⇨ **Formal Systems**: Formal Systems play an important role in computer science, linguistics, and logic. What are they? Can they be used to model human thought.
A formal system consists of a language over some alphabet of symbols together with (axioms and) inference rules that distinguish some of the strings in the language as theorems.

A formal system is broadly defined as any well-defined system of abstract thought based on the model of mathematics. The entailment of the system by its logical foundation is what distinguishes a formal system from others which may have some basis in an abstract model. Often the formal system will be the basis for or even identified with a larger theory or field (e.g. Euclidean geometry) consistent with the usage in modern mathematics such as model theory. A formal system need not be mathematical as such; for example, Spinoza's Ethics imitates the form of Euclid's Elements.

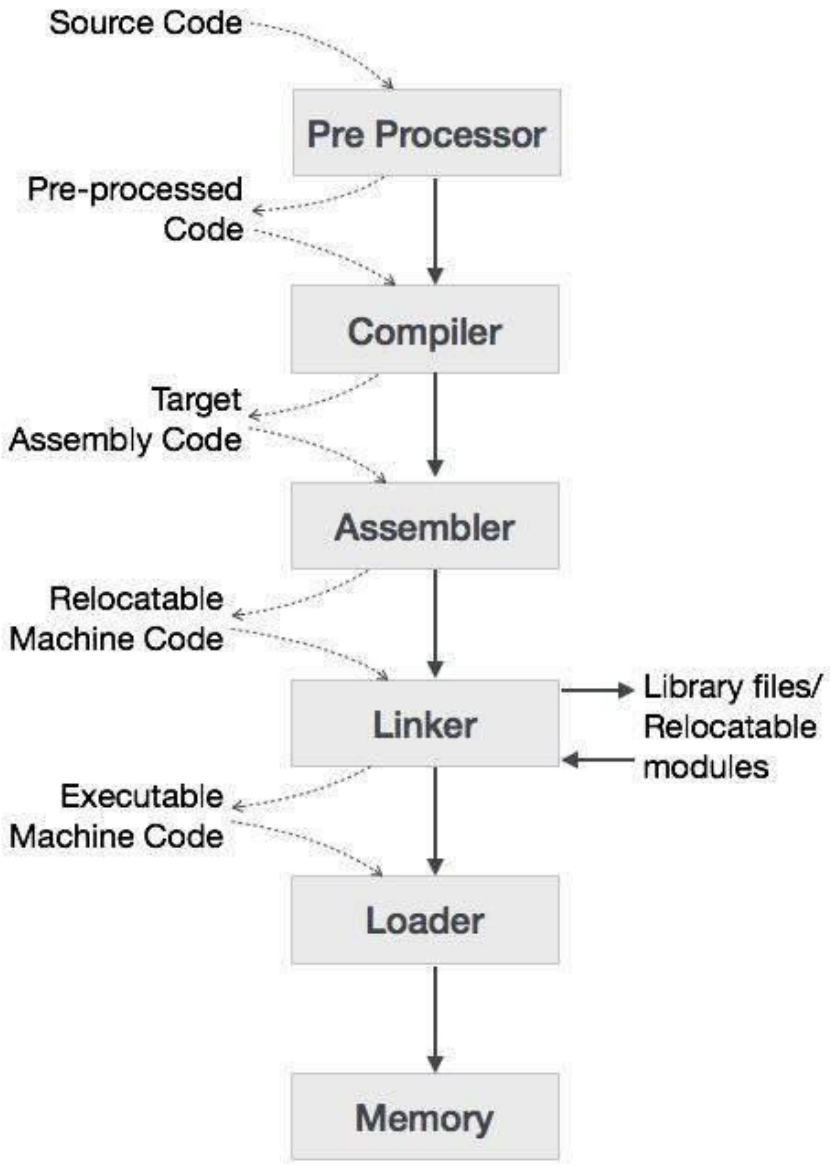
Each formal system has a formal language, which is composed by primitive symbols. These symbols act on certain rules of formation and are developed by inference from a set of axioms. The system thus consists of any number of formulas built up through finite combinations of the primitive symbols—combinations that are formed from the axioms in accordance with the stated rules.

Formal systems in mathematics consist of the following elements:

- A finite set of symbols (i.e. the alphabet), that can be used for constructing formulas (i.e. finite strings of symbols).
- A grammar, which tells how well-formed formulas (abbreviated wff) are constructed out of the symbols in the alphabet. It is usually required that there be a decision procedure for deciding whether a formula is well formed or not.
- A set of axioms or axiom schemata: each axiom must be a wff.
- A set of inference rules.
- A formal system is said to be recursive (i.e. effective) or recursively enumerable if the set of axioms and the set of inference rules are decidable sets or semi decidable sets, respectively.

Language Processing System

The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. Whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Cross-compiler

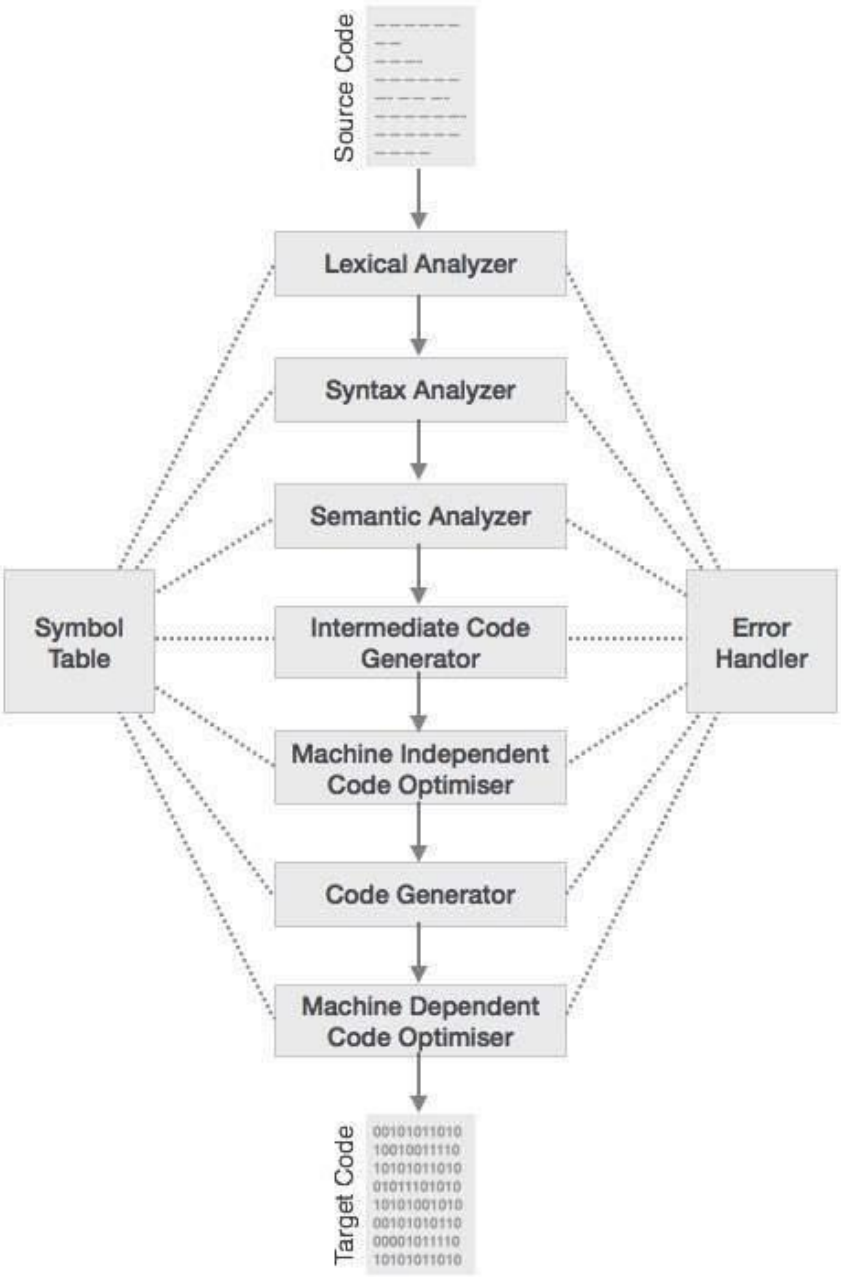
A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

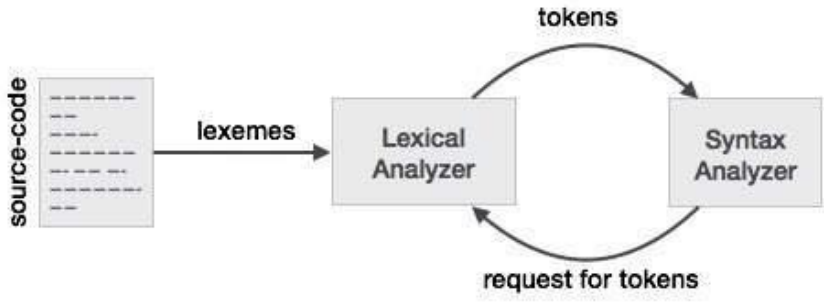
Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Compiler Design - 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.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).
```

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Regular Expressions

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as

  L U M = {s | s is in L or s is in M}

- Concatenation of two languages L and M is written as

  LM = {st | s is in L and t is in M}

- The Kleene Closure of a language L is written as

  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are left associative
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.

  i.e., it can generate { e, x, xx, xxx, xxxx, … }

- x+ means one or more occurrence of x.

  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*

- x? means at most one occurrence of x

  i.e., it can generate either {x} or {e}.

  [a-z] is all lower-case alphabets of English language.

  [A-Z] is all upper-case alphabets of English language.

  [0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = $(sign)^{?}(digit)^{+}$

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Compiler Design - Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on-terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, L = { w | w = w^R } is not a regular language. But it can be described by means of CFG, as illustrated below:

```
G = ( V, Σ, P, S )
```

Where:

```
V = { Q, Z, N }
Σ = { 0, 1 }
P = { Q → Z | Q → N | Q → ε | Z → 0Q0 | N → 1Q1 }
S = { Q }
```

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Limitations of Syntax Analyzers

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks:

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.

Compiler Design - Semantic Analysis

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

```
E → E + T
```

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

```
CFG + semantic rules = Syntax Directed Definitions
```

For example:

```
int a = "value";
```

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking
-

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Passes of Compiler:

Each of compiler tasks described previously (in Compiler Structure) is a phase. Phases can be organized into a number of passes.

– a pass consists of one or more phases acting on some representation of the complete program.

– representations produced between source and target are Intermediate Representations (IRs)

Single Pass Compilers:

- one pass compilers very common because of their simplicity
- No IRs: all phases of compiler interleaved.
- Compilation driven by parser.
- Scanner acts as subroutine of parser, returning a token on each call.
- As each phrase recognized by parser, it calls semantic routines to process declarations, check for semantic errors and generate code.
- Code not as efficient as multi-pass

Multi-Pass Compilers

- Number of passes depends on number of IRs and on any optimizations
- Multi-pass allows complete separation of phases

– more modular

– easier to develop

– more portable

- Main forms of IR:

– Abstract Syntax Tree (AST)

– Intermediate Code (IC)

- Postfix
- Tuples
- Virtual Machine Code


FORMAL GRAMMAR

Intoduction to Formal Specification

- Two things are associated with a language: *syntax* and *semantics*.
  E.g.
  - The frog writes neatly. (Valid, Nonsensical)
  - Swims quickly mathematics (Not valid, Nonsensical)
- *Generative specification*
  - *Generates all the valid strings of a language.*
  - *Generatively Decidable language*: if all legal strings can be generated in some finite time.
- Analytic Specification
  - Checks string is legal or not.
  - *Analytically Decidable language:* if the analyzer always stops after finite number of steps.

Definition of Languages

- Natural languages cannot be used to define other Languages
- Some Formal Representation is required.
- Language: **Meta language + Object Language**
- A Formal system is a Meta Language.
- Terminal Symbols are the Object language symbols.

Formal Specification:

- **Alphabet**: A or Σ or S An Alphabet is a finite set of symbols (“terminal symbols”).
- **Formula/String/Sentence**: A Sentence is a concatenation of the symbols.
- **Language**: L is a subset of the set of all the concatenations of symbols in an alphabet T. i.e. L C T*.

Development of Formal Specification

- Let us Overview English Language
  - There is an Underlying Structure for connecting the words.
  - We can Specify that:
    - A sentence is made up of Noun phrase(NP) followed by a Verb Phrase(VP)
    - NP is made up of an Article(Art) followed by an Adjective(Adj.) followed by a Noun(N), or
    - NP is made up of an Article(Art) followed by a Noun(N).
    - VP is made up of verb followed by an Adverb(Adv), or
    - VP is made up of verb

Introduction to Formal Grammar

Models of Computation are used to determine:

- What can be performed by a computer?
- How it can be performed?

Grammar is one of the models of computation

Grammars are used to determine:

- Whether a word is in a language-*Parsing*
- How to create word of a language-*Derivation*

Notation for Formal Grammar Derivations

- Let $\omega_0 = a\ \beta_0\ b$ and $\omega_1 = a\ \beta_1 b$ be strings over V.
  - If $\beta_0 \rightarrow \beta_1$ is a Production of G, we say that β_1 is *directly derivable (or immediately derived)* from β_0 and we write

    $\beta_0 \Rightarrow \beta_1$ (β_1 is obtained from β_0 in one step).

    e.g. <S> => <NP> <VP>
  - If $\beta_0, \beta_1, \beta_2, \beta_3, \ldots\ldots \beta_n$ are strings over V such that

    $\beta_0 \Rightarrow \beta_1$, $\beta_1 \Rightarrow \beta_2$, $\beta_{n-1} \Rightarrow \beta_n$, then we say that β_n is *derivable* from β_0 and we write $\beta_0 \overset{*}{\Rightarrow} \beta_n$ (β_n is obtained from β_0 in zero or many steps).

    The sequence of steps used to obtain β_n from β_1 is called a *derivation.*

Notation for Formal Grammar Derivations

- If we replace the *left most non-terminal* by one of its production bodies. Such a derivation is called a <u>left most derivation</u>, and we indicate that a derivation is left most by using the relations $\Rightarrow_{lm}$ and $\Rightarrow_{lm}^{*}$, for one or many steps, respectively.
- Similarly, it is possible to require that each step the *right most non-terminal* is replaced by one of its bodies, if so, we call the derivation <u>*right most derivation*</u> and use the symbols $\Rightarrow_{rm}$ and $\Rightarrow_{rm}^{*}$ to indicate one or many right most derivation steps, respectively.

Notation for Formal Grammar Derivations

- Any derivation has an equivalent leftmost and an equivalent right most derivation, i.e., if ß is a terminal string, and A a non-terminal symbol, then $A \overset{*}{\underset{lm}{\Rightarrow}} ß$ if and only if $A \overset{*}{\underset{rm}{\Rightarrow}} ß$ and $A \Rightarrow ß$.

Example 1:

Let G = (V,T,S,P), where V ={A,B,S}, T={a,b}, S is the start symbol, and production rules are:

$$
\begin{aligned}
\{\ S &\rightarrow AB && (1)\\
A &\rightarrow aA && (2)\\
A &\rightarrow a && (3)\\
B &\rightarrow Bb && (4)\\
B &\rightarrow b\ \} && (5)
\end{aligned}
$$

Left most derivation

$$
\begin{aligned}
S &\Rightarrow AB && (1)\\
&\Rightarrow aAB && (2)\\
&\Rightarrow aABb && (4)\\
&\Rightarrow aaBb && (3)\\
&\Rightarrow aabb && (5)
\end{aligned}
$$

Hence $S \overset{*}{\Rightarrow} aabb$

Example 2:

Let G = (V,T,S,P), where V ={a,b,A,B,S}, T={a,b}, S is the start symbol, and production rules are:

$$\{ \quad S \rightarrow ABa \quad (1)$$
$$A \rightarrow BB \quad (2)$$
$$B \rightarrow ab \quad (3)$$
$$AB \rightarrow b \ \} \quad (4)$$

Right most derivation

$$S \rightarrow ABa \quad (1)$$
$$\Rightarrow Aaba \quad (3)$$
$$\Rightarrow BBaba \quad (2)$$
$$\Rightarrow Bababa \quad (3)$$
$$\Rightarrow abababa \quad (3)$$

Hence $S \overset{*}{\Rightarrow}$ abababa

Language of the Grammar

Let G = (V,T,S,P) be a phrase-structure grammar the language generated by G(or the language of G) denoted by L(G), is the set of all strings of terminals that are derivable from the starting state S. In other words,

$$L(G) = \{ \omega \in T^* \mid S \overset{*}{\Rightarrow} \omega \}$$

Example 3:

Let G = (V,T,S,P), where V ={a,b,A,S}, T={a,b}, S is the start symbol, and production rules are:

{ S → aA (1)

S → b

(2)

A → aa } (3)

Right most derivation

S → aA (1)

=> aaa (3)

No additional words can be derived.

Hence, L(G) = {b,aaa}.