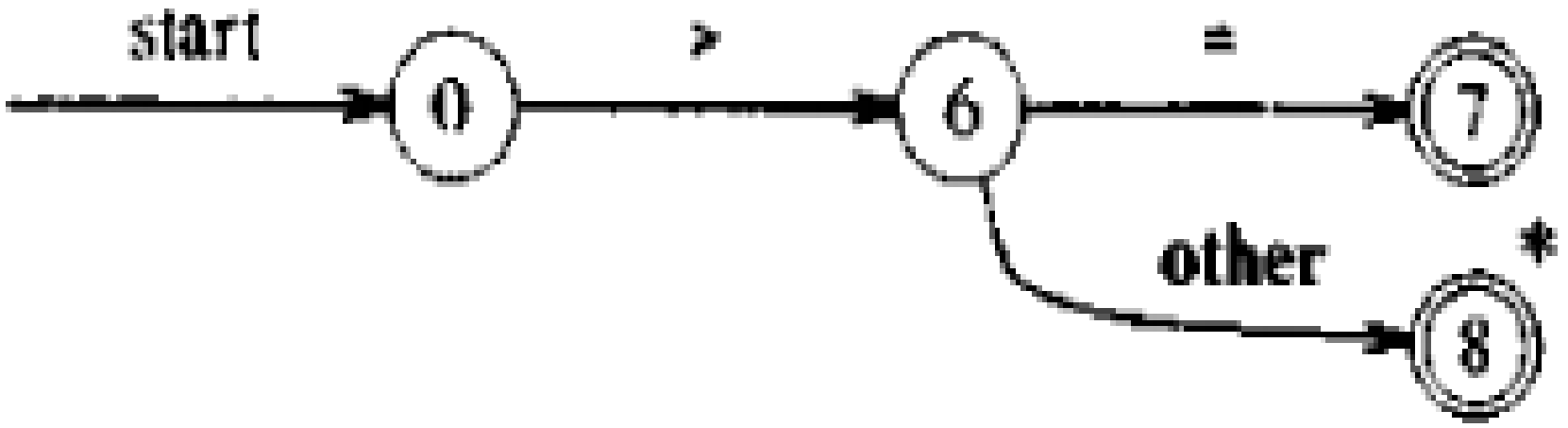
LA → tokens

REGULAR EXPRESSION	TOKEN	ATTRIBUTE-VALUE
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

LT, LE, EQ, etc are symbolic constants.

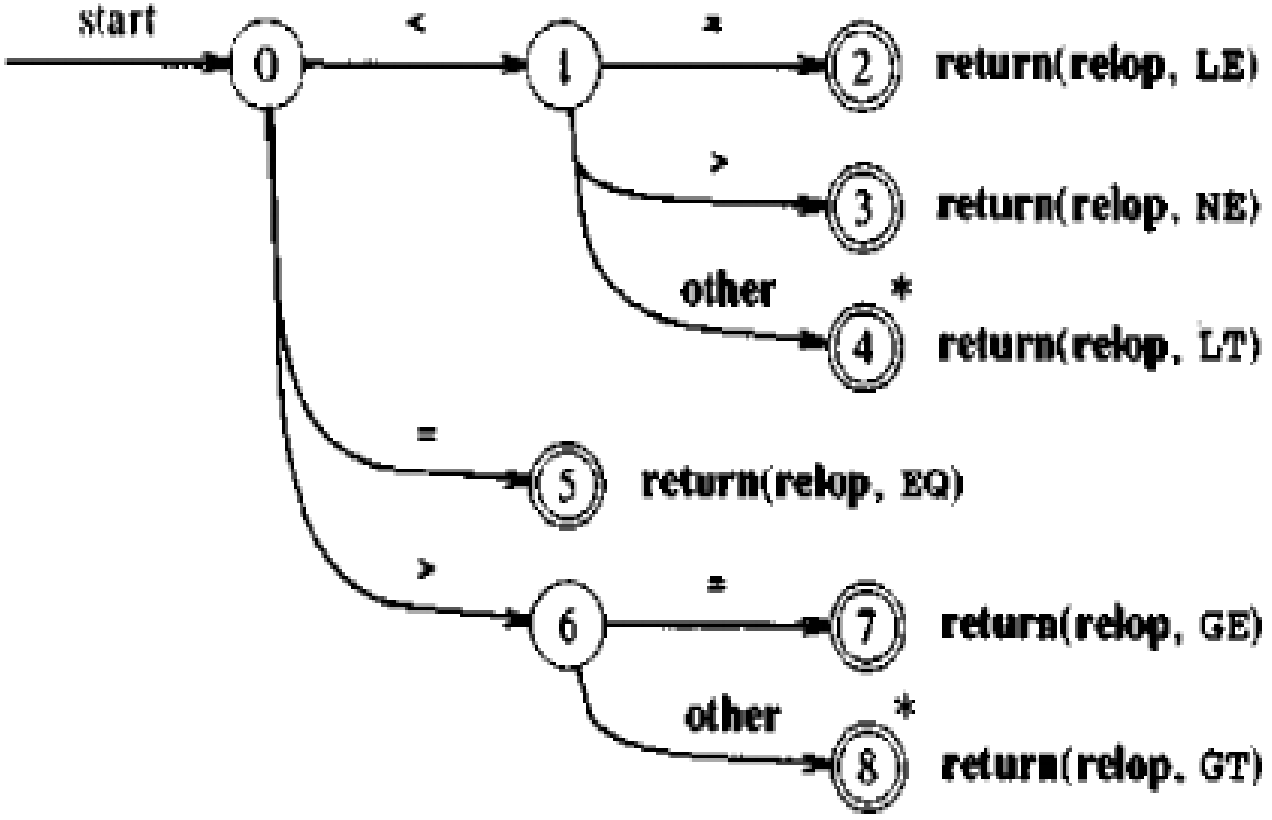
Transition Diagrams

- For the pattern **>=** and **>**

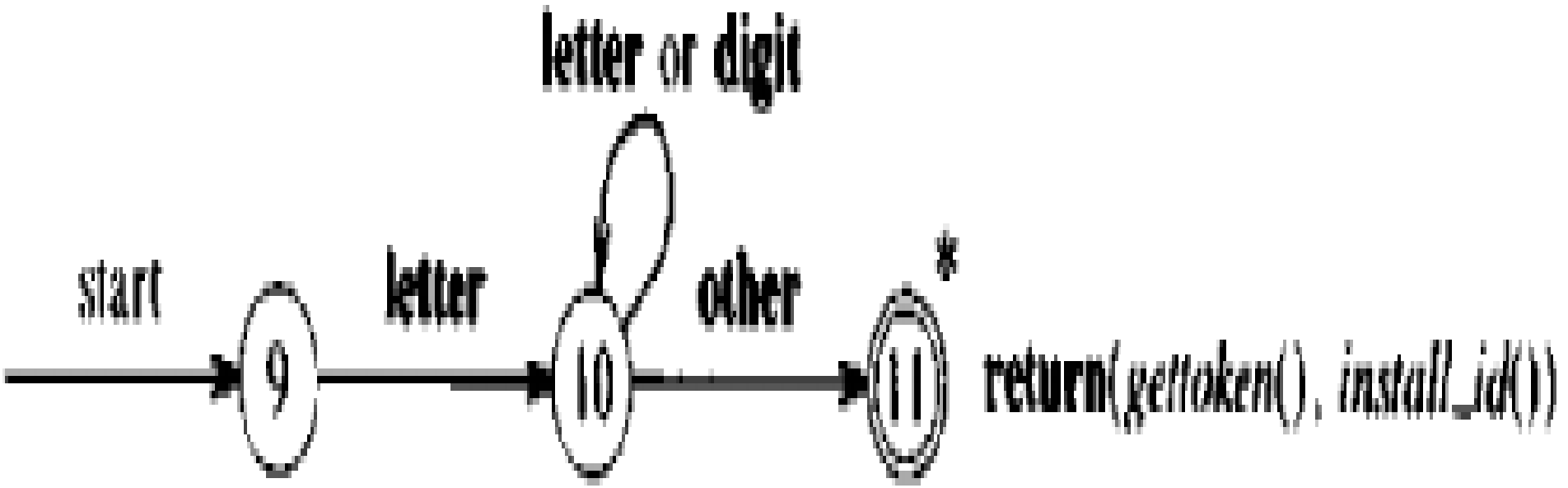


8* means that we need to retract !

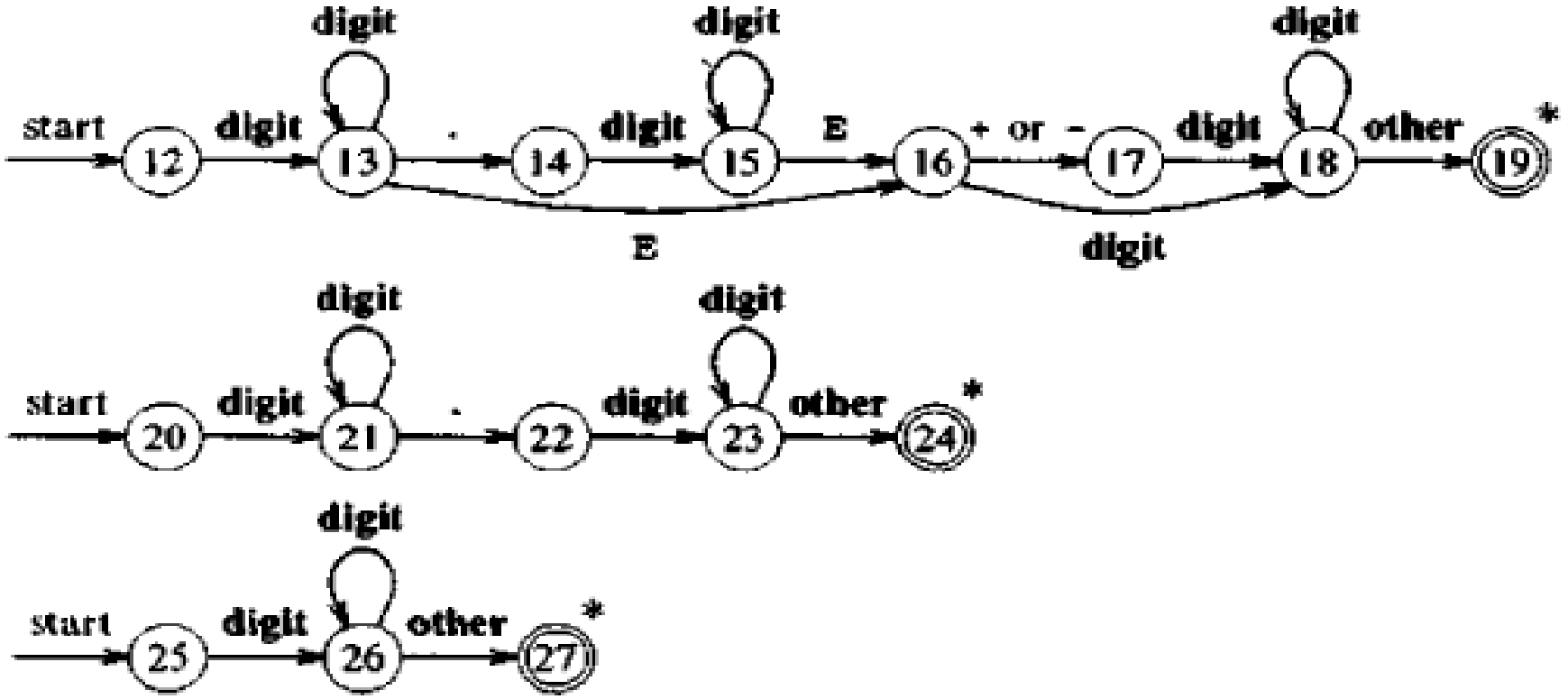
TD (for *relop*)


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)

TD (keywords & identifiers)



TDs for unsigned numbers



TD for whitespaces

- *delim --> blank I tab I newline*
- *ws --> delim+*
-

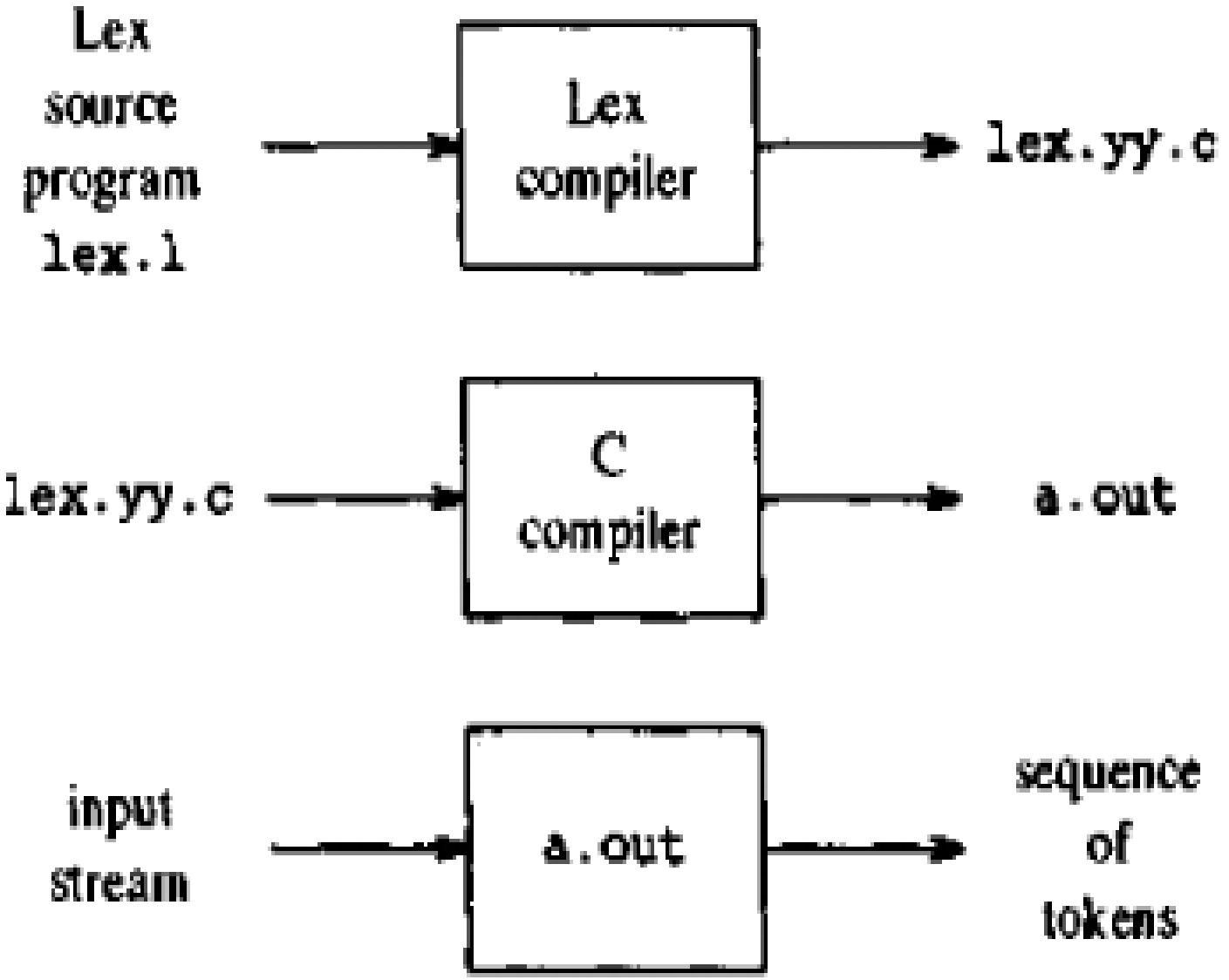
Implementing a TD

- `While (1)` *stmt*

```
int state = 0, start = 0;
int lexical_value;
     /* to "return" second component of token */

int fail()
{
     forward = token_beginning;
     switch (start) {
         case 0:    start = 9; break;
         case 9:    start = 12; break;
         case 12:   start = 20; break;
         case 20:   start = 25; break;
         case 25:   recover(); break;
         default:    /* compiler error */
      }
     return start;
}
```

```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c is lookahead character */
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
                    /* advance beginning of lexeme */
            }
            else if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else state = fail();
            break;

            ... /* cases 1-8 here */

        case 9:   c = nextchar();
            if (isletter(c)) state = 10;
            else state = fail();
            break;
        case 10:  c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        case 11:  retract(1); install_id();
            return ( gettoken() );

            ... /* cases 12-24 here */

        case 25:  c = nextchar();
            if  (isdigit(c)) state = 26;
            else state = fail();
            break;
        case 26:  c = nextchar();
            if (isdigit(c)) state = 26;
            else state = 27;
            break;
        case 27:  retract(1); install_num();
            return ( NUM );
        }
    }
}
```

Language for specifying LA



Lex Specifications

declarations

%%

translation rules

%%

auxiliary procedures

Translation rules in Lex

```
p1        { action1 }
p2        { action2 }
. ...
pn        { actionn }
```

```
%{
    /* 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 = 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);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%

install_id() {
    /* procedure 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 */
}

install_num() {
    /* similar procedure to install a lexeme that
    is a number */
}
```

Lex: Conflict Resolution

- Prefer longer prefix
- When longest prefix matches >= 2 patterns

  prefer the first in Lex