University of sulaimani
College of science
Department of Computer Science

Compiler

First Phase of Compiler
Lexical Analyzer

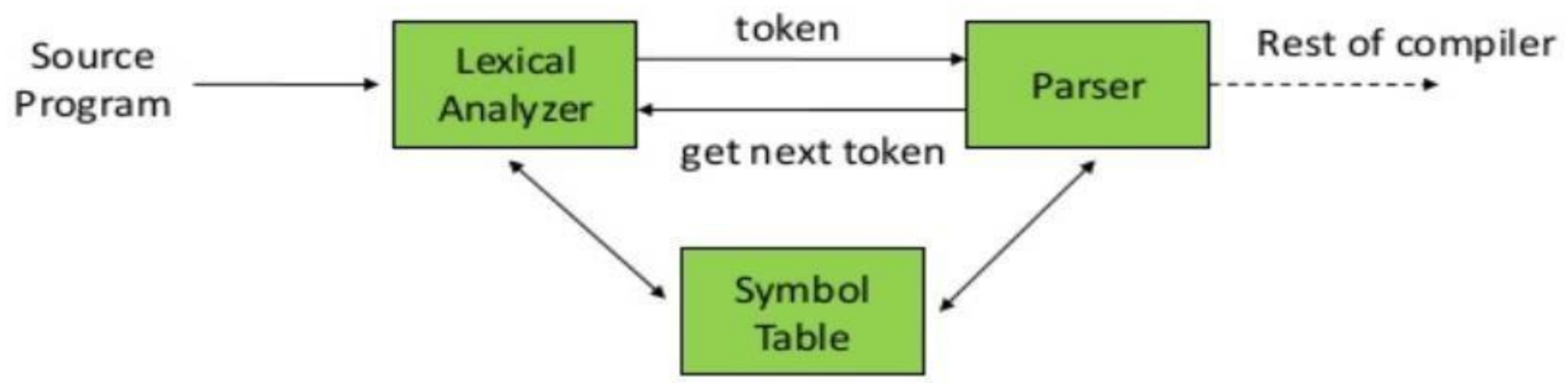
2024-2025

Mzhda Hiwa

Lexical Analyzer

- The lexical analyzer takes a source program as input, and produces a stream of tokens as output. The lexical analyzer might recognize particular instances of tokens Typically :

- Each keyword is a token.
- Each identifier is a token.
- Each constant is a token.
- Each sign, operator is a token.



Terms

Token: A token is a group of characters having collective meaning: typically a word or punctuation mark, separated by a lexical analyzer and passed to a parser.

Lexeme: is the actual character sequence forming a token, as the token is the general class that a lexeme belongs to.

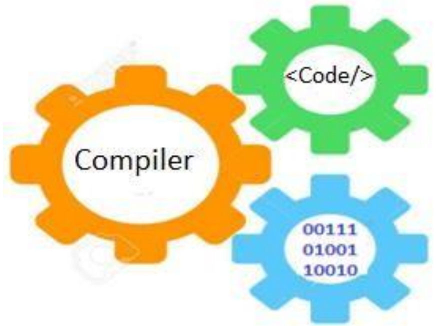
Pattern: A rule that describes the set of strings associated to a token. Expressed as a regular expression and describing how a particular token can be formed.

For example, [A-Za-z][A-Za-z_0-9] *

Example

int sum = 3 + 2 ;

Lexem	Token
int	Keyword
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	Punctuation symbol



Token Classes

1. **Identifier:** variable names, constant ,methods, classes, parameters, user defined data. Identifiers in java must be composed of letters, numbers, the underscore _and the dollar sign $. Identifiers may only begin with a letter, the underscore or a dollar sign.

2. **Keyword:** is any words that have a predefined meaning in the language; programmers cannot use keywords as names for variables, methods, classes, or as any other identifiers.

3. ~~2~~. **Operator** : mathematical & logical operations , +,- ,=, >,*, &&, || .

4. ~~3~~. **Separator**: `[ ]   ( )   { }   ,   ;   .   "`

5. ~~4~~. **Literal** : Boolean, integer, floating point, string, character, true ,false, 2, 6.02e23 , “music”, null.

.

Functions of Lexical Analyzer

1. It produces stream of tokens.

2. It eliminates (comments(single line and multiple line comment) and whitespace (blank, newline, tab, etc…)

3. It generates symbol table which stores the information about identifiers, constants seen in the input.

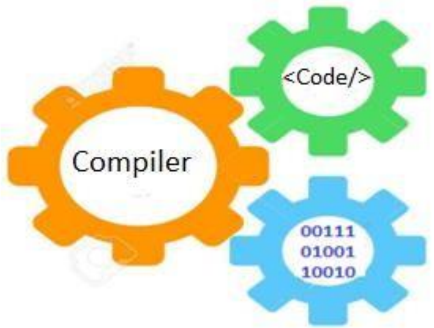
4. It keeps track of line numbers. the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

5. It reports the errors encountered while generating the tokens.

(Scanner)

Error handling in Lexical Analyzer

- The scanner is tasked with determining that the input stream can be divided into valid symbols in the source language, but has no knowledge about which token should come where.
- Few errors can be detected at the lexical level alone because the scanner has a very localized view of the source program without any context.
- The scanner can report about characters that are not valid tokens (e.g., an illegal or unrecognized symbol) and a few other malformed entities (illegal characters within a string constant, unterminated comments, etc.)



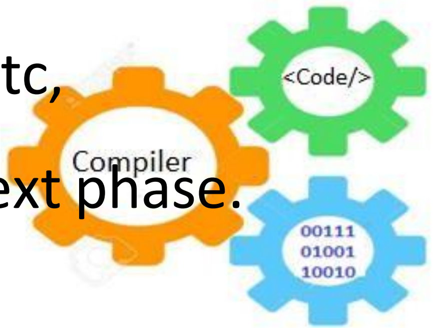
Error handling

```
Example 1 : printf(“Compiler");$
This is a lexical error since an illegal
character $ appears at the end of statement.

Example 2 : This is a comment */ This is an
lexical error since end of comment is
present but beginning is not present.
```

It does not look for or detect garbled sequences, tokens out of place, undeclared identifiers, misspelled keywords...etc,

The syntax analyzer will catch this error later in the next phase.

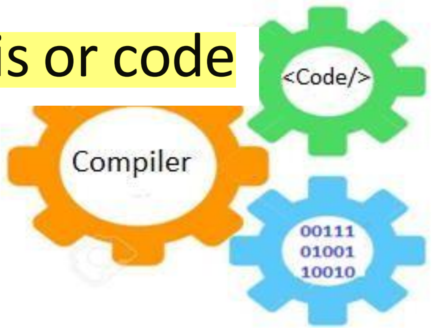


How symbol table is used by Lexical Analyzer

When an identifier in the source program is detected by the lexical analyzer, it is inserted into the symbol table which hold information about the identifier, such as its name and type.

Symbol table in the context of a lexical analyser might contain:

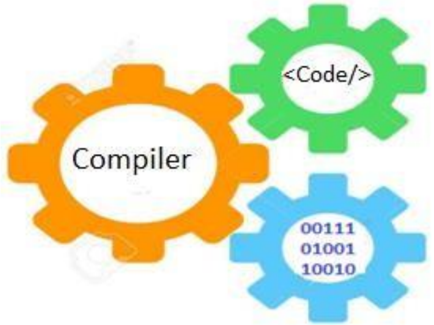
1. Identifiers (variable names, function names, etc.).
2. Associated information like data type, scope, and memory location.
3. Additional attributes required for semantic analysis or code generation phases of compilation.



Symbol table

The remaining phases also insert information about identifiers into the symbol table and then use this information in various ways.

- For example, when doing semantic analysis we need to know what the types of identifiers are.



Reading Ahead

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character >. If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the "greater than or equal to" operator. Otherwise > itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

Reading Ahead

A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer.

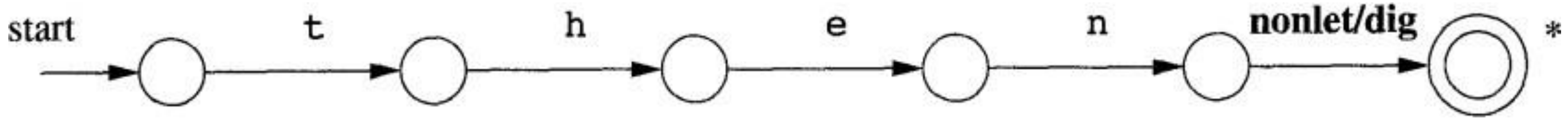
Recognizing Keywords and Identifiers

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.

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

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent . Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is identifier.

2. Create separate transition diagrams for each keyword. such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit”.



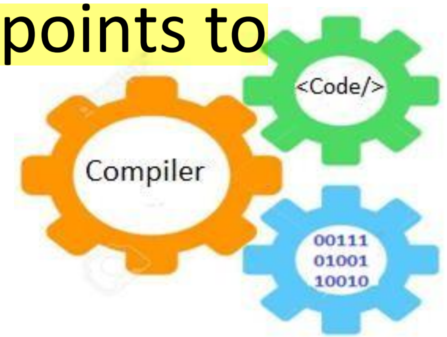
Hypothetical transition diagram for the keyword then

Output of Lexical Analyzer

- The lexical analyzer produces as output a token of the form (token-name, attribute-value).

- The first component token-name is an abstract symbol that is used during syntax analysis,

- and the second component attribute-value points to an entry in the symbol table for this token.



Example

[illegible]

- Suppose a source program contains the assignment statement

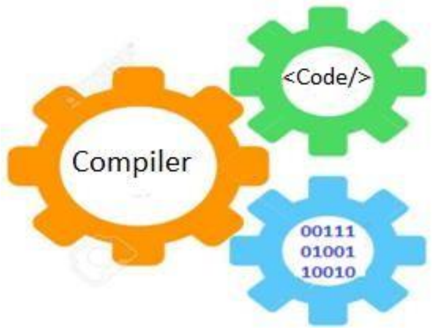
  position = initial + rate * 60

- **position** is a lexeme that would be mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position.

- The assignment symbol = is a lexeme that is mapped in- to the token (=).Since this token needs no attribute-value

- **initial** is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial.

- 60 is a lexeme that is mapped into the token (60).



Token names and values

- So, this is the representation of the whole statement after lexical analysis

position = initial + rate * 60

$$\begin{tikzcd}
\boxed{\text{Lexical analyzer}} \arrow[d] \\
\text{(id,1) (=) (id, 2) (+) (id, 3) (*) (60)} \arrow[d] \\
\boxed{\text{Parser}}
\end{tikzcd}$$

Simple Example of Symbol table

```
Lexeme: variable_name
Token Type: IDENTIFIER
Attribute:<symbol_table_entry_ptr>
```

- **Lexeme**: "variable_name" is the actual string encountered in the source code.
- **Token Type**: Indicates the type of token associated with the lexeme. In this case, it's an identifier.
- **Attribute**: Points to the symbol table entry associated with the lexeme. This pointer allows the lexical analyzer to access additional information about the identifier, such as its type, scope, memory address, etc.

Implementing Lexical Analyzer

Since the lexical structure of *every* programming language can be specified by a regular language, a common way to implement a lexical analyzer is to:

- Specify **regular expressions** for all of the kinds of tokens in the language. Then, use the alternation operator to create a single regular expression that recognizes the language of all valid tokens.
- Convert the overall **regular expression** specifying all possible tokens into a **deterministic finite automaton (DFA).**
- Translate the **DFA** into a program that simulates the **DFA**. This program is the lexical analyzer.

Homework

1. List elements of each token types:

a. Keyword (all keyword in java)
b. Identifier (only write role)
c. Separator   [ ]   ( )   { }   ,   ;   .   "
d. Operator

2. Write regular expression for keyword and identifiers?

3. What are (Lex or JFlex)?

4. In how many ways we can implement Lexical Analyzer?