BHCS15B: System Programming

Lexical Analysis

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Role of a Lexical Analyzer
2. Specification of Tokens
3. Recognition of Tokens
4. Symbol Table
5. Lexical Analyzer Generator – Lex (covered in the course wiki)

Role of a Lexical Analyzer

- As the *first phase* of a compiler, the main task of the *scanner* is to:
  - *Read* the input characters of the source program,
  - *Group* them into *lexemes,*
  - *Produce* as output a sequence of *token* for each lexeme in the source program.
- The stream of tokens is sent to the *parser* for syntax analysis.
- *Scanner* also interacts with the *symbol table* for:
  - Storing lexeme(identifiers)
  - Reading information regarding the kind of identifier, to assist it in determining the proper token it must pass to the parser.

Role of a Lexical Analyzer (2)

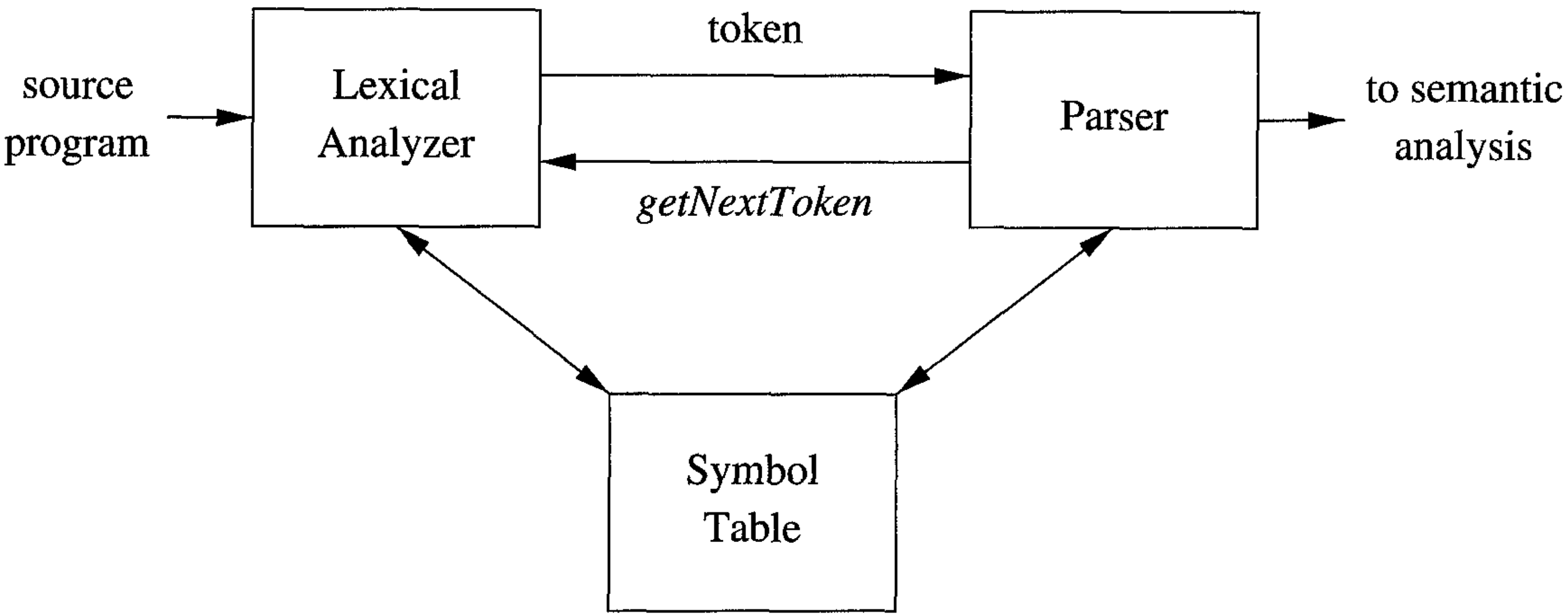


Figure 3.1: Interactions between the lexical analyzer and the parser [1]

Role of a Lexical Analyzer (3)

- Other important tasks performed by lexical analyzer (besides identifications of lexemes) :
  - Stripping out comments and *whitespace* (*blank, newline, tab, etc.*)
  - Correlating error messages generated by the compiler with the source program.
    - → *Keep track of no. of newline characters seen, so that it can associate a line no. with each error message.*
    - → *In some compilers, it makes a copy of the source program, with error messages inserted at the appropriate positions.*
  - Expansion of macros may also be performed by the lexical analyzer.
    - → *if macro-preprocessors are used in the source program.*

Role of a Lexical Analyzer (4)

- Sometimes, lexical analyzer are divided into a cascade of two processes:

  a. ***Scanning*** consists of the simple processes that do not require tokenization.
     - → *Deletion of comments.*
     - → *Compaction of consecutive whitespace characters into one.*

  b. ***Lexical Analysis*** proper is the more complex portion.
     - → *Where the scanner produces the sequence of tokens as output.*

Lexical Analysis vs. Parsing

- Number of reasons why analysis portion of a compiler is normally *separated* into lexical analysis(*scanner*) and syntax analysis(*parser*) phases:

  a. Simplicity of design (*separating concerns results in a cleaner overall design*)
     - → *Lexical analysis will remove unwanted things, such as comments, whitespaces, etc.*
     - → *Parser will focus on snytactic concerns.*

  b. Compiler efficiency is improved
     - → *Specialized techniques can be applied to each phase.*
     - → *Specialized buffering techniques for reading input characters can speed up the compiler.*

  c. Compiler portability is enhanced
     - → *Input-device-specific peculiarities can be restricted to the lexical analysis.*

Tokens, Patterns, and Lexemes

- All three are related but distinct terms:

  ⓐ Token (*a pair consisting of a token name and optional attribute value*)

  → *Token name is an abstract symbol, representing a kind of lexical unit. For example: a keyword, or an identifier.*

  → *Token names are the input symbols that the parser processes.*

  → *Tokens are often referred by its token name, and we shall generally write the name of token in bold face.*

  ⓑ Pattern (*a description of the form that the lexemes of a token may take*)

  → *For a keyword as a token, the pattern is just the sequence of characters that form the keyword.*

  → *For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.*

Tokens, Patterns, and Lexemes (2)

- Lexeme (the piece of the original program from which token is made)
  - → *A sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.*

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"

Figure 3.2: Examples of tokens

Tokens, Patterns, and Lexemes (3)

- In many programming languages, the following *classes* cover most or all of the tokens.

  1. One token for each *keyword*. The pattern for a keyword is the same as the keyword itself.
  2. Tokens for the *operators*, either individually or in classes such as the token comparison.
  3. One token representing all *identifiers*.
  4. One or more tokens representing *constants*, such as numbers and literal strings.
  5. Tokens for each *punctuation symbol*, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the additional information about the lexeme that matched, to the subsequent phase.

  - For example: the pattern for token *number* matches both 0 and 1, but it is extremely important for the *code generator* to know which lexeme was found in the source program

    → *The lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token.*

    → *The token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.*

- We assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information.

Attributes for Tokens (2)

- The most important example is the token id, where we need to associate with the token a great deal of information

- Normally, information about an identifier – eg., its lexeme, its type, and the location at which it is first found is kept in the symbol table.

- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Attributes for Tokens (3)

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<**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>

Lexical Errors

- Without the aid of other components, it is hard for a lexical analyzer to tell that there is a source-code error.

- For example:

fi (a == f (x)) ...

→ *a lexical analyzer cannot tell whether **f i** is a misspelling of the keyword **i f** or an undeclared function identifier.*

→ *Since **f i** is a valid lexeme for the token **id**, the lexical analyzer must return the token **i d** to the parser and let some other phase of the compiler(probably the parser) handle an error due to transposition of the letters.*

Lexical Errors (2)

- Some of the possible error-recovery actions, when lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input:

    1. Delete one character from the remaining input.
    2. Insert a missing character into the remaining input.
    3. Replace a character by another character.
    4. Transpose two adjacent characters.
    5. The simplest recovery strategy is "panic mode" recovery.

        → *delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.*

- Transformations like these may be tried in an attempt to repair the input.

Specification of Tokens

- ***Regular expression*** are an important notation for specifying lexeme patterns. (A formal way to specify patterns)

- Following concepts will be covered in this section:

  1. Strings and Languages
  2. Operations on Languages
  3. Regular Expressions
  4. Regular Definitions
  5. Extensions of Regular Expressions.

Alphabets

- An ***alphabet*** is any finite set of symbols (letters, digits, punctuation)
  - In English an Alphabet is a finite set of 26 letters {A,B,C...,Z}.
  - In Hindi an Alphabet is a finite set of 52 letters {अ, आ, इ,...,क्ष त्र ज्ञ}.
  - An alphabet is denoted by **Σ** (*Greek letter sigma*).
  - For example:
    - → ***Σ = {0,1}*** *is a binary alphabet over symbols* ***0, 1***
    - → ***Σ = {a,b,c}*** *is an alphabet over symbols* ***a, b, c***
  - ASCII is an important example of an alphabet, used in many software systems.
  - Unicode is another important example of an alphabet, contains characters from most written languages all over the world.

Strings

- A ***string*** over an alphabet is a finite sequence of symbols chosen from that alphabet. (formed by concatenating a finite number of symbols in the alphabet)
  - For example:
    - → If **Σ = {a,b,c}** : abc, abaa, baba, cbbbaaabb, cccccc, cbcab, ... etc. are strings.
    - → If **Σ = {0,1}** some of the strings could be 0, 1, 00, 11, 0101011, 000111, ...
- *Length* of a string ***s***, usually written **| *s* |**, is the *number of occurrences of symbols in s*. For example:
    - → **ANDC** is a string of length four.
- *Null or Empty* string, denoted by (**e** or **Λ** or **ε**), is the string of length zero.
- In language theory, the terms "sentence" and "word" are often used as synonyms for "string".

Strings (2)

- ***Concatenation of strings:*** If **x** and **y** are strings, then the concatenation of **x** and **y**, denoted **xy**, is the string formed by appending **y** to **x**.
  - For example:
    - → if **x** = *Delhi* and **y** = *University*, then **xy** = *DelhiUniversity*.
  - The empty string is the identity under concatenation;
    - → *for any string* **s**, **εs** = **sε** = **s**
  - *Concatenation as a product*, we can define the *"exponentiation"* of strings as follows:
    - → Define $\mathbf{s}^0$ to be **ε**, and and for all **i > 0**, define $\mathbf{s}^i$ to be $\mathbf{s}^{i-1}\,\mathbf{s}$.
    - → Since **εs = s**, it follows that $\mathbf{s}^1 = \mathbf{s}$. Then $\mathbf{s}^2 = \mathbf{ss}$, $\mathbf{s}^3 = \mathbf{sss}$, and so on

Terms for Parts of Strings

- The following string related terms are commonly used:

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

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

3. *Substring of **s** is obtained by deleting any prefix and any suffix from **s**. For example: **banana, nan,** and **e** are substrings of banana.*

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

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

Language

- A ***language*** is set of strings over an alphabet.
  - For example:
    - → If **Σ = {a,b}**: then {a, ab, baa} is a language over alphabet{a,b}.
    - → If **Σ = {0,1}**: then {0, 111} is a language over alphabet{0,1}.
- *Empty* language (**ϕ**):
  - Just like **Empty set** is the language that **has no words**.
  - **Ɛ** or **Λ** is not a string in the language **ϕ** since this language has no words at all.
- Abstract languages like **ϕ** , the **empty set**, or { **Ɛ** } is the set containing only the empty string.

Operations on Languages

- In *lexical analysis*, the most important operations on languages are union, concatenation, and closure, as shown:

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

Figure 3.6: Definitions of operations on languages

Operations on Languages (2)

- In general, we can summarize Powers of **∑**
  - Σ^k = the set of all strings of length k
  - $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
  - $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$

Operations on Languages (3)

- ***Example 3.3:***

Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z } and

Let D be the set of digits {0,1,. . .9}.

We may think of L and D in two, essentially equivalent, ways.

One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits.

The *second way* is that L and D are languages, all of whose strings happen to be of length one.

Some other languages that can be constructed from languages L and D, using the operators are shown in the next slide:

Operations on Languages (4)

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.

2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

3. L^4 is the set of all 4-letter strings.

4. L^* is the set of all strings of letters, including ϵ, the empty string.

5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.

6. D^+ is the set of all strings of one or more digits.

Regular Expressions

- A *regular expression* is useful for describing all the languages that can be built from the operations applied to the symbols of some alphabet.

- Here are the ***rules*** that define the regular expressions over some alphabet **∑** and the languages that those expressions denote.

- BASIS: There are two rules that form the basis:

  1. ***Ɛ*** *is a regular expression, and* ***L(Ɛ)*** *is* ***{Ɛ)*** *, that is, the language whose sole member is the empty string.* ***or*** *(* ***Ɛ*** *is a regular expression for null string {****Ɛ****})*

  2. *If* ***a*** *is a symbol in* ***Σ*** *, then* ***a*** *is a regular expression, and* ***L(a) = {a}****, that is, the language with one string, of length one, with a in its one position. or (if* ***a*** *is symbol in* ***Σ*** *then* ***a*** *is regular expression for {****a****})*

Regular Expressions (2)

- INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose **r** and **s** are regular expressions denoting languages **L(r)** and **L(s)**, respectively:

1. **(r)|(s)** is a regular expression denoting the language **L(r) U L(s).**
2. **(r)(s)** is a regular expression denoting the language **L(r)L(s)** .
3. **(r)*** is a regular expression denoting **(L(r))*.**
4. **(r)** is a regular expression denoting **L(r)**.

Regular Expressions (3)

- *Regular expressions* often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

  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.

- Under these conventions, for example, we may replace the regular expression **(a)|((b)*(c))** by **a|b*c.**

  → *Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.*

Regular Expressions (4)

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \dots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Expressions – Algebraic Law

- A language that can be defined by a regular expression is called a *regular set*.
- If two regular expressions **r** and **s** denote the same regular set, we say they are equivalent and write **r = s**. For instance, **(a|b) = (b|a)**.

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

Figure 3.7: Algebraic laws for regular expressions

Regular Definitions

- It is a name given to the regular expressions, and you can use those names in the subsequent expressions, as if the names were themselves symbols.
- If **Σ** is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{array}{ccc} d_1 & \to & r_1 \\ d_2 & \to & r_2 \\ & \cdots & \\ d_n & \to & r_n \end{array}$$

- *where:*
  - → Each $\mathbf{d_i}$ is a new symbol, not in **Σ** and not the same as any other of the **d's**, and
  - → Each $\mathbf{r_i}$ is a regular expression over the alphabet $\mathbf{\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}}$.

Regular Definitions (2)

Example 3.5 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

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

Regular Definitions (3)

Example 3.6: Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

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

Extensions of Regular Expressions

1. *One or more instances* (+)

- The unary, postfix operator **+** represents the positive closure of a regular expression and its language.

- If **r** is a regular expression, then **(r)**$^+$ denotes the language **(L(r))**$^+$.

- The operator **+** has the same precedence and associativity as the operator *****.

- Two useful algebraic laws as shown below, relate the **Kleene closure** and **positive closure**:

  a. $r^* = r^+ | \varepsilon$

  b. $r^+ = rr^* = r^*r$

Extensions of Regular Expressions (2)

2 Zero or one instance (?)

- The unary, postfix operator **?** means "zero or one occurrence."
- That is, **r?** is equivalent to **r|ε**, or put another way, **L(r?) = L(r) U {ε}.**
- The operator **?** has the same precedence and associativity as the operator **+ and ***.

Extensions of Regular Expressions (3)

3 *Character Classes* (shorthand)

- A regular expression $\mathbf{a_1|a_2| \ldots |a_n}$, where the $\mathbf{a_i}$**'s** are each symbols of the alphabet, can be replaced by the shorthand $\mathbf{[a_1a_2 \ldots a_n]}$.

- More importantly, when $\mathbf{a_1 , a_2, \ldots , a_n}$, form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by $\mathbf{a_1\text{-}a_n}$, that is, just the first and last separated by a hyphen.

- Thus, **[abc]** is shorthand for **a|b|c**, and **[a-z]** is shorthand for **a|b| . . . |z**.

Extensions of Regular Expressions (4)

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{[A-Za-z\_]} \\
digit & \rightarrow & \texttt{[0-9]} \\
id & \rightarrow & letter\_ \ (\ letter \mid digit\ )^*
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^+ \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )?
\end{array}
$$

Recognition of Tokens

- Tokens are recognized with transition diagram.

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

(A grammar for branching statements)

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

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

(Patterns for tokens)

Recognition of Tokens (2)

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

Tokens, their patterns, and attribute values

Transition Diagrams

- Intermediate step in constructing lexical analyzer, we first convert patterns into flowcharts called *transition diagrams*. (the conversion from regular-expression patterns to transition diagrams)

- *Transition diagrams* have:

  1. *States*: collection of nodes or circles
     - → Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

  2. *Edges*: directed from one state to another.
     - → Each edge is labeled by a symbol or set of symbols.

Transition Diagrams (2)

- Some important conventions about transition diagrams are:

1. *Start state or initial state*
   - → indicated by an edge, labeled "start", entering from nowhere.
   - → The transition diagram always begins in the start state before any input symbols have been read.

2. *Accepting or final states*:
   - → indicates that a lexeme has been found, represented using a double circle.
   - → if there is an action to be taken, typically returning a token and an attribute value to the parser, we shall attach that action to the accepting state.

3. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state.

Transition Diagrams (3)

- Transition Diagram for **relop**

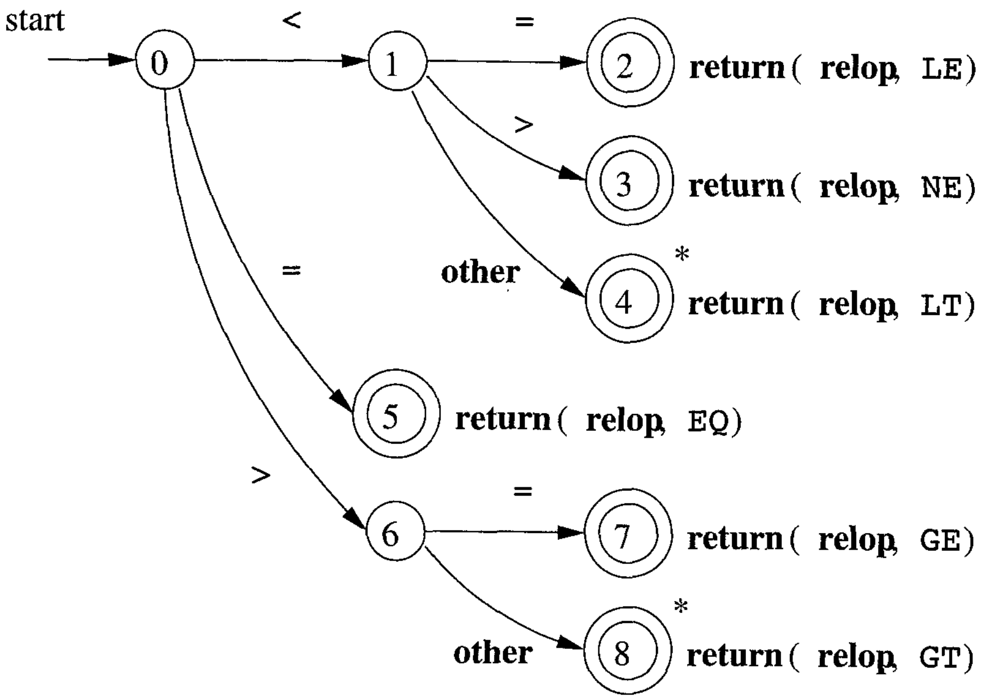


Figure 3.13: Transition diagram for **relop**

Transition Diagrams (3)

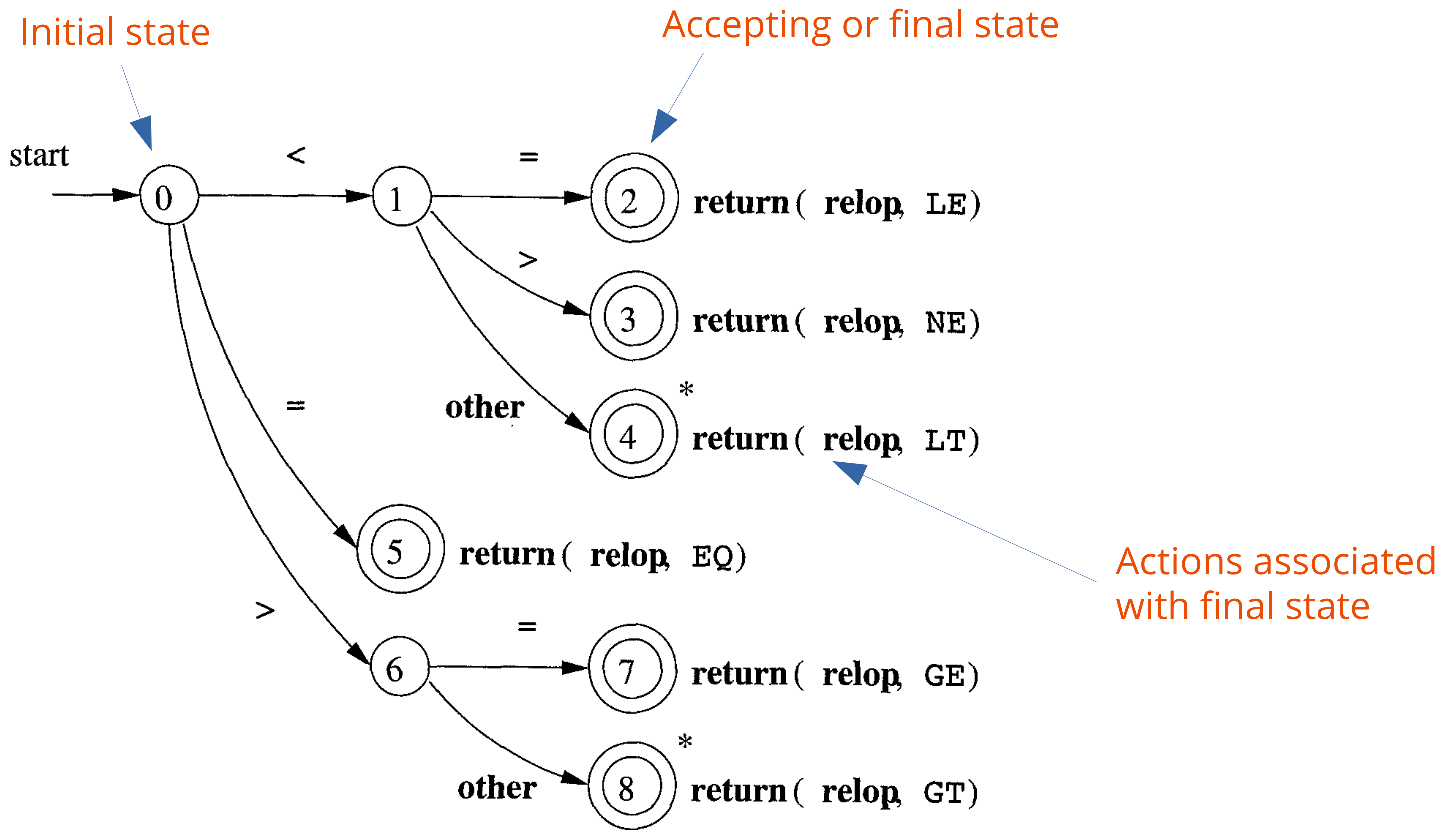


Figure 3.13: Transition diagram for **relop**

Transition Diagrams (3)

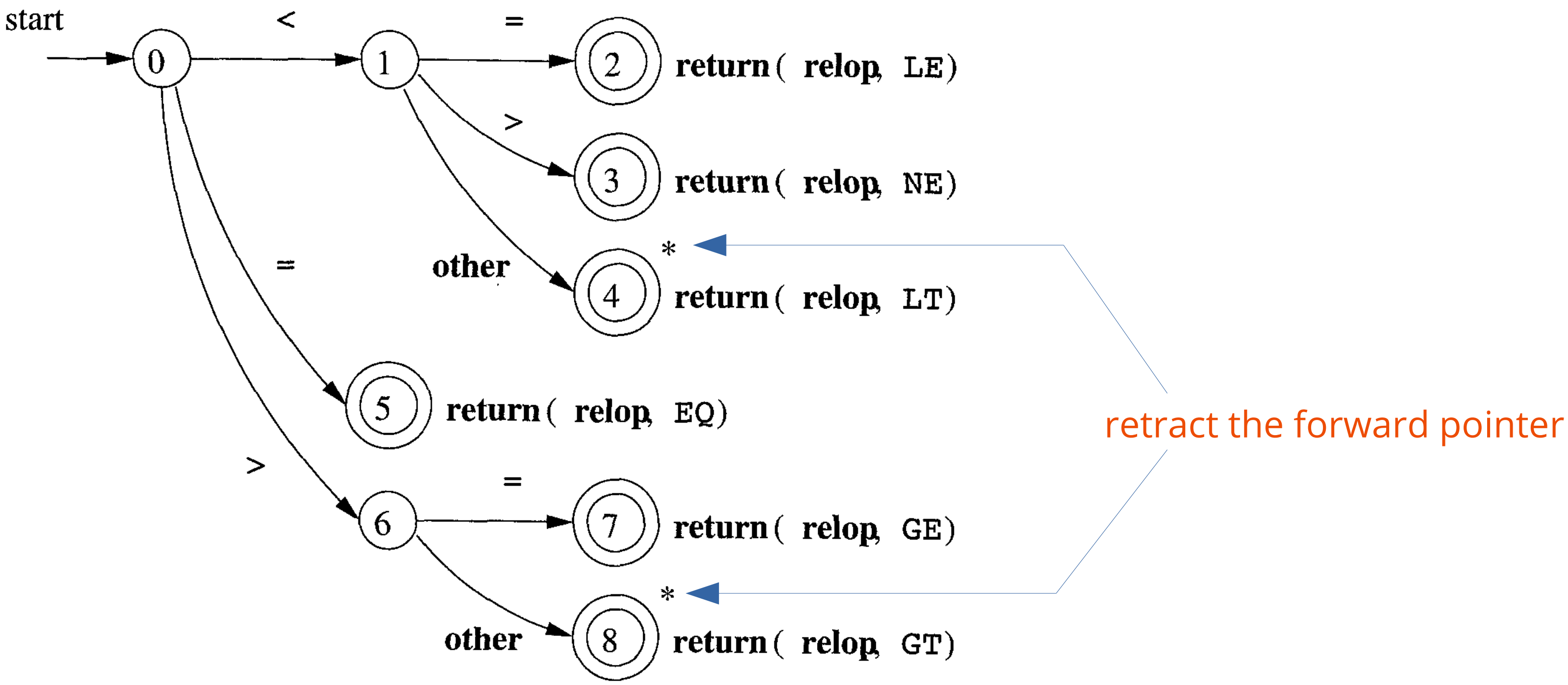


Figure 3.13: Transition diagram for **relop**

Transition Diagrams (4)

- Transition Diagram for **id's and keywords**

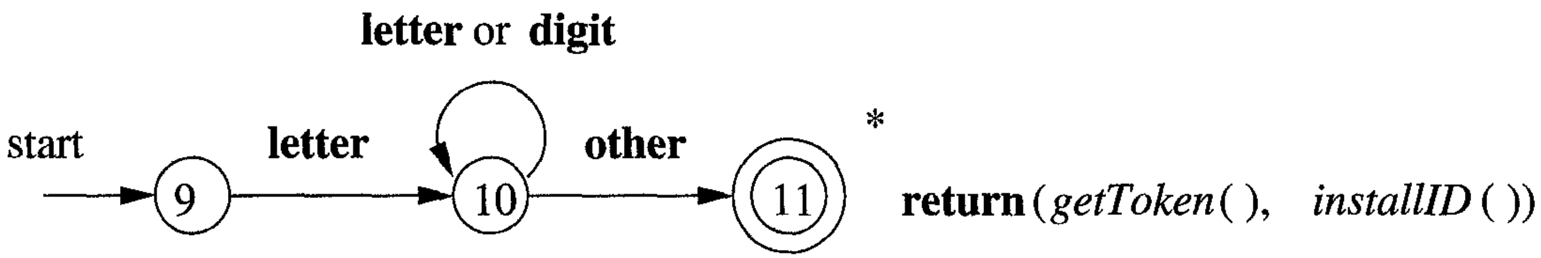


Figure 3.14: A transition diagram for **id**'s and keywords

Transition Diagrams (5)

- Transition Diagram for **number**

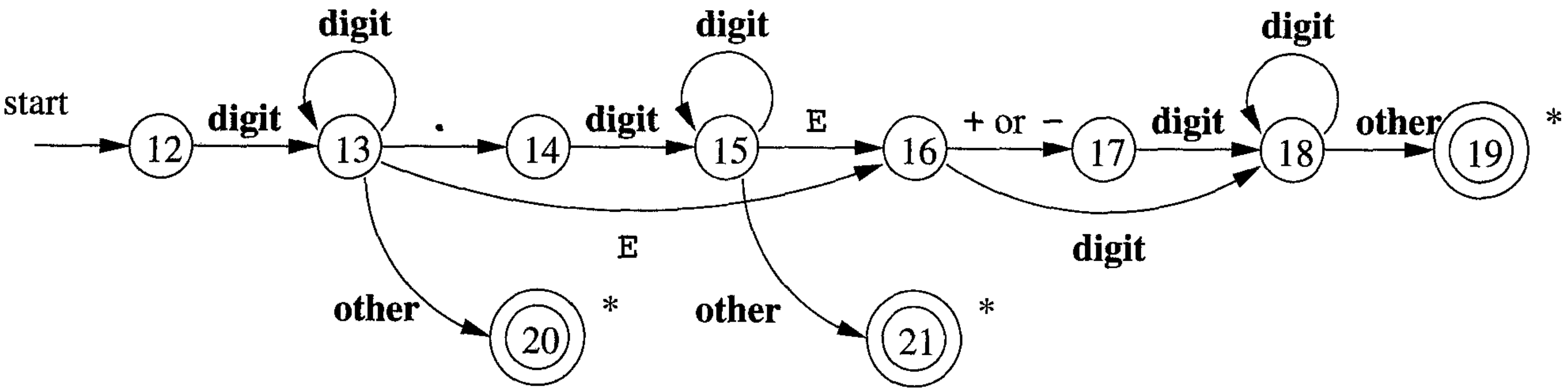


Figure 3.16: A transition diagram for unsigned numbers

Transition Diagrams (6)

- Transition Diagram for **whitespaces**

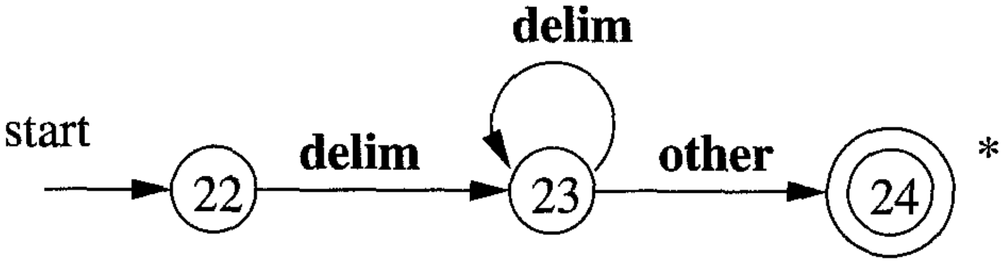
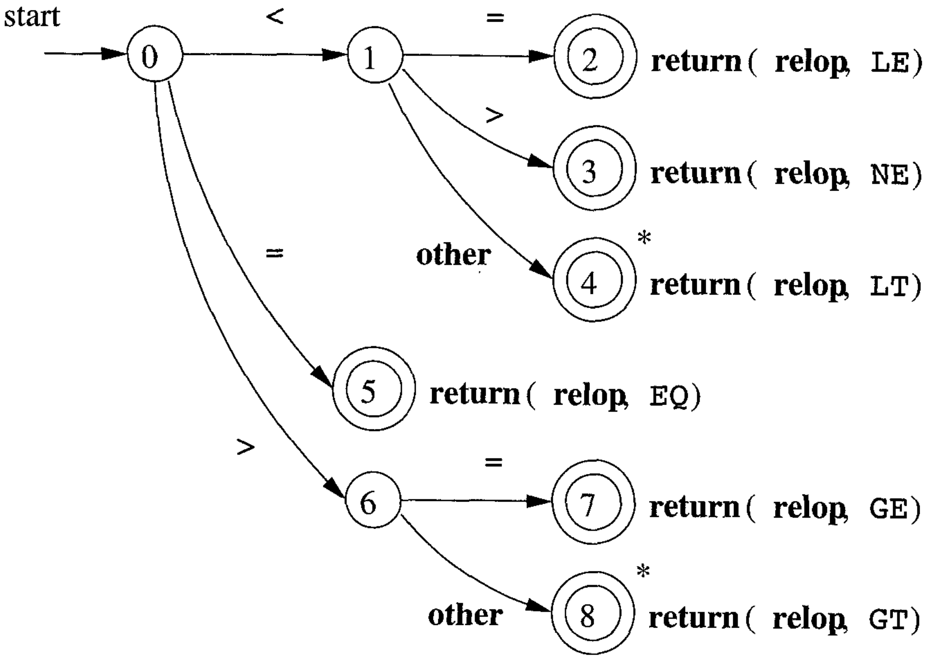


Figure 3.17: A transition diagram for whitespace

Implementation of Transition Diagrams

```
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);
        }
    }
}
```



References

Reference for this topic

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, J D Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Prentice Hall, 2006.

- **Web:** *CS143 Compilers*, Lecture 4, Stanford University.
[https://cs.nyu.edu/courses/Fall12/CSCI-GA.2130-001/lecture4.pdf]

- **Web:** Theory of Computation by Ms. Vandita Grover, Assistant Professor, ANDC, University of Delhi

  [https://sites.google.com/view/courses-vanditagrover/home/ToC-Resources]