CS 4300: Compiler Theory

Chapter 3
Lexical Analysis

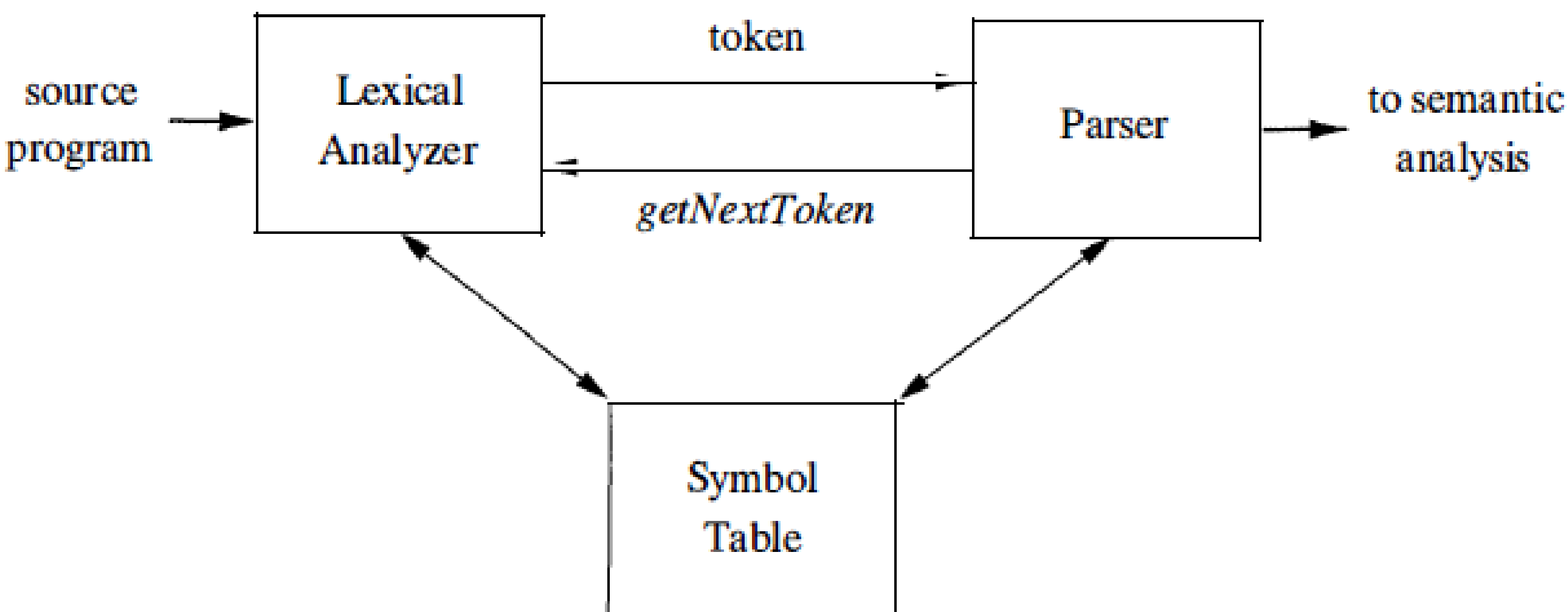
Dr. Xuejun Liang

Outlines (Sections)

1. The Role of the Lexical Analyzer
2. Input Buffering (Omit)
3. Specification of Tokens
4. Recognition of Tokens
5. The Lexical -Analyzer Generator Lex
6. Finite Automata
7. From Regular Expressions to Automata
8. Design of a Lexical-Analyzer Generator
9. Optimization of DFA-Based Pattern Matchers*

1. The Role of the Lexical Analyzer

- As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.



Why Lexical Analysis and Parsing (Syntax Analysis) are Separate

- Simplifies the design of the compiler
  - LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be normalized (e.g. UTF8, trigraphs)

Tokens, Patterns, and Lexemes

- A *token* is a pair consisting of a token name and an optional attribute value
  - The token name is an abstract symbol representing a kind of lexical unit
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **abc** and **123**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"

Examples of Tokens

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:

1. One token for each keyword
2. Tokens for the operators
3. One token representing all identifiers
4. One or more tokens representing constants
5. Tokens for each punctuation symbol

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
  - **number**
- We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information
  - E.g. **id** has its lexeme, its type, and the location at which it is first found
  - So the appropriate attribute value for an **id** is a pointer to the symbol-table entry for that identifier (lexeme)

Example of Attributes for Tokens

- Example: lexemes, token names and associated attribute values for the Fortran statement.

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>

3. 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 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

String Operations

- The *concatenation* of two strings x and y is denoted by xy
- The *exponentiation* of a string s is defined by

$$s^0 = \varepsilon$$
$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that $s\varepsilon = \varepsilon s = s$

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*

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

Example:
Compute
$L \cup D$
LD
D^4
D^*
$L(L \cup D)^*$
D^+

where
$L = \{A, B, ..., Z, a, b, ..., z\}$
and $D = \{0, 1, . . . 9\}$

Regular Expressions Over Some Alphabet Σ

- 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*

Precedence of regular expression operations

a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative
c) | has lowest precedence and is left associative

Algebraic laws for regular expression operations

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Example 3.4 : Let $\Sigma = \{a, b\}$, what are languages denoted by The following regular expressions:

a|b, **(a|b)(a|b)**, **a***, **(a|b)***, **a|a*b**

Regular Definitions Over Some Alphabet Σ

- Regular definitions introduce a naming convention with name to regular expression bindings:

$$
\begin{aligned}
&d_1 \to r_1 \\
&d_2 \to r_2 \\
&\ldots \\
&d_n \to r_n
\end{aligned}
$$

  where:
  - Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
  - each r_i is a regular expression over

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

Regular Definitions: Examples

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id & \rightarrow & letter\_ \, (\, letter\_ \mid digit \,)^*
\end{array}
$$

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \rightarrow & digit\ digit^* \\
optionalFraction & \rightarrow & \texttt{.}\ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\, \texttt{E}\, (\, \texttt{+} \mid \texttt{-} \mid \epsilon \,)\ digits \,) \mid \epsilon \\
number & \rightarrow & digits\ optionalFraction\ optionalExponent
\end{array}
$$

Numbers: 5280, 0.01234, 6.336E4, or 1.89E-4.

Regular Definitions: Extensions

- The following shorthands are often used:

One or more instances: +	$r^+ = rr^*$
Zero or one instance: ?	$r? = r \mid \varepsilon$
Character classes:	$[\mathtt{a}\text{-}\mathtt{z}] = \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \ldots \mid \mathtt{z}$

- Examples:

$$
\begin{aligned}
\mathit{letter\_} &\rightarrow [\mathtt{A\text{-}Za\text{-}z\_}] \\
\mathit{digit} &\rightarrow [0\text{-}9] \\
\mathit{id} &\rightarrow \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^*
\end{aligned}
$$

$$
\begin{aligned}
\mathit{digit} &\rightarrow [0\text{-}9] \\
\mathit{digits} &\rightarrow \mathit{digit}^+ \\
\mathit{number} &\rightarrow \mathit{digits}\ (.\ \mathit{digits})?\ (\ \mathtt{E}\ [+-]?\ \mathit{digits}\ )?
\end{aligned}
$$

4. Recognition of Tokens

Example 3.8: A Grammar for branching statements

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

The terminals of the grammar, which are **if**, **then**, **else**, **relop** , **id**, and **number**, are the names of tokens for lexical analyzer.

Patterns for tokens of Example 3.8

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

Tokens, patterns, and attribute values

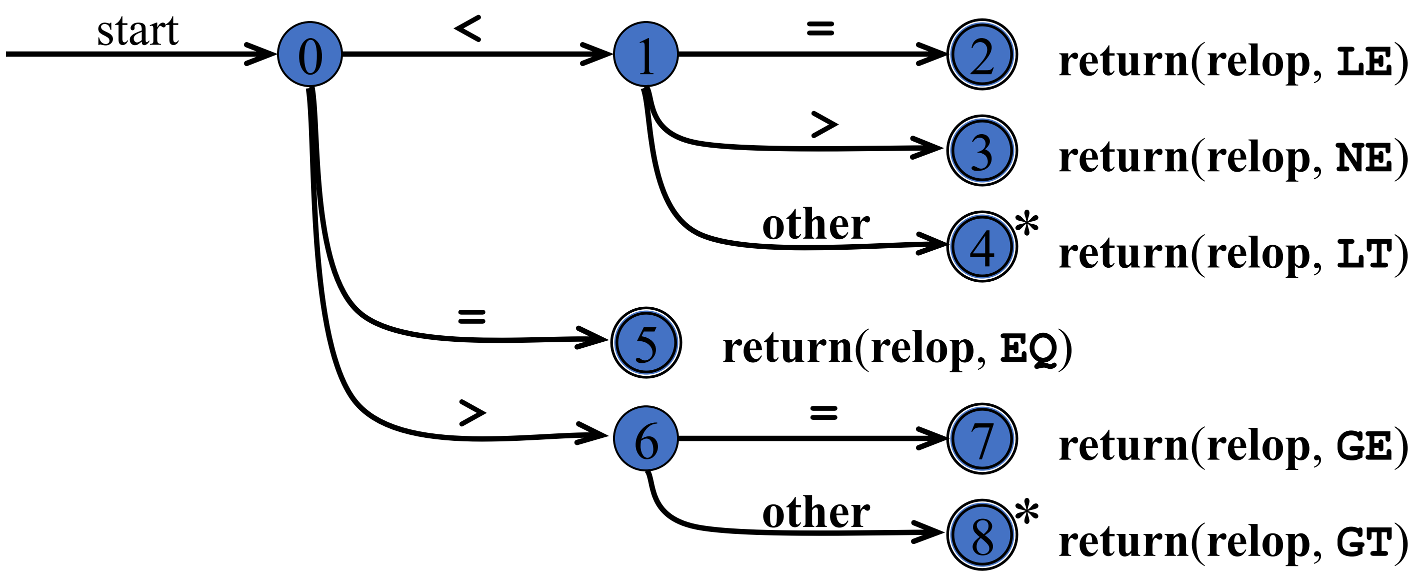
Lexemes	Token Name	Attribute Value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

whitespace

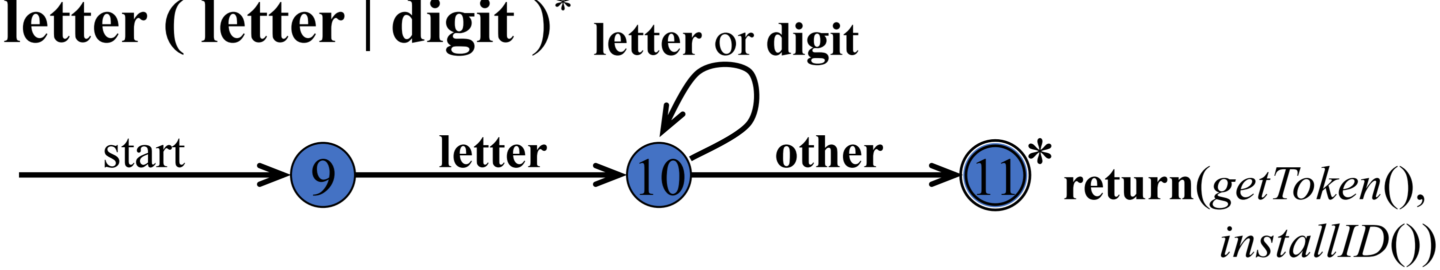
$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^+$

Transition Diagrams

relop → < | <= | <> | > | >= | =

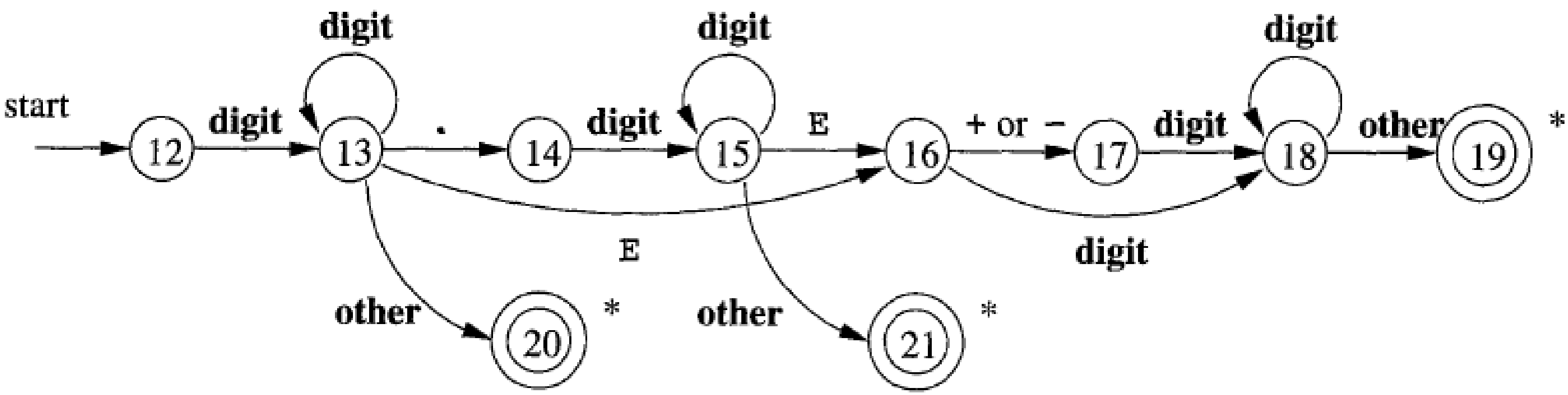


id → **letter** (**letter** | **digit**)*

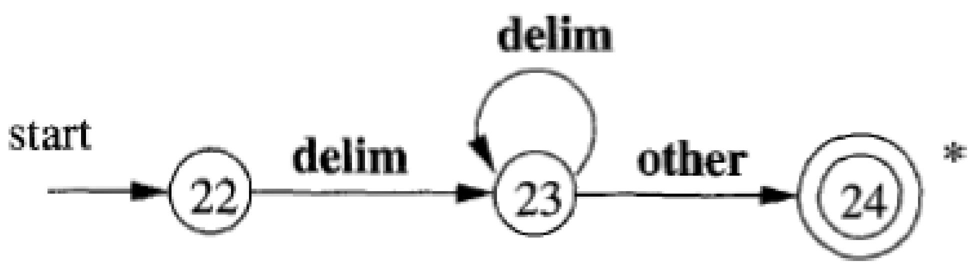


Transition Diagrams (Cont.)

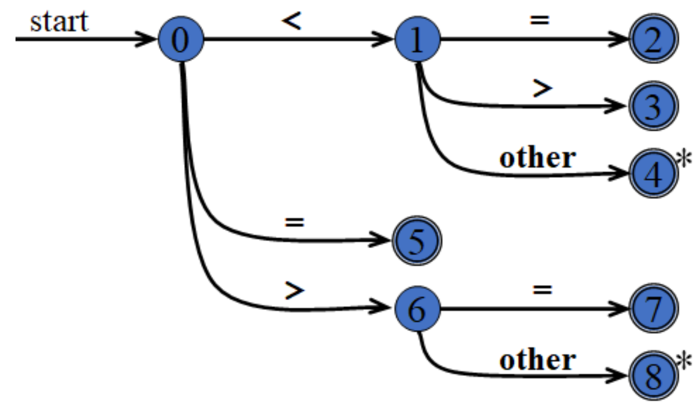
Unsigned number



Whitespace



Sketch of implementation of relop 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 relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```