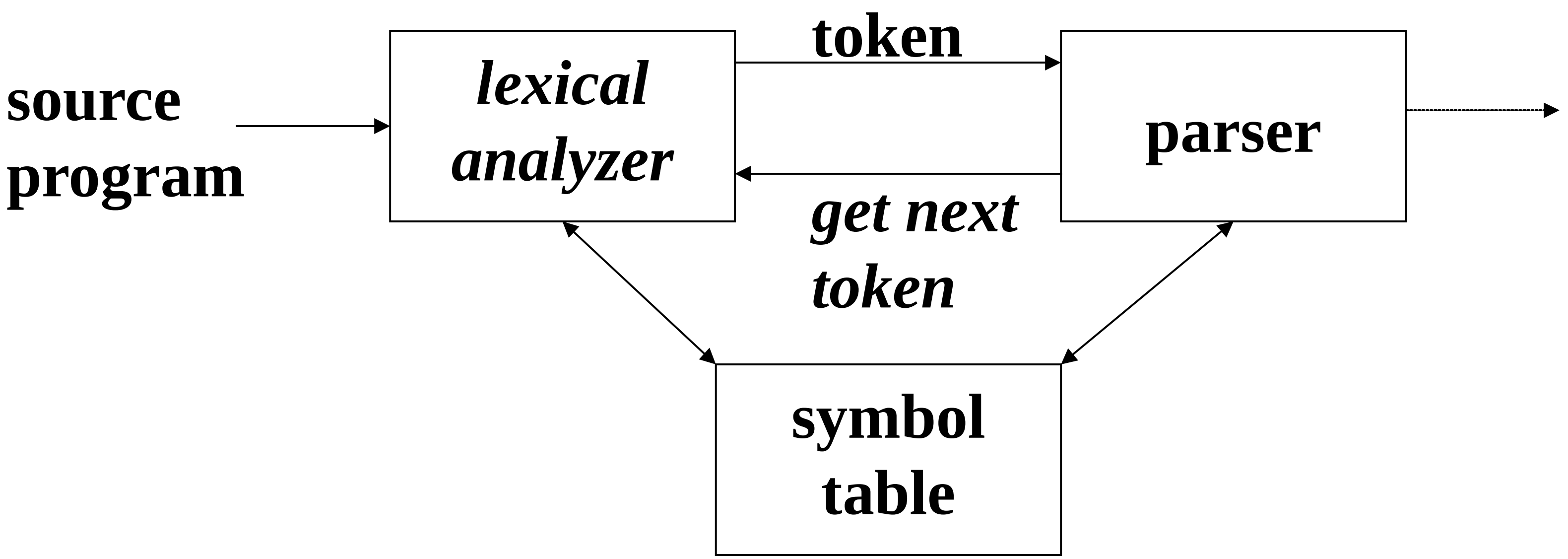
Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser

Why to separate Lexical analysis and parsing

- Simplicity of design
- Improving compiler efficiency
- Enhancing compiler portability

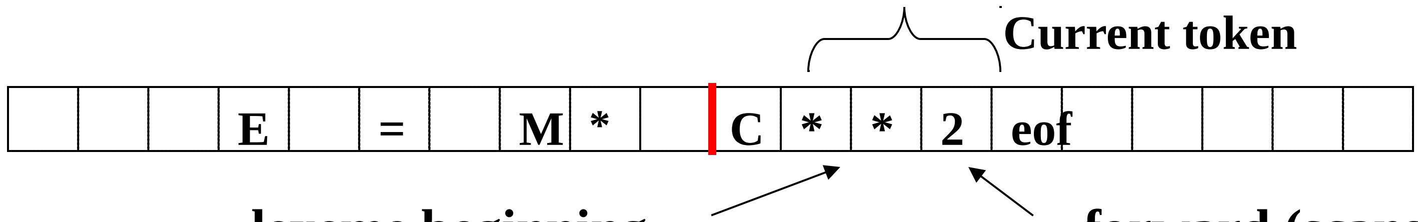
Tokens, Patterns, and Lexemes

- A token is a pair a token name and an optional token attribute
- 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
relation	< 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"

Using Buffer to Enhance Efficiency

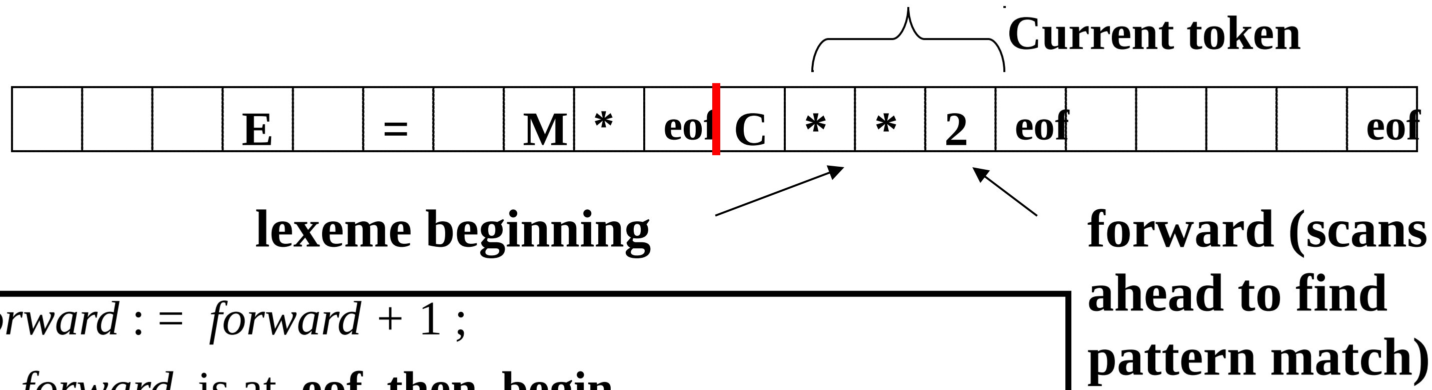


lexeme beginning

forward (scans ahead to find pattern match)

```
if forward at end of first half then begin
      reload second half ;  ←—— Block I/O
      forward := forward + 1
   end
   else if forward at end of second half then begin
      reload first half ;  ←— Block I/O
      move forward to biginning of first half
   end
   else forward := forward + 1 ;
```

Algorithm: Buffered I/O with Sentinels




```
forward : =  forward + 1 ;
if  forward  is at  eof  then  begin
    if  forward  at end of first half  then  begin
        reload second half ;  ← Block I/O
        forward  : =  forward  + 1
    end
    else if  forward  at end of second half  then  begin
        reload first half ;  ← Block I/O
        move  forward  to biginning of first half
    end
    else  / * eof within buffer signifying end of input  * /
        terminate lexical analysis
end
```

2nd **eof** ⇒ no more input !

Chomsky Hierarchy

0 Unrestricted	$\alpha A \beta \rightarrow \alpha \gamma \beta$
1 Context-Sensitive	$\mid LHS \mid \leq \mid RHS \mid$
2 Context-Free	$\mid LHS \mid = 1$
3 Regular	$\mid RHS \mid = 1$ or 2 , $A \rightarrow a \mid aB$, or $A \rightarrow a \mid Ba$

Formal Language Operations

OPERATION	DEFINITION
union **of L and M written $L \cup M$**	**$L \cup M$ = {s | s is in L or s is in M}**
concatenation of **L and M written LM**	**LM = {st | s is in L and t is in M}**
Kleene closure **of L written L***	$$L^* = \bigcup_{i=0}^{\infty} L^i$$ **L* denotes "zero or more concatenations of " L**
positive closure **of L written L^+**	$$L^+ = \bigcup_{i=1}^{\infty} L^i$$ **L^+ denotes "one or more concatenations of " L**

Formal Language Operations Examples

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

$L \cup D = \{A, B, C, D, 1, 2, 3\}$

$LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$

$L^2 = \{ AA, AB, AC, AD, BA, BB, BC, BD, CA, \ldots DD\}$

$L^4 = L^2 L^2 = ??$

$L^* = \{$ All possible strings of L plus $\in \}$

$L^+ = L^* - \in$

$L (L \cup D) = ??$

$L (L \cup D)^* = ??$

Language & Regular Expressions

- A Regular Expression is a Set of Rules / Techniques for Constructing Sequences of Symbols (Strings) From an Alphabet.

- Let Σ Be an Alphabet, r a Regular Expression Then L(r) is the Language That is Characterized by the Rules of r

Rules for Specifying Regular Expressions:

fix alphabet Σ

- ∀ **∈ is a regular expression denoting {∈ }**
- **If a is in Σ, a is a regular expression that denotes {a}**
- **Let r and s be regular expressions with languages L(r) and L(s). Then**

precedence

(a) (r) | (s) is a regular expression ⇒ L(r) ∪ L(s)

(b) (r)(s) is a regular expression ⇒ L(r) L(s)

(c) (r)* is a regular expression ⇒ (L(r))*

(d) (r) is a regular expression ⇒ L(r)

All are Left-Associative. Parentheses are dropped as allowed by precedence rules.

EXAMPLES of Regular Expressions

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

$A \mid B \mid C \mid D = L$

$(A \mid B \mid C \mid D)(A \mid B \mid C \mid D) = L^2$

$(A \mid B \mid C \mid D)^* = L^*$

$(A \mid B \mid C \mid D)((A \mid B \mid C \mid D) \mid (1 \mid 2 \mid 3)) = L(L \cup D)$

Algebraic Properties of Regular Expressions

AXIOM	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = r s | r t (s | t) r = s r | t r	concatenation distributes over |
∈ r = r r∈ = r	∈ Is the identity element for concatenation
r* = (r | ∈)*	relation between * and ∈
r** = r*	* is idempotent

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

Given Tokens, What are Patterns ?

Grammar:

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

expr → *term* relop *term* | *term*

term → id | num

if ← **if**

then ← **then**

else ← **else**

relop ← < | <= | > | >= | = | <>

id ← **letter (letter | digit)***

num ← **digit $^+$ (. digit $^+$) ? (E(+ | -) ? digit $^+$) ?**

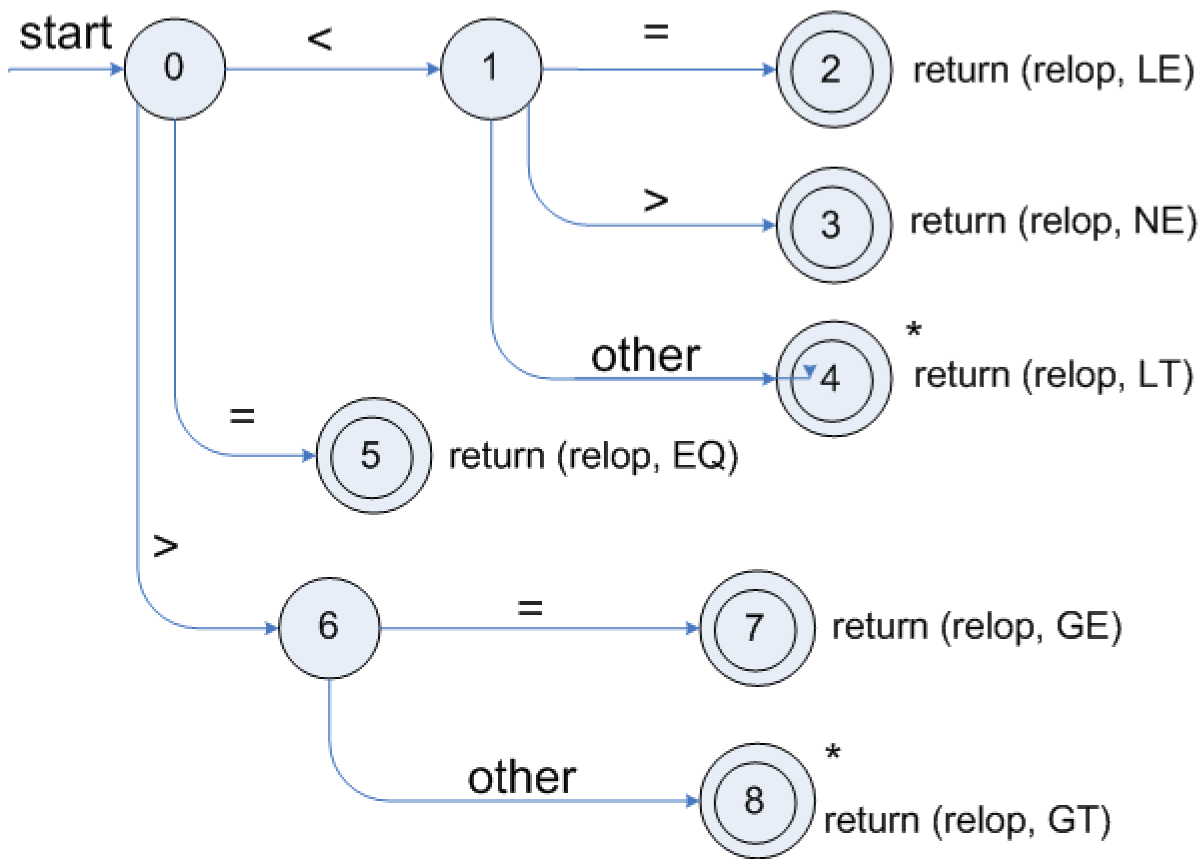
Overall

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

Note: Each token has a unique token identifier to define category of lexemes

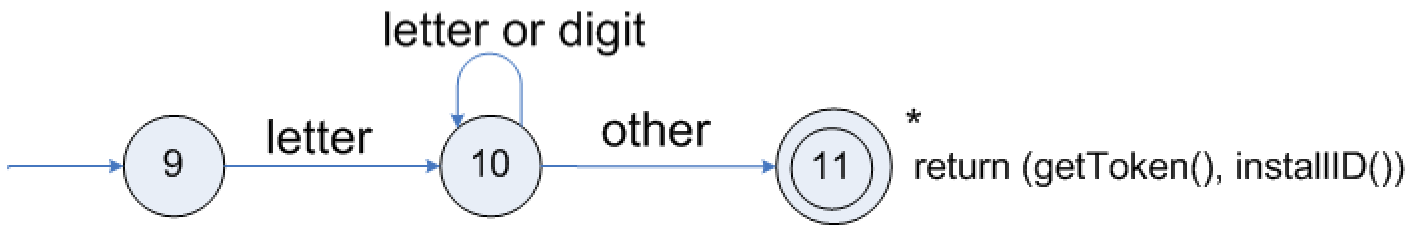
Transition diagrams

- Transition diagram for relop



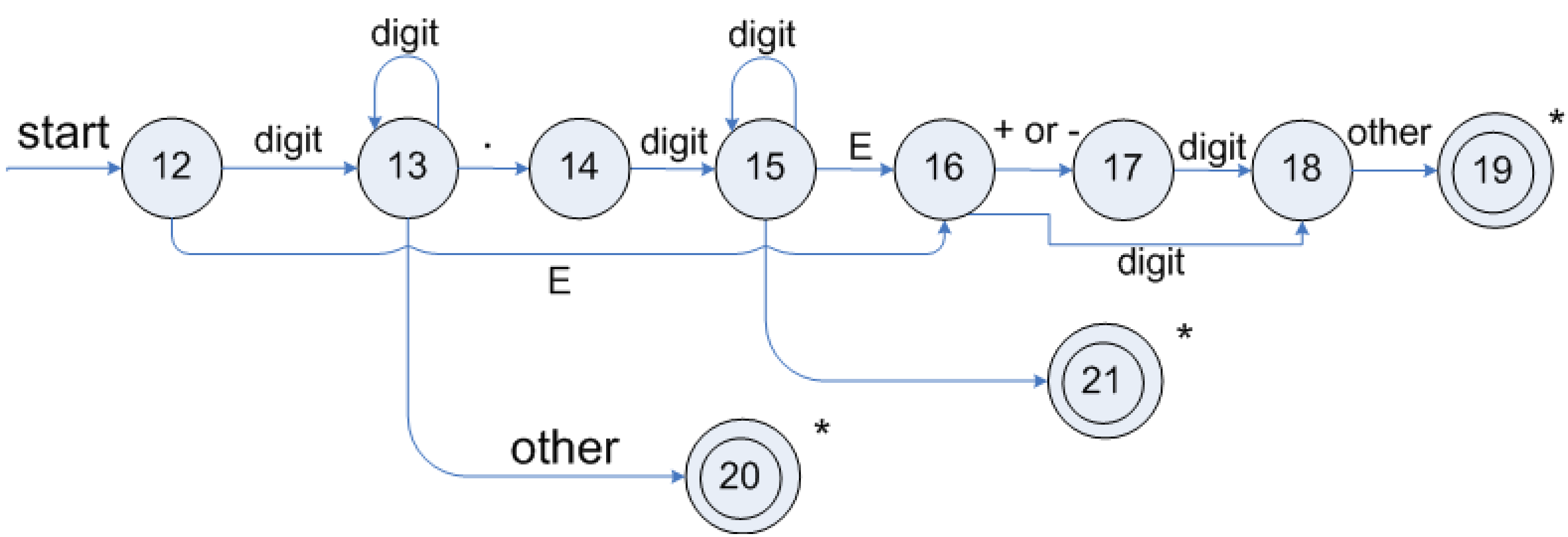
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



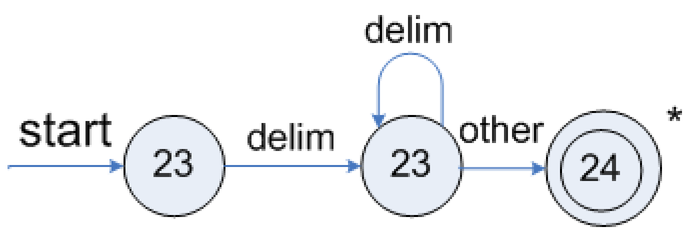
Transition diagrams (cont.)

- Transition diagram for unsigned numbers

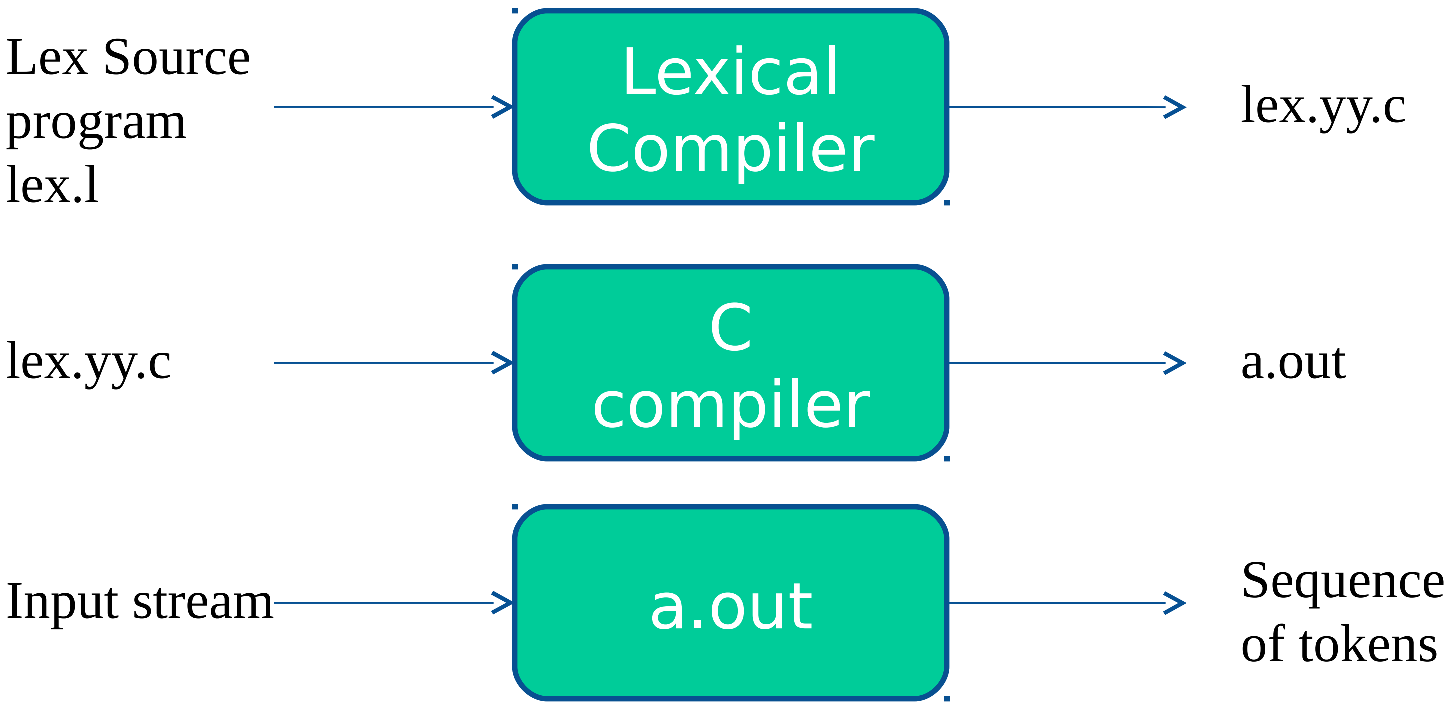


Transition diagrams (cont.)

- Transition diagram for whitespace



Lexical Analyzer Generator - Lex



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
- Minimal Distance