Unit - I | Part-2

OVERVIEW OF COMPILATION

Lexical Analysis

- The Role of the Lexical Analyzer
  - ✓ Lexical Analysis Versus Parsing
  - ✓ Tokens, Patterns, and Lexemes
  - ✓ Attributes for Tokens
  - ✓ Lexical Errors
- Regular Grammar and Regular Expression for Common Programming Language Features
  - Specification of Tokens (Using Regular Expressions)
  - Recognition of Tokens
  - Regular Grammar

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Lexical Analysis

- Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- Lexical analysis is also called as scanner.
- We can also produce a lexical analyzer automatically by specifying the *lexeme* patterns to a lexical analyzer generator and compiling those patterns into code that functions as a lexical analyzer.
- A lexical analyzer generator called *Lex* (or Flex in a more recent embodiment).

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.
- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

The Role of the Lexical Analyzer

Interactions between the lexical analyzer and the parser:

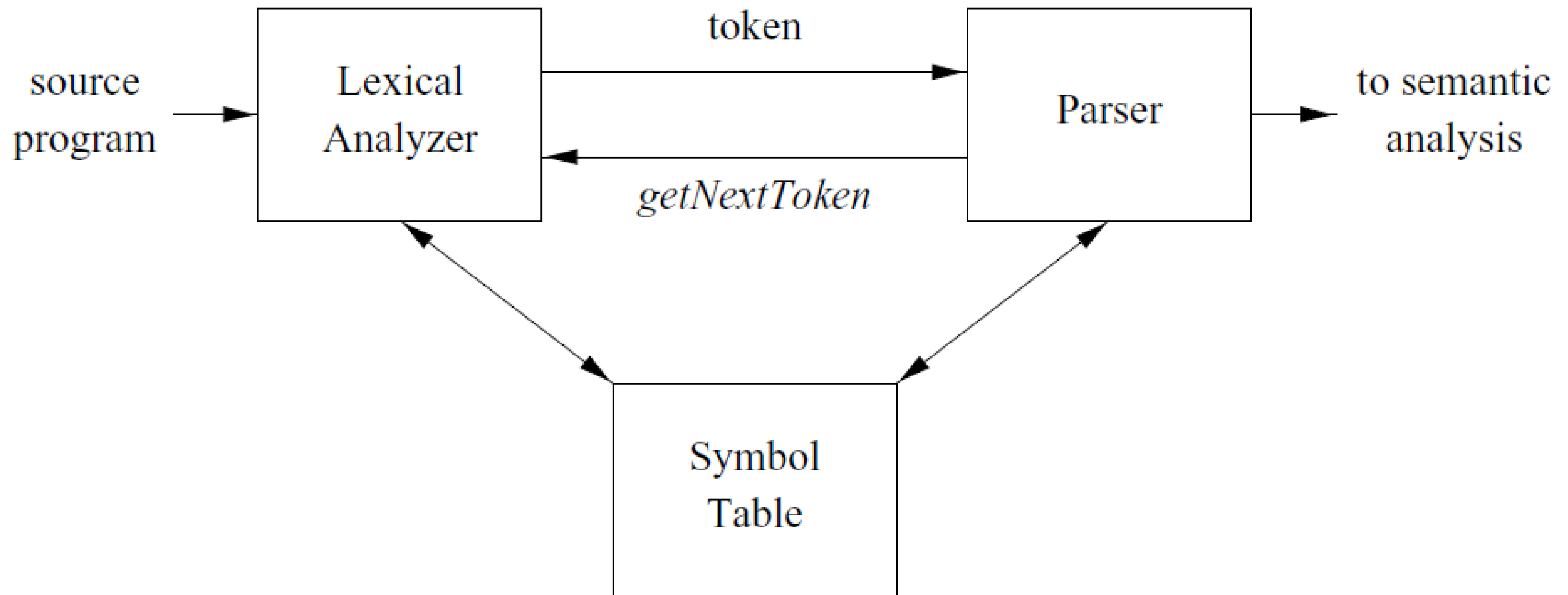


Figure: Interactions between the lexical analyzer and the parser
Source: Compilers: Principles, Techniques and Tools, 2nd Edition, Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffry D. Ullman.

The Role of the Lexical Analyzer

- Commonly, the interaction is implemented by having the parser call the lexical analyzer.
- The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.
- Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.
- One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

The Role of the Lexical Analyzer

- Another task is correlating error messages generated by the compiler with the source program.
- For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

The Role of the Lexical Analyzer

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

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, which produces tokens from the output of the scanner.

Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

- Simplicity of design is the most important consideration.
- Compiler efficiency is improved.
- Compiler portability is enhanced

Tokens, Patterns, and Lexemes

Tokens:

- Smallest logically cohesive sequence of characters of interest in source program
- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
- A token is a pair consisting of token name and an optional attribute value.

Example of tokens:

- Type token (id, num, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Tokens, Patterns, and Lexemes

Example of tokens: cont.

- Single-character operators: =, +, -, >
- Multi-character operators: :=, ==, < >, ->
- Keywords: if, while etc.
- Identifiers: my_variable, flag1, My_Variable
- Numeric constants/literals: 123, 45.67, 8.9e+05
- Character literals:'a', '~', '\'
- String literals: "abcd"

Tokens, Patterns, and Lexemes

Example of non tokens:

- Comments
- Preprocessor directive
- Macros
- Blanks
- *White space:* space, tab, end-of-line
- Newline, . . .

Tokens, Patterns, and Lexemes

Patterns:

- A *pattern* is a description of the form that the *lexemes* of a token may take.
- A *pattern* is described as a rule describing set of *lexemes*.

Examples of Patterns:

- Keyword
- Identifier
- Relational operator
- Symbols (#, $, (,), [,], {, }, :, ; ….)

Tokens, Patterns, and Lexemes

Lexemes:

- A *lexeme* is 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.
- A lexeme is a string of patterns read from the source file that corresponds to a token

Examples of Lexemes:

- char, str, [,], =, “hello”, ; ….

Tokens, Patterns, and Lexemes: *Example*

Source code: char str[] = “hello”;

Pattern	Lexemes	Tokens
keyword	char	CHAR
identifier	str	ID, 1
left bracket	[	LEFT_BRACKET
right bracket	]	RIGHT_BRACKET
operator	=	ASSIG_OP
string	“hello”	LITERAL
symbol	;	SEMI_COLON

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.
- *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.
- Thus, in many cases 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.

Attributes for Tokens

- The token names are basically integer codes represented using symbolic names written in capital letters such as INT, FLOAT, SEMI_COLON etc.
- The attribute values are optional and will not be present for keywords, operators and symbols.

Attributes for Tokens

Example:

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> (**id** or **ID**)

<**assign_op**> (or < **ASSIGN_OP**>)

<**id**, pointer to symbol-table entry for M>

<**mult_op**> (or < **MULT_OP** >)

<**id**, pointer to symbol-table entry for C>

<**exp_op**> (or < **EXP_OP** >)

<**number**, integer value 2>

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string *fi* is encountered for the first time in a C program in the context:

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

  - a lexical analyzer cannot tell whether *fi* is a misspelling of the keyword if or an undeclared function identifier.
  - Since *fi* is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler-probably the parser in this case-handle an error due to transposition of the letters.

Lexical Errors

- However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- The simplest recovery strategy is *"panic mode"* recovery.
- We 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.
- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Lexical Errors

Other possible error-recovery actions are:

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.

Specification of Tokens

Strings and Languages

Symbol:

A symbol is an abstract entity that we shall not define formally.

Examples:

Letters: A to Z (upper case) or a to z (lower case)

Digits: 0 to 9

Special characters

Specification of Tokens

Strings and Languages

Alphabet:

An alphabet is a non-empty finite set of symbols. It is denoted by the symbol Σ (sigma)

Examples:

- $\Sigma = \{0, 1\}$ is the *binary alphabet* which consisting of digits.
- *ASCII* is an important example of an alphabet; it is used in many software systems.
- *Unicode*, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.
- $\Sigma = \{a, b, c\}$ is an alphabet which consisting of letters.

Specification of Tokens

Strings and Languages

String or word:

- A string or word is defined as finite sequence of symbols over an alphabet (Σ).
- In language theory, the terms "sentence" and "word" are often used as synonyms for "string"

Examples:

1. Alphabet $\Sigma = \{a, b\}$

Strings:

w = {a, b, aa, ab, ba, bb, aaa, aab, aba, baa, bbb, ….}

2. $\Sigma = \{0, 1\}$

Strings:

w = {0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 111, ….}

Specification of Tokens

Strings and Languages

String or word:

- The length of a string **s**, usually written |**s**|, is the number of occurrences of symbols in **s**.
- For example, **banana** is a string of length six.
- The *empty string*, denoted ϵ, is the string of length zero.
- 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 = dog and y = house, then xy = doghouse.
- The *empty string* is the identity under concatenation; that is, for any string s, $\epsilon s = s\epsilon = s$.

Specification of Tokens

Strings and Languages

Terms for Parts of Strings:

The following string-related terms are commonly used:

- 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**.
- 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**.

Specification of Tokens

Strings and Languages

Terms for Parts of Strings:

- 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**.
- 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.

Specification of Tokens

Strings and Languages

Language:

- A language is a set of strings of symbols from some one alphabet (Σ).
- A language L is subset of Σ^*
- An empty set ϕ and the set consisting of empty string i.e., $\{\epsilon\}$ are languages and these two are distinct.

Specification of Tokens

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L_1 and L_2	$L_1 \cup L_2 = \{s \mid s \text{ is in } L_1 \text{ or } s \text{ is in } L_2\}$
Concatenation of L_1 and L_2	$L_1L_2 = \{st \mid s \text{ is in } L_1 \text{ and } t \text{ is in } L_2\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Table: Definitions of operations on languages

Specification of Tokens

Operations on Languages

Example:

Let L be the set of letters {A, B, ….., Z, a, b, …..,z} and let D be the set of digits {0, 1, ..,9}

- *L* U *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.
- *LD* is the set of 520 strings of length two, each consisting of one letter followed by one digit.
- L^5 is the set of all 5-letter strings.
- L^* is the set of all strings of letters, including ϵ, the empty string.
- $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
- D^+ is the set of all strings of one or more digits.

Specification of Tokens

Regular Expressions

- The language accepted by Finite Automata (FA) can be easily described by simple expressions called Regular Expressions.
- Regular Expressions are useful for representing certain sets of strings in an algebraic fashion. These describes the languages accepted by Finite Automata (FA).

Specification of Tokens

Regular Expressions

A recursive definition of Regular Expression over an alphabet Σ is as follows:

1. Any terminal symbol '*a*' (i.e., an element of Σ, $a \in \Sigma$), an empty set Φ and empty string ϵ or λ are regular expressions.
2. The *UNION* of two regular expressions R_1 and R_2 written as R_1+R_2 is also a regular expression.
3. The *CONCATENATION* of two regular expressions R_1 and R_2 written as R_1R_2 is also a regular expression.
4. The *CLOSURE* or *ITERATION* (*) of a regular expression R written as R^* is also a regular expression.
5. If R is a regular expression, then (R) is also a regular expression.
6. The regular expressions over an alphabet Σ are precisely those obtained recursively by the applications of the rules 1 to 5 once or several times.

Specification of Tokens

Regular Expressions

Example:

S. No.	Regular Sets	Regular Expressions
1	{101}	101
2	{abba}	abba
3	{01, 10}	01+10
4	{ϵ, ab}	ϵ+ ab
5	{aba, a, b, bba}	aba+a+b+bba
6	{ϵ, 0, 00, 000, …..}	ϵ+0+00+000+….. $= 0^*$
7	{1, 11, 111, …..}	1+11+111+….. $= 11^*$
8	{a, b}	a+b
9	$\{a, b\}^*$	$(a+b)^*$

Specification of Tokens

Regular Expressions

Example:

S. No.	Regular Sets	Regular Expressions
10	L_1 = {The set of all strings of 0's and 1's ending in 00}	***Solution:*** $\Sigma = \{0, 1\}$ $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ $= \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}\ 00$ = **$(0+1)^*00$**
11	L_2 = {The set of all strings of 0's and 1's beginning with 0 and ending with 1}	***Solution:*** $\Sigma = \{0, 1\}$ $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ $= 0\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}\ 1$ = **$0(0+1)^*1$**

Specification of Tokens

Regular Expressions

- 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 $r = s$

Specification of Tokens

Identities or Algebraic Laws of Regular Expressions

The following are the *Identities* or *Algebraic Laws* of Regular Expressions:

I_1: $\Phi + R = R$

I_2: $\Phi R = R\Phi = \Phi$

I_3: $\epsilon R = R\epsilon = R$ // ϵ is the identity for concatenation

I_4: $\epsilon^* = \epsilon$ and $\Phi^* = \epsilon$

I_5: $R + R = R$

I_6: $R^*R^* = R^*$

I_7: $RR^* = R^*R = R^*$

I_8: $(R^*)^* = R^*$

Specification of Tokens

Identities or Algebraic Laws of Regular Expressions

I_{10}: $(PQ)^*P = P(QP)^*$

I_{11}: $(P + Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$

I_{12}: $(P + Q)R = PR + QR$ and $R(P + Q) = RP + RQ$ // Concatenation Distributive

I_{13}: $P + Q = Q + P$ // Commutative

I_{14}: $P+(Q + R) = (P+Q)+ R$ // Associative

I_{15}: $P(QR) = (PQ)R$ // Concatenation Associative

I_{16}: $P(QR) = (PQ)R$ // Concatenation Associative

I_{17}: $R^* = (R+\epsilon)^*$ // ϵ is guaranteed in a closure

I_{18}: $R^{**} = R^*$ // * is idempotent

Here P, Q and R are **Regular Expressions**.

Specification of Tokens

Regular Definitions

- For notational convenience, we may wish to give names to certain regular expressions and use those names in 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:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

Where

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

Specification of Tokens

Regular Definitions

Example 1:

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.

```
letter_ → A | B | … | Z | a | b |…|z | _
digit → 0 | 1 | … | 9
id → letter_(letter_ | digit)*
```

Specification of Tokens

Regular Definitions

Example 2:

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

```
digit → 0|1| … |9
digits → digit digit*
optionalFraction → .digits |ϵ
optionalExponent → (E (+ |-|ϵ)digits)|ϵ
number → digits optionalFraction optionalExponent
```

Specification of Tokens

Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns.

- *One or more instances*: $(r)^+$
  - The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, 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, $r^* = r^+ \mid \epsilon$ and $r+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

Specification of Tokens

Extensions of Regular Expressions

- *Zero or one instance*: r?
  - 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 * and +.
- *Character classes:* [abc]
  - A regular expression $a_1|a_2|..|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2..a_n]$.

Example

```
letter_ → [A-Za-z_]

digit → [0-9]

id → letter_(letter_|digit)*
```

Recognition of Tokens

Outline:

- Recognition of Tokens
  - Transition Diagrams
  - Recognition of Reserved Words and Identifiers
  - Example

Recognition of Tokens

- Consider the following grammar for branching statements:

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt$$
$$\quad | \textbf{ if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\quad | \ \epsilon$$
$$expr \rightarrow term \textbf{ relop } term \ | \ term$$
$$term \rightarrow \textbf{id} \ | \ \textbf{number}$$

For relop, we use the comparison operators of languages like Pascal or SQL, where **=** is "equals" and **< >** is "not equals"

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

Recognition of Tokens

The *patterns* of these *tokens* are described using *regular definitions*, as shown below:

```
digit → [0-9]
digits → digit+
number → digits(. digits)? (E[+-]? digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>
ws → (blank | tab | newline)+ // ws is white space
```

Table: Patterns for tokens

Recognition of Tokens

Lexical Analyzer is summarized in below table which shows, for each lexeme of family of lexemes, which token name is returned to the parser and what attribute value is returned.

Table: Tokens, their 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

Transition Diagrams

- As an intermediate step in the construction of a ***lexical analyzer***, we first convert *patterns* into *stylized flowcharts*, called "***transition diagrams***".
- ***Transition diagrams*** have a collection of nodes or circles, called ***states***. 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.
- States are summarized to know about what characters we have seen between the ***lexemeBegin*** pointer and the ***forward*** pointer.

Transition Diagrams

- ***Edges*** are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state ***s***, and the next input symbol is ***a***, we look for an edge out of state ***s*** labeled by ***a***. If we find such an edge, we advance the ***forward*** pointer and enter the state of the transition diagram to which that edge leads.
- We shall assume that all out ***transition diagrams*** are ***deterministic***, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Transition Diagrams

- Some important conventions about ***transition diagrams*** are:

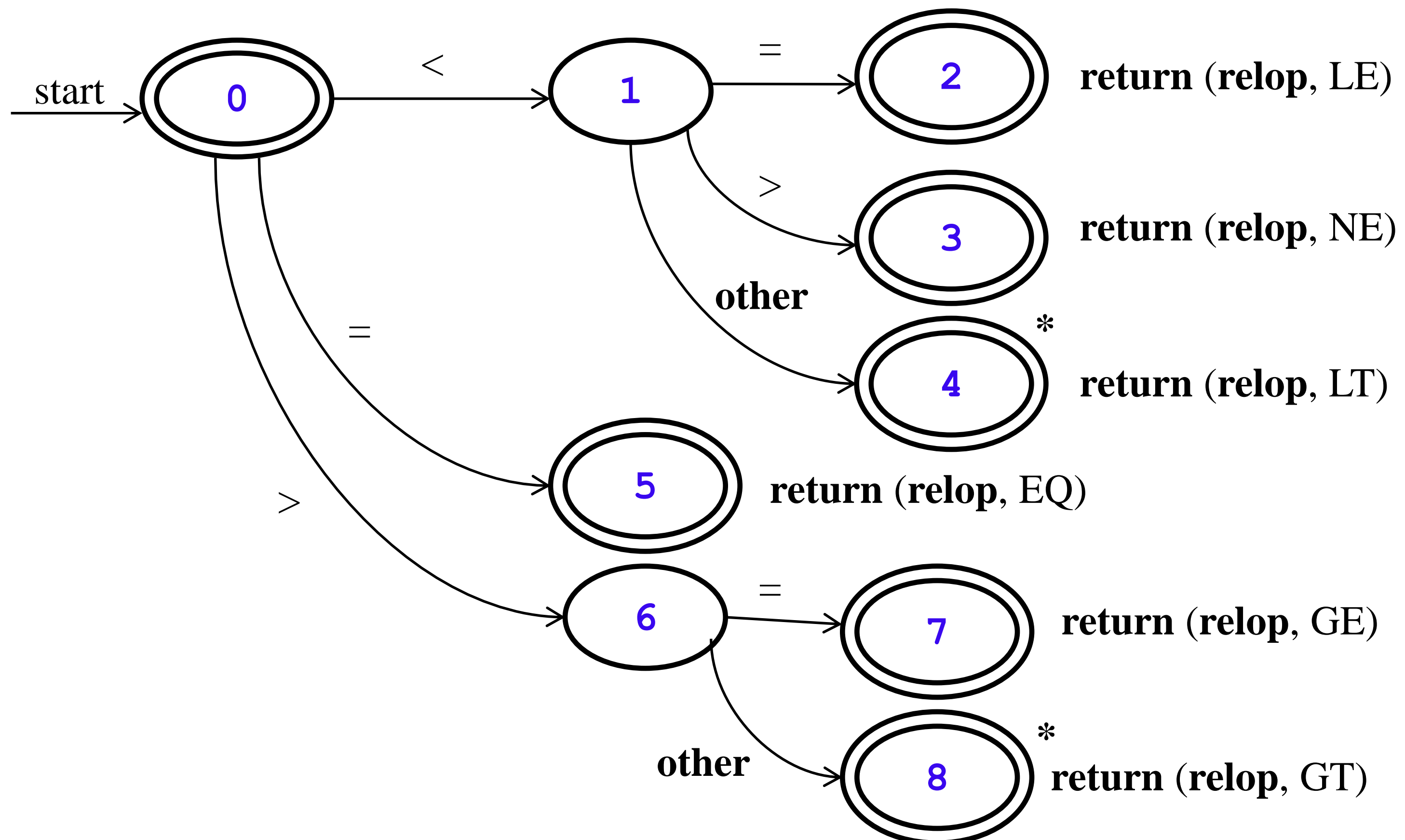
1. Certain states are said to be ***accepting*** or ***final***. These states indicate that a *lexeme* has been found, although the actual lexeme may not consist of all positions between the ***lexemeBegin*** and ***forward*** pointers. We always indicate an *accepting state* by a *double circle*, and 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*.

Transition Diagrams

2. 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*.
3. One state is designated the ***start state***, or ***initial state***, it is 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.

Transition Diagrams

Example: The following *figure* is a *transition diagram* that recognizes the lexemes matching the token ***relop*** (relational operators)



Recognition of Reserved Words and Identifiers

- Recognizing *keywords* and *identifiers* presents a problem. Usually, keywords like *if* or *then* are reserved, so they are not identifiers even though they *look* like identifiers.
- The following figure shows *transition diagram* for *identifiers* and *keywords*:

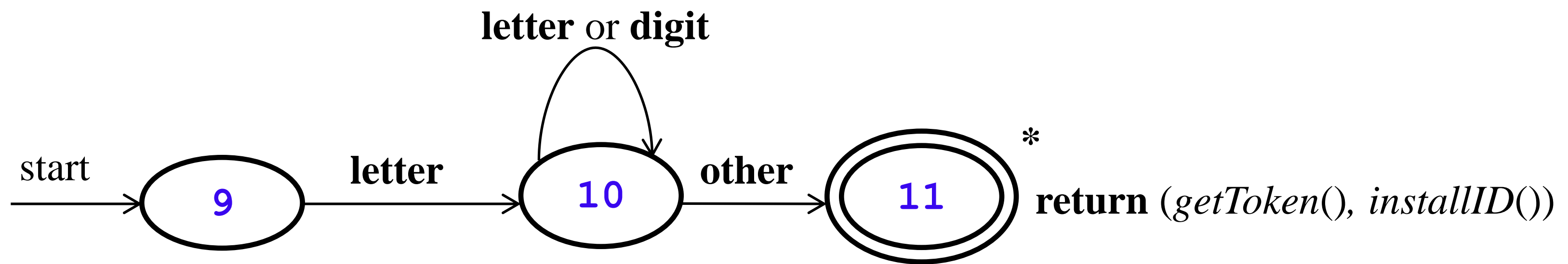


Fig: Transition Diagram for **identifiers** and **keywords**

Recognition of Reserved Words and Identifiers

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially.
2. Create a separate transition diagrams for each keyword; an example for the keyword then is shown in figure:

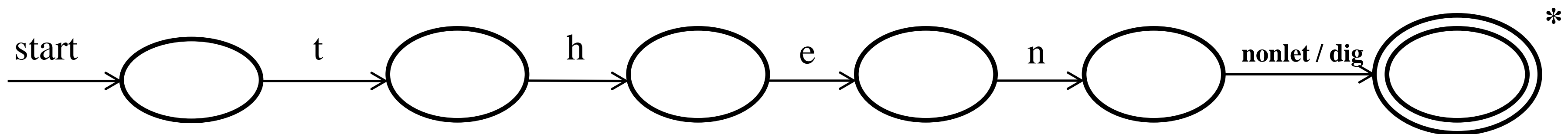


Fig: Hypothetical Transition Diagram for the keyword **then**

Completion of the Running Example

- The *transition diagram* for *identifiers* for token **number** (*unsigned numbers*) is shown in below figure:

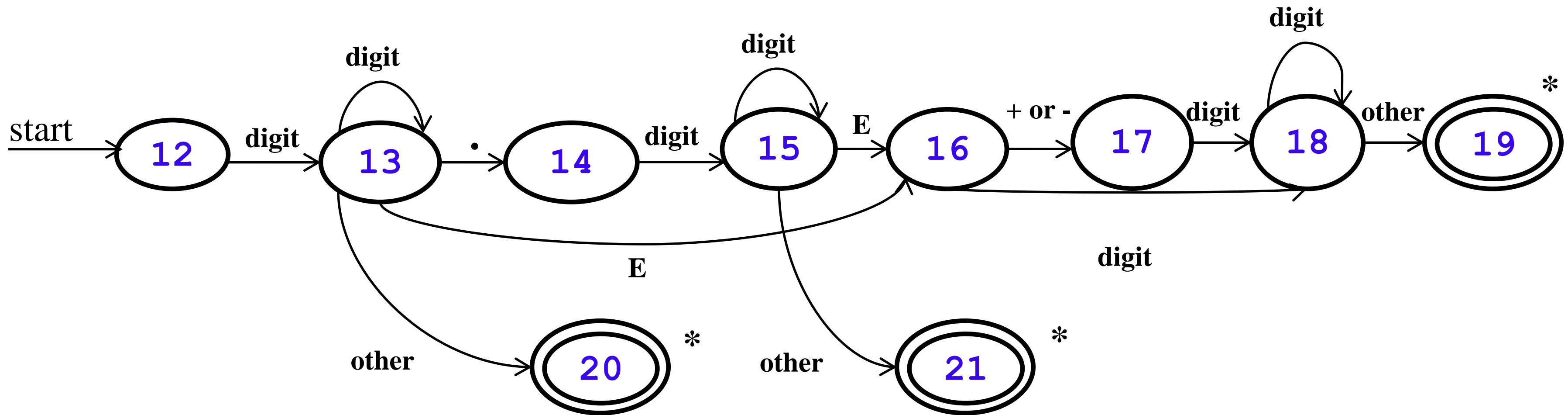


Fig: Transition Diagram for **unsigned numbers**

Completion of the Running Example

- The *transition diagram* for **whitespace** is shown in below figure:

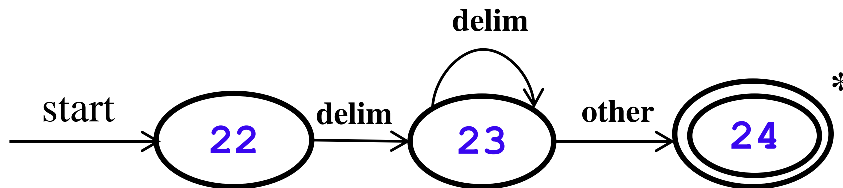


Fig: Transition Diagram for **whitespace**

Definition of Formal Grammar

A **Grammar** or **Formal Grammar G** is a 4-tuple i.e., **G = (V, T, S, P)**.

Where

V: Set of Variables or Non-terminals

T: Set of Terminal symbols

S: Start Variable, $S \in V$

P: Set of Productions

Production Rules: **P:** $\alpha \rightarrow \beta$, where $\alpha \in (V \cup T)^+$ and $\beta \in (V \cup T)^*$

Regular Grammar

A ***Grammar*** **G = (V, T, S, P)** is **right-linear** if each production has one of the following three forms:

1. $A \rightarrow cB$
2. $A \rightarrow c$
3. $A \rightarrow \epsilon$

Where $A, B \in V$ (with $A = B$ allowed) and $c \in T$

Regular Grammar

A ***Grammar*** **G = (V, T, S, P)** is **left-linear** if each production has one of the following three forms:

1. $A \rightarrow Bc$
2. $A \rightarrow c$
3. $A \rightarrow \epsilon$

Where $A, B \in V$ (with $A = B$ allowed) and $c \in T$

Regular Grammar

A **right** or **left-linear** grammar is called a **`regular grammar`**.

Example-1:

$S \rightarrow 0A$

$A \rightarrow 10A \mid \epsilon$

is **right-linear** grammar

Example-2:

$S \rightarrow S10 \mid 0$

is **left-linear** grammar

Regular Grammar: Examples

Example-3:

$G_1 = (\{S\}, \{a, b\}, S, P_1)$

$P_1: S \rightarrow abS \mid a$ is **right-linear** grammar

$G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$

P_2:

$S_2 \rightarrow S_1ab$

$S_1 \rightarrow S_1ab \mid S_2$

$S_2 \rightarrow a$ is **left-linear** grammar

Regular Grammar: Examples

Example-4:

G = ({S, A, B}, {a, b}, S, P)

P:

S $\rightarrow$ **A**

A $\rightarrow$ **aB |ε** // is **right-linear** grammar

B $\rightarrow$ **Ab** // is **left-linear** grammar

This grammar is not a **regular grammar**, but this is a **grammar**.

Regular Grammars and Finite Automata

Regular grammar and **Finite Automata** are equivalent as stated in the following theorem.

Theorem :

A language **L** is regular if and only if it has a regular grammar.

We use the following two lemmas to prove the above theorem.

Regular Grammars and Finite Automata

Lemma 1:

If `L` is a regular language, then `L` is generated by some right-linear grammar.

Lemma 2:

Let `G = (V, T, S, P)` be a right-linear grammar. Then `L(G)` is a regular language.

Construction of Regular Grammar to Finite Automata (FA)

Procedure:

Let **G = (V, T, S, P)** be a **Regular Grammar**. We can construct **DFA M** whose

i) States correspond to Variables

ii) Initial state correspond to **S**

iii) Transitions in **M** correspond to Productions in **P**.

Construction of Regular Grammar to Finite Automata (FA)

Procedure: cont.

If there is a production of the form $A_i \rightarrow a$, the corresponding transition terminates at a new state. This is the unique final state. Thus **DFA M** can be

$M = (\{q_0, q_1, \dots, q_n, q_f\}, \Sigma, \delta, q_0, \{q_f\})$ where δ is defined as:

1. Each production $A_i \rightarrow aA_j$ induces a transition from q_i to q_j with label **a**.
2. Each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label **a**.

L(G) can be given by corresponding **DFA M**.

Construction of Regular Grammar to Finite Automata (FA)

Example Problem:

Construct DFA equivalent to the grammar given below:

$S \rightarrow aS \mid bS \mid aA$

$A \rightarrow bB \mid b$

$B \rightarrow aC$

$C \rightarrow \epsilon$

Construction of Regular Grammar to Finite Automata (FA)

Example Problem: Solution

DFA can be constructed using the following rules:

1. Each production $A_i \rightarrow aA_j$ induces a transition from q_i to q_j with label a.
2. Each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label a.

Construction of Regular Grammar to Finite Automata (FA)

Example Problem: Solution cont.

DFA M = ({S, A, B, C}, Σ, δ, S, {C}) where

Σ = T = {a, b} , F = {C} and δ is given below:

Transitions	Productions
δ(S, a) = {S, A} δ(S, b) = S	S → aS | bS | aA
δ(A, b) = B δ(A, b) = C (for A → b by Rule 2 it is going to final state C.)	A → bB | b
δ(B, a) = C	B → aC
--	C → ϵ

Construction of Regular Grammar to Finite Automata (FA)

Example Problem: Solution cont.

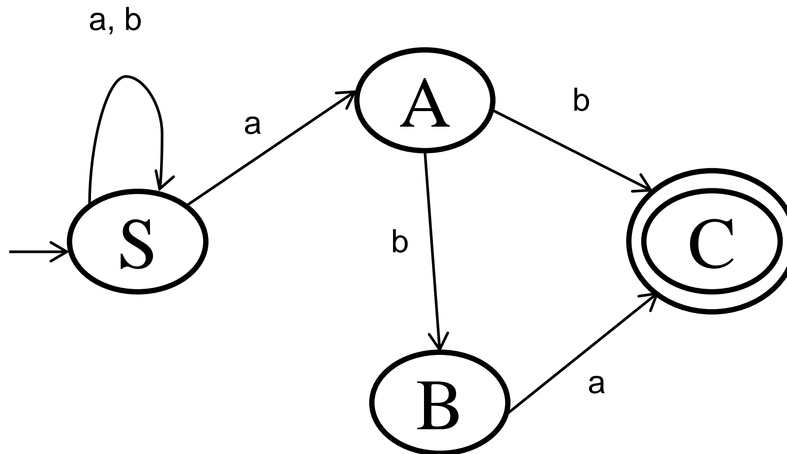


Fig : DFA