**Introduction to Grammars**

Grammars denote syntactical rules for conversation in natural languages. Linguistics have attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.

The theory of formal languages finds its applicability extensively in the fields of Computer Science. Noam Chomsky gave a mathematical model of grammar in 1956 which is effective for writing computer languages.

**Grammar**

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where −

 **N** or **V*N*** is a set of variables or non-terminal symbols.

 **T** or **Σ** is a set of Terminal symbols.

 **S** is a special variable called the Start symbol, S ∈ N

 **P** is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are strings on V*N* ∪ Σ and least one symbol of α belongs to VN.

**Example**

Grammar G1 −

({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})

Here,

 **S, A,** and **B** are Non-terminal symbols;

 **a** and **b** are Terminal symbols

 **S** is the Start symbol, S ∈ N

 Productions, **P : S → AB, A → a, B → b**

**Finite Automata:**

Finite Automata(FA) is the simplest machine to recognize patterns.

A Finite Automata consists of the following :

Q : Finite set of states.

Σ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

Formal specification of machine is { Q, Σ, q, F, δ }.

**FA is characterized into two types:**

**1) Deterministic Finite Automata (DFA)**

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. ( Symbols which machine takes as input )

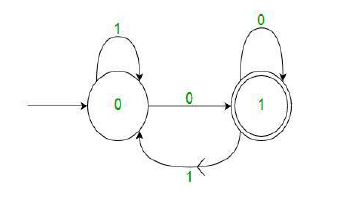
q : Initial state. ( Starting state of a machine )

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.



One important thing to note is, there can be many possible DFAs for a pattern. A DFA with minimum number of states is generally preferred.

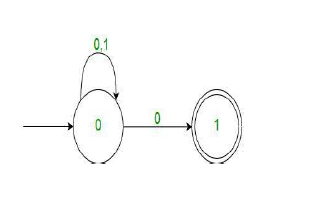
**2) Nondeterministic Finite Automata(NFA)** NFA is similar to DFA except following additional features: 1. Null (or ε) move is allowed i.e., it can move forward without reading symbols. 2. Ability to transmit to any number of states for a particular input. However, these above features don’t add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to above additional features, NFA has a different transition function, rest is same as DFA.

δ: Transition Function

δ: Q X (Σ U ϵ ) --> 2 ^ Q.

As you can see in transition function is for any input including null (or ε), NFA can go to any state number of states. For example, below is a NFA for above problem



One important thing to note is, in NFA, if any path for an input string leads to a final state, then the input string accepted. For example, in above NFA, there are multiple paths for input string “00”. Since, one of the paths leads to a final state, “00” is accepted by above NFA.

**COMPILER BOOTSTRAPPING**

In computer science, **bootstrapping** is the process of writing a compiler (or assembler) in the source programming language that it intends to compile. Applying this technique leads to a self-hosting compiler. An initial minimal core version of the compiler is generated in a different language (which could be assembly language); from that point, successive expanded versions of the compiler are run using the minimal core of the language.

Many compilers for many programming languages are bootstrapped, including compilers for BASIC, Algol, C, D, Pascal, PL/I, Factor, Haskell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Go, Java, Rust, Python, Scala, Nim, Eiffel, and more.

**Advantages**

Bootstrapping a compiler has the following advantages:

● it is a non-trivial test of the language being compiled, and as such is a form of dogfooding.

● compiler developers and bug reporting part of the community only need to know the language being compiled.

● compiler development can be done in the higher level language being compiled.

● improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.

● it is a comprehensive consistency check as it should be able to reproduce its own object code.

**Lexical Analysis**

It can Be defined as :

• A lexical analyzer is a patter matcher.

• A lexical analyzer recognizes strings of characters as tokens.

• Lexical analyzers (scanners) extract lexemes (tokens) from a given input string.

• Lexical analyzers skip comments and blanks.

**Basic Terminologies**

**What’s a lexeme?**

A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

**What’s a token?**

Tokens in compiler design are the sequence of characters which represents a unit of information in the source program.

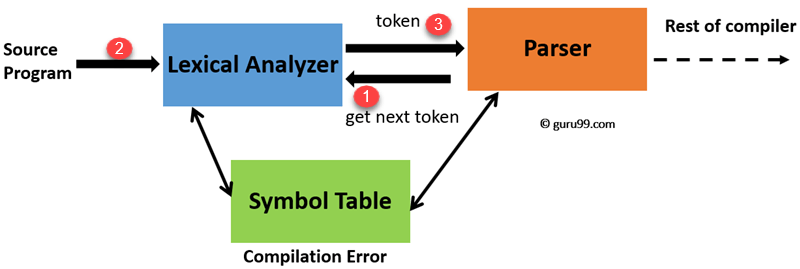
### What is Pattern?

A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

## Lexical Analyzer Architecture: How tokens are recognized

The main task of lexical analysis is to read input characters in the code and produce tokens.

Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how recognition of tokens in compiler design works-



Lexical Analyzer Architecture

1. “Get next token” is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

**Roles of the Lexical analyzer**

Lexical analyzer performs below given tasks:

* Helps to identify token into the symbol table
* Removes white spaces and comments from the source program
* Correlates error messages with the source program
* Helps you to expands the macros if it is found in the source program
* Read input characters from the source program

**Example of Lexical Analysis, Tokens, Non-Tokens**

* Consider the following code that is fed to Lexical Analyzer

#include <stdio.h>

int maximum(int x, int y) {

// This will compare 2 numbers

if (x > y)

return x;

else {

return y;

}

}

**Examples of Tokens created**

|  |  |
| --- | --- |
| **Lexeme** | **Token** |
| int | Keyword |
| maximum | Identifier |
| ( | Operator |
| int | Keyword |
| x | Identifier |
| , | Operator |
| int | Keyword |
| Y | Identifier |
| ) | Operator |
| { | Operator |
| If | Keyword |

**Examples of Nontokens**

|  |  |
| --- | --- |
| **Type** | **Examples** |
| Comment | // This will compare 2 numbers |
| Pre-processor directive | #include <stdio.h> |
| Pre-processor directive | #define NUMS 8,9 |
| Macro | NUMS |
| Whitespace | /n /b /t |

**Lexical Errors**

A character sequence which is not possible to scan into any valid token is a lexical error. Important facts about the lexical error:

* Lexical errors are not very common, but it should be managed by a scanner
* Misspelling of identifiers, operators, keyword are considered as lexical errors
* Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

**Error Recovery in Lexical Analyzer**

Here, are a few most common error recovery techniques:

* Removes one character from the remaining input
* In the panic mode, the successive characters are always ignored until we reach a well-formed token
* By inserting the missing character into the remaining input
* Replace a character with another character
* Transpose two serial characters

**Advantages of Lexical analysis**

* Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer’s code to create a compiled binary executable code
* It is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
* A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task

**Disadvantage of Lexical analysis**

* You need to spend significant time reading the source program and partitioning it in the form of tokens
* Some regular expressions are quite difficult to understand compared to PEG or EBNF rules
* More effort is needed to develop and debug the lexer and its token descriptions
* Additional runtime overhead is required to generate the lexer tables and construct the tokens

Online Notes

<https://www.tutorialspoint.com/automata_theory/automata_theory_tutorial.pdf>

<https://www.vssut.ac.in/lecture_notes/lecture1423726104.pdf>

<https://www.iitg.ac.in/dgoswami/Flat-Notes.pdf>

<https://people.cs.vt.edu/prsardar/classes/cs3304-Spr19/lectures/CS3304-9-LanguageSyntax-2.pdf>

<https://www2.cs.sfu.ca/CourseCentral/383/dma/notes/lecture_sep29.pdf>

<https://www.cs.york.ac.uk/fp/lsa/lectures/slideshow1.pdf>

[Lexical Analysis (Analyzer) in Compiler Design with Example (guru99.com)](https://www.guru99.com/compiler-design-lexical-analysis.html)

Video links

<https://www.youtube.com/watch?v=BgNdtk9h8Ok>

<https://www.youtube.com/watch?v=LjB7ffXWzD0>

<https://www.youtube.com/watch?v=WccZQSERfCM>

<https://www.youtube.com/watch?v=Qa6csfkK7_I>

<https://www.youtube.com/watch?v=eqCkkC9A0Q4&list=PLEbnTDJUr_IdM___FmDFBJBz0zCsOFxfK>

<https://www.youtube.com/watch?v=58N2N7zJGrQ&list=PLBlnK6fEyqRgp46KUv4ZY69yXmpwKOIev>