Lexical Analysis

Table of Contents

[Lexical Errors 2](#_Toc138878996)

[Input Buffering 3](#_Toc138878997)

[Tokens, Patterns and Lexemes 4](#_Toc138878998)

[Languages 4](#_Toc138878999)

[Grammar 5](#_Toc138879000)

**Lexical Analysis** is the process of converting high-level inputs, such as the code we write, into a series of tokens. This process is handled using a **lexical analyser**. A lexical analyser is the first section of a compiler.

The lexical analyser creates the tokens and passes them to the next part of the compiler, the **syntax analyser**. This can be done in two ways:

1. **Batch** – The lexical analyser generates all of the tokens and passes them at once to the syntax analyser.
2. **Iterative** – The syntax analyser asks for tokens as they are required and the lexical analyser processes the input to create the next token.

The objective of a lexical analyser is to be **simple**, **efficient** and **portable**.

## Lexical Errors

Misspelled identifiers, keywords and operators are examples of errors that will be caught by the lexical analyser. The analyser will be unable to generate tokens due to such errors. There are a few strategies discussed below which can be used to correct these errors, but automatic correction of such errors is not practically feasible, which is why we don’t ever see code that corrects itself.

A few possible (hypothetical) error recovery strategies are:

1. **Panic Mode** – If the analyser finds an error, it starts discarding characters one by one until it either reaches a point where the error is fixed or there is no more input to process.
2. **Delete** a single character in the hope that there was an extra character added by mistake.
3. **Insert** a missing character in the hope that a character was missed.
4. **Replace** one incorrect character.
5. **Transpose** two adjacent characters.

It should be obvious why none of these strategies are likely to work well practically.

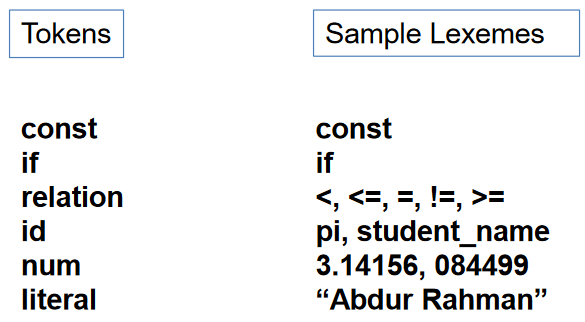
## Input Buffering

If the lexical analyser is reading each character of the input one by one, it is a very slow process since we need to make repeated read operations on the hard drive. Instead, it is faster to use an **input buffer** to store some of the input.

However, there is one issue with using a buffer. As the syntax analyser requests tokens, it is possible that it will realize that it asked for some token too soon and need to **backtrack**. One way to deal with this is for the syntax analyser itself to have another buffer. Another way is to let the lexical analyser ‘**unread**’ a token. This second approach might cause an issue where the **last token** in the buffer was passed, the next set of tokens loaded into the buffer after which there is a need to unread. Since a new set of tokens has been loaded, the token which needs to be unread is no longer available. To deal with this we can use two input buffers in a **cyclic manner**.

## Tokens, Patterns and Lexemes

A **token** is actually a category of inputs. Which category some input falls into is determined using a regular expression, which looks for a **pattern**. A specific instance of a token is called a **lexeme**.



## Languages

A **language** is a finite set of strings over some finite set of alphabets. Computers languages are finite sets on which we can perform mathematical set operations. Such languages can be described using regular expressions.

On the other hand, we have **natural languages** which are spoken languages like English or French. These languages are extremely complicated and it does not seem possible to specify all the rules of syntax for these languages.

If a language can be specified by a well-defined set of rules of syntax, it is called a **formal language**. The rules of syntax is called the **grammar**.

## Grammar

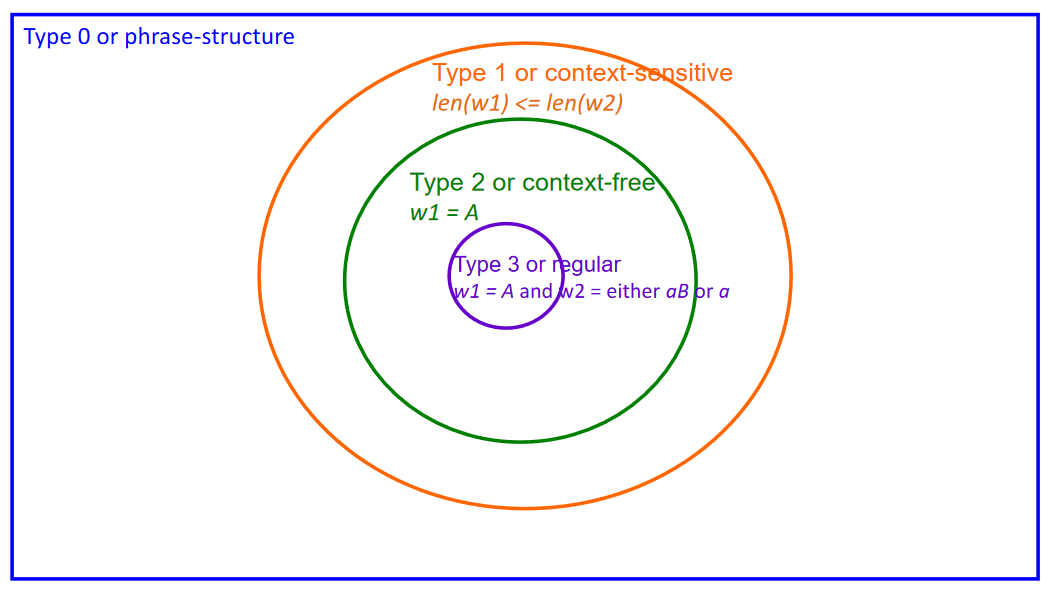
A **Phrase-Structure Grammar** is defined as , where is the **vocabulary**, is the subset of that contains **terminal elements**, is the **start symbol** and is the set of **production rules**. **Non-terminal** elements are not shown directly in but are given by . Every production rule must have at least one non-terminal on the left-hand side.

Phrase-Structure Grammars can be divided into one of four classes based on the type of production rules, , that it has:

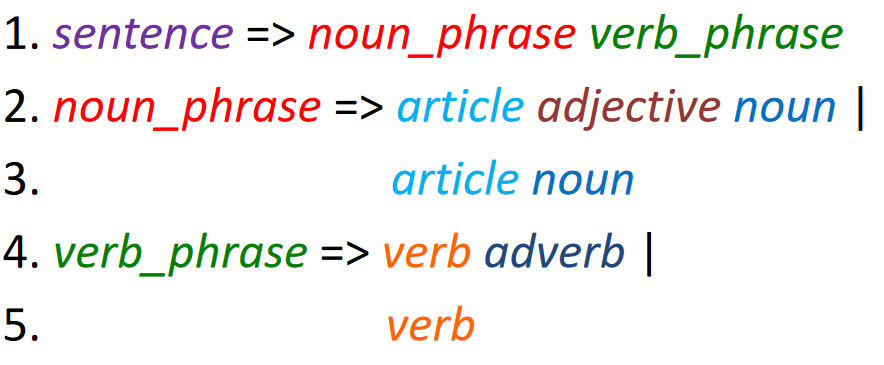
1. **Type 0** – No restrictions.
2. **Type 1** – length of length of
3. **Type 2** -
4. **Type 3** - and or

Here, and are non-terminals while is a terminal.

Each of these types are better known by other names, as shown in the diagram below:

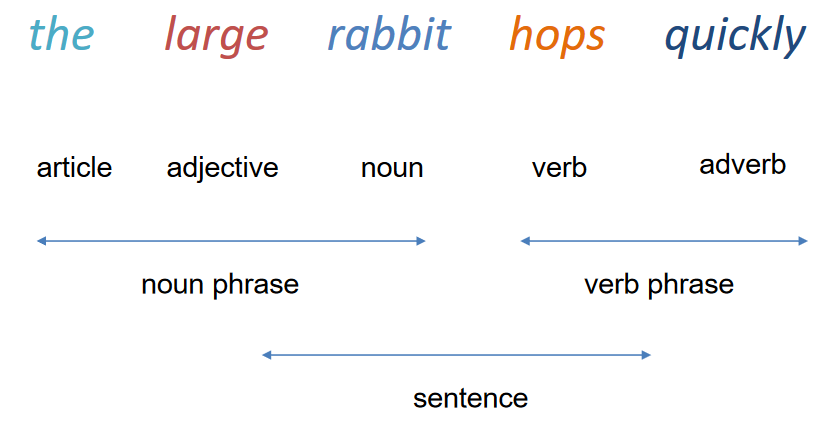


A sample grammar could be as follows:



Thus, a given sentence is divided into two parts, a noun section and a verb section. The noun section will consist of either an article followed by an adjective followed by a verb, or an article followed by a noun. The verb section will consist of a verb followed by an adverb or a verb alone.

An example of a sentence that follows this grammar is as follows:



The grammar would **parse** this sentence using a **parse tree**:

