Lexical Analysis

Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.

A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

- Scanning
- Tokenization

Token, Pattern and Lexeme

Token

Token is a valid sequence of characters which are given by lexeme. In a programming language,

- keywords,
- constant,
- identifiers,
- numbers,
- operators and
- punctuations symbols

are possible tokens to be identified.

Pattern

Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.

Lexeme

Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a

token.

(eg.) c=a+b*5;

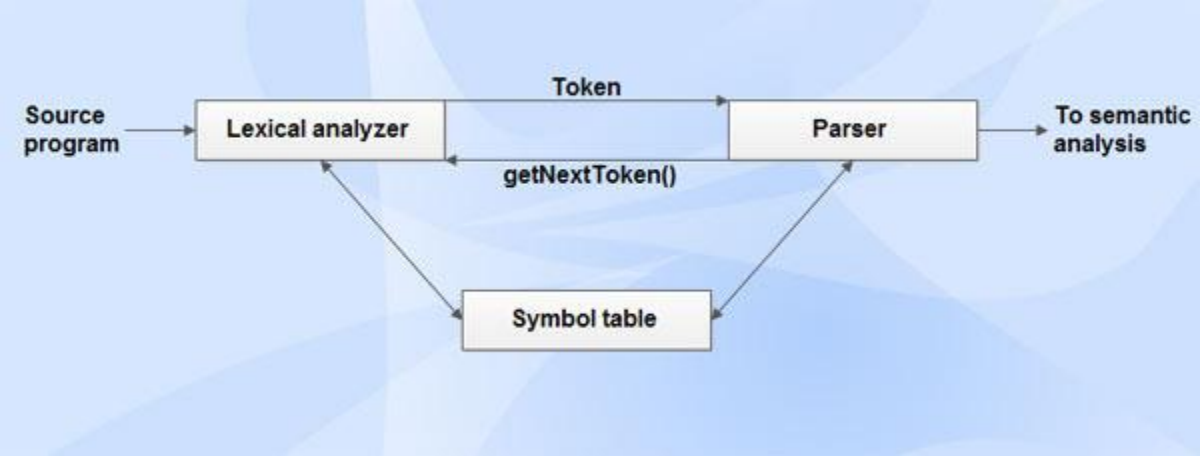
Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol

a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of Lexical Analyzer



Lexical analyzer performs the following tasks:

• Reads the source program, scans the input characters, group them into lexemes and produce the token as output.

• Enters the identified token into the symbol table.

• Strips out white spaces and comments from source program.

• Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.

• Expands the macros if it is found in the source program.

Tasks of lexical analyzer can be divided into two processes:

Scanning: Performs reading of input characters, removal of white spaces and comments.

Lexical Analysis: Produce tokens as the output.

Need of Lexical Analyzer

Simplicity of design of compiler The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.

Compiler efficiency is improved Specialized buffering techniques for reading characters speed up the compiler process.

Compiler portability is enhanced

Running example:

float abs_zero_Kelvin = -273;

Token (also called word)

A string of characters which logically belong together float, identifier, equal, minus, intnum, semicolon Tokens are treated as terminal symbols of the grammar specifying the source language

Pattern

The set of strings for which the same token is produced The pattern is said to match each string in the set

float, l(l+d+_)*, =, -, d+, ;

Lexeme

The sequence of characters matched by a pattern to form the corresponding token

“float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

Issues in Lexical Analysis

Lexical analysis is the process of producing tokens from the source program. It has the following issues:

• Lookahead

• Ambiguities

Lookahead

Lookahead is required to decide when one token will end and the next token will begin. The simple example which has lookahead issues are i *vs. if,* = *vs.* ==. Therefore a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

• Is if it is two variables *i* and *f* or if?

• Is == is two equal signs =, = or ==?

• arr(5, 4) vs. fn(5, 4) *II* in Ada (as array reference syntax and function call syntax are similar.

Hence, the number of lookahead to be considered and a way to describe the lexemes of each token is also needed.

Regular expressions are one of the most popular ways of representing tokens.

Ambiguities

The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

• The longest match is preferred.

• Among rules which matched the same number of characters, the rule given first is preferred.

Lexical Errors

• A character sequence that cannot be scanned into any valid token is a lexical error.

• Lexical errors are uncommon, but they still must be handled by a scanner.

• Misspelling of identifiers, keyword, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery Schemes

• Panic mode recovery

• Local correction

  o Source text is changed around the error point in order to get a correct text.

  o Analyzer will be restarted with the resultant new text as input.

• Global correction

  o It is an enhanced panic mode recovery.

  o Preferred when local correction fails.

Panic mode recovery

In panic mode recovery, unmatched patterns are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

(eg.) For instance 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 *f* iis a misspelling of the keyword if or an undeclared function identifier.

Since *f* i is a valid lexeme for the token **id,** the lexical analyzer will return the token **id** to the parser.

Lexical error handling approaches

Lexical errors can be handled by the following actions:

- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

Languages

Symbol: An abstract entity, not defined
Examples: letters and digits
String: A finite sequence of juxtaposed symbols
abcb, caba are strings over the symbols a,b, and c
|w| is the length of the string w, and is the #symbols in it
 is the empty string and is of length 0
Alphabet: A finite set of symbols
Language: A set of strings of symbols from some alphabet
Φ and {} are languages
The set of palindromes over {0,1} is an infinite language
The set of strings, {01, 10, 111} over {0,1} is a finite language
If Σ is an alphabet, Σ ∗ is the set of all strings over Σ

Example: A *scanner* groups input characters into tokens. For example, if the input is

```
x = x*(b+1);
```

then the scanner generates the following sequence of tokens:

```
id(1)
=
id(1)
*
(
id(2)
+
num(1)
)
;
```

where `id(1)` indicates the identifier with name `x` (a program variable in this case) and `num(1)` indicates the integer `1`. Each time the parser needs a token, it sends a request to the scanner. Then, the scanner reads as many characters from the input stream as it is necessary to construct a single token. The scanner may report an error during scanning (eg, when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the scanner is suspended and returns the token to the parser. The parser will repeatedly call the scanner to read all the tokens from the input stream or until an error is detected (such as a syntax error).

Tokens are typically represented by numbers. For example, the token `*` may be assigned the number 35. Some tokens require some extra information. For example, an identifier is a token (so it is represented by some number) but it is also associated with a string that holds the identifier name. For example, the token `id(x)` is associated with the string, `"x"`. Similarly, the token `num(1)` is associated with the number, 1.

Specification of Tokens

Tokens are specified by patterns, called *regular expressions*. For example, the regular expression `[a-z][a-zA-Z0-9]*` recognizes all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.
Describe the languages denoted by the following regular expressions:

a(a|b)*a
((ε|a)b*)*
(a|b)*a(a|b)(a|b)
a*ba*ba*ba*
(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Answer

String of a's and b's that start and end with a.
String of a's and b's.
String of a's and b's that the character third from the last is a.
String of a's and b's that only contains three b.
String of a's and b's that has a even number of a and b.

Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

By Dr. Sini Anna Alex, RIT, Bangalore

Answer

```
select -> [Ss][Ee][Ll][Ee][Cc][Tt]
```

Write regular definitions for the following languages:

All strings of lowercase letters that contain the five vowels in order.
All strings of lowercase letters in which the letters are in ascending lexicographic order.
Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (")
All strings of a's and b's that do not contain the substring abb.
All strings of a's and b's that do not contain the subsequence abb.

Answer

1、

```
want -> other* a (other|a)* e (other|e)* i (other|i)* o (other|o)* u (other|u)*
other -> [bcdfghjklmnpqrstvwxyz]
```

2、

```
a* b* ... z*
```

3、

```
\/\*([^*"]*|".*"|\*+[^/])*\*\/
```

4

```
b*(a+b?)*
```

5、

```
b* | b*a+ | b*a+ba*
```

Write character classes for the following sets of characters:

The first ten letters (up to "j") in either upper or lower case.
The lowercase consonants.
The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
The characters that can appear at the end of alegitimate English sentence (e.g. , exclamation point) .

Answer

[A-Ja-j]
[bcdfghjklmnpqrstvwxzy]
[0-9a-f]
[.?!]

Note that these regular expressions give all of the following symbols (operator characters) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression "**" matches the string ** . We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Answer

```
\"\\
```

The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, ^[^aeiou]*$ matches any complete line that does not contain a lowercase vowel.

How do you tell which meaning of ^ is intended?

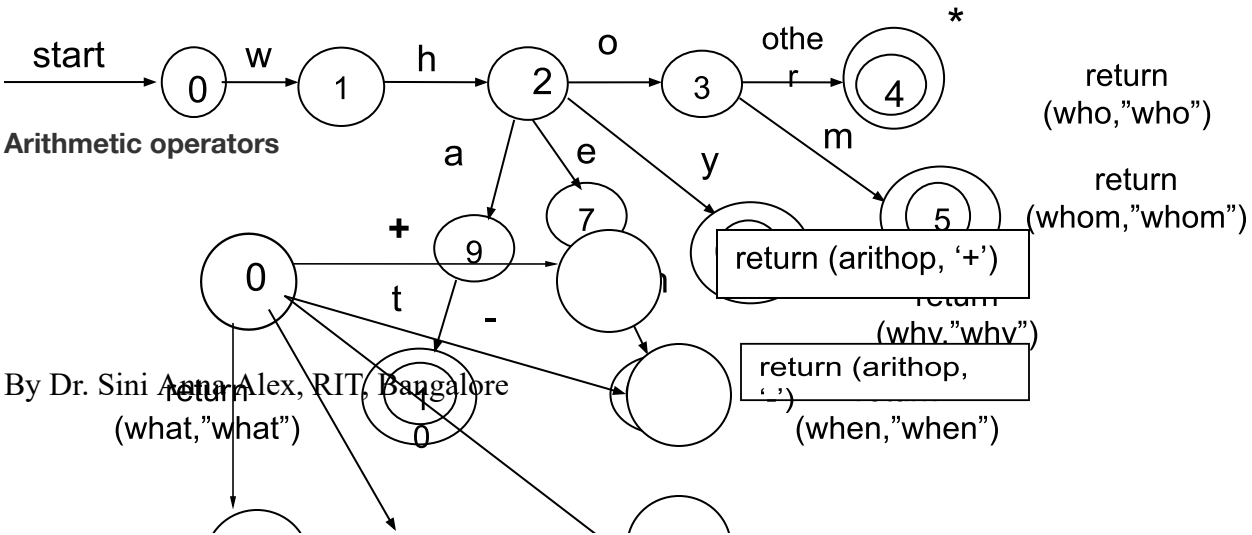
if ^ is in a pair of brakets, and it is the first letter, it means complemented classes, or it means the left end of a line.

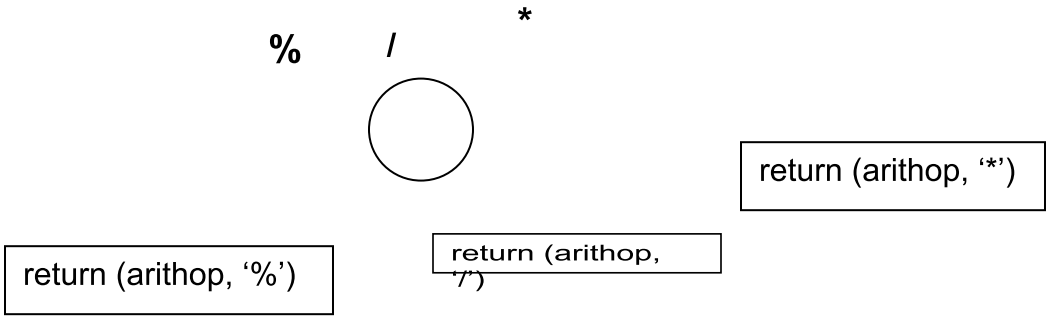
Recognition of Tokens

Construct transition diagram for the following

who, when, what, why, whom
arithmetic operators

who, when, what, why, whom




%
/
*
return (arithop, '%')
return (arithop, '/')
return (arithop, '*')