Formal Languages

Dr D D Karunaratna

Lesson objectives

After successful completion of this lesson, you should be able to

1. describe the need for meta-languages.
2. describe the terms alphabet, syntax and semantics with examples.
3. describe the difference between abstract syntax and concrete syntax.
4. describe what a "string" is and operations that can be performed on strings.
5. define the Kleene closure of an alphabet and its properties.
6. define a language.
7. List the main components of BNF and to define simple languages by using BNF.
8. Identify whether a given string is in the language or not, given the definition of the language in BNF.
9. give the limitations of BNF.
10. define components of a language by using Syntax charts.
11. describe the main parts of a grammar.
12. describe the language defined by a given grammar and to define simple languages by using grammars.

Lesson objectives …

13. Specify the conditions to be satisfied for the grammars to be equivalent.
14. classify grammars by using the **Chomsky's** scheme of classification.
14. construct a derivation sequence/parse tree for a given string.
15. Identify ambiguous grammars.

Formal Languages

- A useful programming language must be suited both for describing and for implementing the solution to a problem.
- There are many ways for specifying the syntax of languages.
- A formal language has to be described by using another language – **meta-language.**
- A precise specification of a language requires an unambiguous meta-language.

- Definition of a language involves three fundamental aspects
  - Alphabet (Σ)
  - Syntax
  - Semantics

Alphabet (Σ)

- Alphabet (Σ) : A finite nonempty set of symbols.
    - The members of the alphabet can be considered as abstract entities with no meaning by themselves along.

Example :

C language alphabet includes symbols such as *a,+,{,if* ....

Syntax and Semantics

- Consider the following sentences
  1. Snake is a mammal.
  2. Snake not mammal is.
  3. Snake is a reptile.

Which of the above statements are correct?

Syntax

- Syntax : linguistic form of sentences in the language
  - Only concerned with the form and structure of symbols of the language rather than the meaning.
  - The syntax of a programming language is commonly divided into two parts namely lexical syntax and phrase-structure syntax.
    - Lexical syntax - describes the smallest units with significance - tokens
    - Phrase-structure syntax - describe how tokens can be combined into programs.

Syntax …

Example : Lexical syntax

<token> ::= <identifier> | <numeral> | <reserved word>

Example : Phrase-structure syntax

<program> ::= **program** <identifier> <block>

<block> ::= <declaration seq> **begin** <command seq> **end**

Semantics

- Semantics : Linguistic meaning of **syntactically correct** sentences.
  - In programming languages semantics are ascribed in terms of the structure of the phrases.
  - For programming languages, semantics describes the behavior of a computer when executing a program in the language.
  - A syntactically correct program need not make any sense semantically.

  Eg : Saman is a married bachelor.

Abstract Vs. Concrete Syntax

- Abstract syntax : Provides the definition of constructs.
- Concrete syntax : Provides the definition of the form of the constructs.

Example: Concrete syntax

C	**Pascal**
while (i <n){	*While i < n do begin*
i = i + 1	*i := i + 1*
}	*end*

String (Word)

- **String(Word)** : finite sequence, $w = a_1a_2a_3 \ldots a_n$, of symbols from the alphabet.
- *Example*
- *let* $\Sigma = \{a, b\}$
- then aa,abab are strings on Σ
  - Two strings are considered the same if all their letters are the same and in the same order.

Notation

- Lower case letters a,b,c,… are used for elements of Σ.
- Lower case letters u,v,w,.. Are used for string names.
- *Example*
- *w = abaa string named w has the specific value abaa*

String Operations

- Concatenation of two strings w and v is the string obtained by appending the symbols of v to the right end of w
- Reverse of a string is obtained by writing the symbols in reverse order.
- Length of a string w, denoted by |w| is the number of symbols in the string.
- Empty string, denoted by $\in(\lambda)$, is the string of length zero
  - $| \in | = 0$

  $\in w = w \in = w$ holds for any string w

String Operations …

– Let $w = uv$
– u is said to be a prefix of w.
– v is said to be a suffix of w.

– Example
– $w = abbab$, then $\{\in, a, ab, abb, abba, abbab\}$ is the set of all prefixes of w.

– If w is a string, then w^n is the string obtained by concatenating w, n times.
– $w^0 = \in$ for any string w

String Operations …

- Let Σ be an alphabet
- Σ^k – is the set of strings of length k with symbols from Σ.

Example :

- let $\Sigma = \{0,1\}$
- $\Sigma^1 = \{0,1\}$
- $\Sigma^2 = \{00,01,10,11\}$
- $\Sigma^0 = \{\in\}$

Kleene closure

- If Σ is an alphabet, then Σ^*, call the **Kleene closure** of the alphabet, denotes the set of strings obtained by concatenating zero or more symbols from Σ .
  - Kleene closure is a unary operation on a set Σ defined as $\Sigma^* = \{\in\} \cup \Sigma^1 \cup \Sigma^2 \cup ....$
  - $\Sigma^+ = \Sigma^* - \{\in\}$

What is a language?

- **Informally a language L over an alphabet Σ is a subset of Σ^*.**
  – A language can be empty, finite, infinite.

Example

– let $\Sigma = \{a,b\}$

– then $\{a,aa,aab\}$ is a **finite language** on Σ

– $L = \{a^n b^n \mid n \geq 0\}$ is an. **infinite language** on Σ

- The Kleene closure of two different sets may define the same language.

Example

S = {a, b, ab} T = {a, b, bb}

Then both S* and T* are languages of all strings of a's and b's since any string of a's and b's can be factored into components of either a or b, both of which are in S and T.

Note : Other than letters, strings can also be elements of an alphabet.

- Theorem : $S^* = S^{**}$, for any set S

Proof :

$x \in S^{**} \Rightarrow x = w_1 w_2 .. w_n$ where each $w_i \in S^*$

$\Rightarrow w_i = l_1 l_2 .. l_m$ where each $l_i \in S$

$\Rightarrow$ x is a concatenation of elements of S

$\Rightarrow x \in S^*$

$S^{**} \subseteq S^*$

Similarly prove $S^* \subseteq S^{**}$

Thus $S^* = S^{**}$

- Since languages are sets, the union, intersection, and difference of two languages are automatically defined.
- The complement of a language is defined with respect to Σ^*.

  $L' = \Sigma^* - L$
- Concatenation of two languages L_1 and L_2 contains every string in L_1 concatenated with every string in L_2.

  $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$

- L^n is defined as the concatenation of L with itself n times

$$L^0 = \{\in\}$$

$$L^1 = L$$

- The star-closure of a language is defined as

$$L^* = L^0 \cup L^1 \cup L^2 \ldots..$$

- The positive closure of a language is defined as

$$L^+ = L^1 \cup L^2 \ldots..$$

Language Definition Mechanisms

- Giving a set of rules, which defines all the acceptable words of the language.
  - The set of language-defining rules can be of two kinds
    - Rules that can be used to identify whether a given string of alphabet letters is in the language or not
    - Rules that can be used to generate all the words in the language.
- A language L over an alphabet Σ is a subset of Σ^*. Thus set notations can be used to define languages. However set notation is inadequate to define complex languages.

Matalanguags
BNF (Backus-Naur-Form)

- First used to describe Algol60.
- BNF is a language for defining the semantics of languages **-metalanguage**
- BNF greatly simplifies semantic specifications.

BNF (Backus-Naur-Form)

Components of BNF

- A finite set of terminal symbols(Σ) – alphabet of the language
  - The sentences in the language are composed by assembling these symbols.
- A set of non-terminal symbols (N)- syntactic categories.
  - Represents different types of sentences in the language and their parts.
- Set of rewriting rules (Productions) (ρ)
  - Describe the structure of terminals in terms of terminals and non-terminals.
- A start symbol (S).
  - specifies the principal category being defined—for example, sentence or program.

In Classic BNF

- A non-terminal is usually given a descriptive name, and is written in angle brackets < >.

  Examples:
  <Identifier>,<Integer>,<Expression>, …

- Productions have the form

  Leftside ::= definition or

  Leftside $\rightarrow$ definition

  Where leftside $\in$ N and

  definition $\in (N \cup \Sigma)^*$

Example

How to define signed integers by means of BNF?

<signed integer> ::= <integer>|+<integer>|
-<integer>

<integer> ::= <digit>|<integer><digit>

<digit> ::= 0 | 1| 2|……|9

A number of extensions are employed with BNF grammars to increase readability and for the elimination of unnecessary recursion.

- Extended BNF notation
  - [x] : Optional element x (x or nothing)
  - {..} : An arbitrary sequence of an element

Example

How to define signed integers by using extended BNF?

<signed integer> ::= [+| -]<digit>{<digit>}

- Extended BNF notation .....
  - White space is only meaningful to separate tokens.
  - Rules are normally contained on a single line;
    - rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

Describing Lists in BNF

A rule in BNF is said to be **recursive** if its LHS appears in its RHS.

Example

$<identifier\_list> \rightarrow \text{identifier}$

$<identifier\_list> \rightarrow \text{identifier}, <identifier\_list>$

– When a BNF rule has its LHS also appearing at the beginning of its RHS, the rule is said to be **left recursive**.

– When a BNF rule has its LHS also appearing at the right end of the RHS, the rule is said to be **right recursive**.

- In a grammar for a complete language, the start symbol represents a complete program and is usually named < program >

Example : (Grammar 1) A grammar for a small language

<program > → begin <stmt_list > end

<stmt_list > → <stmt>

| <stmt > ; <stmt_list>

<stmt> → <var> := <expression>

<var> → A | B | C

<expression> → <var> + <var >

| <var> - <var>

| <var>

Derivations in this language

<program> ⇒ begin <stmt_list > end

⇒ begin <stmt>;<stmt_list> end

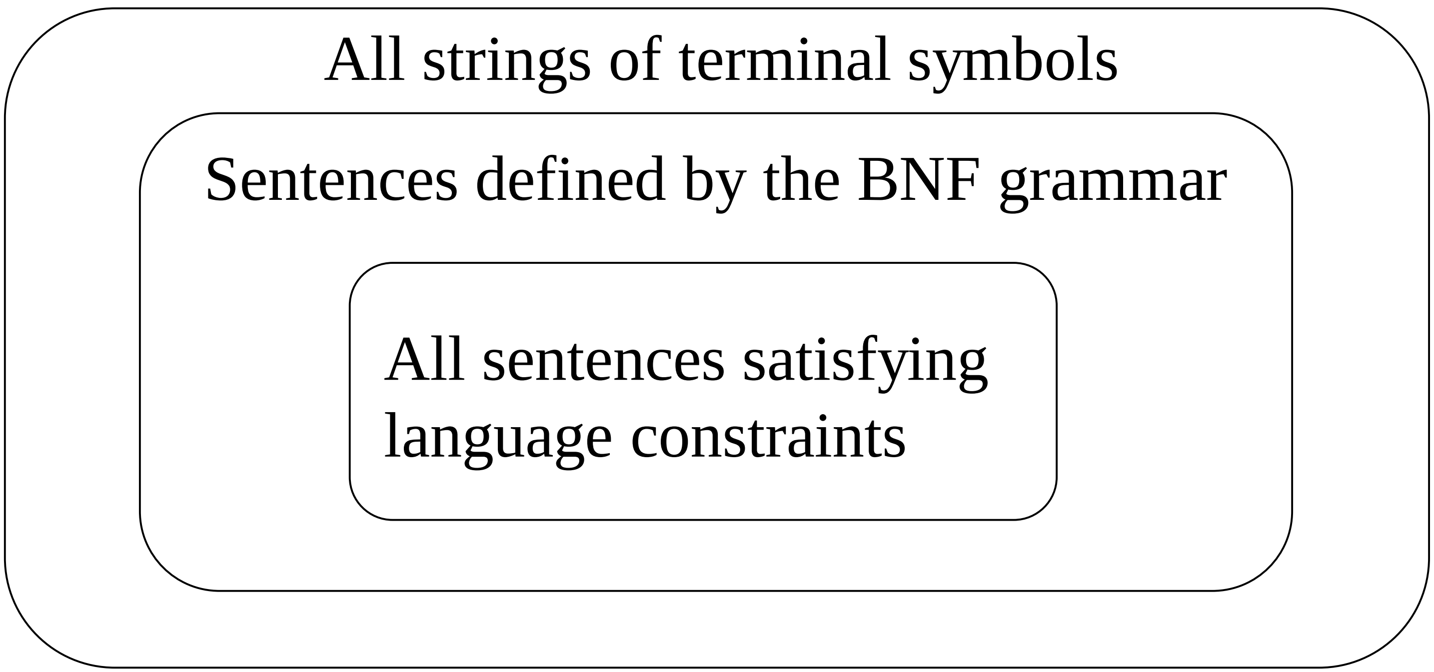
⇒ begin <var> := <expression>;stmt_list> end

……………………………….

⇒ begin A := B +C; B := C end

- ⇒ is read **derives**
- Each of the strings in the derivation, including the <program > …, is called a **sentential form.**

Valid Programs



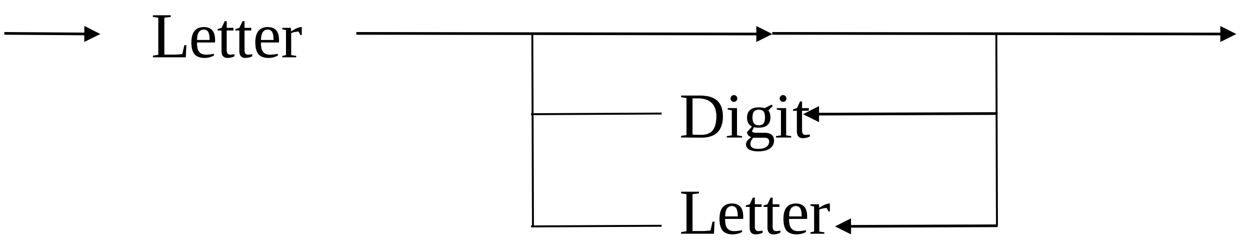
- Derivation :
  - Leftmost derivations
  - Rightmost derivation
  - Neither leftmost nor rightmost
- Derivation order has no effect on the language generated by a grammar.

- The **language defined by BNF grammar** is the set of strings that can be parsed (or derived) using the rules of the grammar.
- BNF grammars have limited power in defining languages.
  - Conceptual dependencies cannot be defined by BNF Grammars.

  Example :
  - The same identifier may not be declared twice in the same block.
  - An identifier cannot be used before declaring it.

Syntax Diagrams(Syntax charts)

Graphical representation for extended BNF rules



Example : Definition of an identifier

Possible paths represents the possible sequence of symbols

Lexical structures and Phrase structures

- The set of productions used to describe a real programming language grammar is usually divided into two distinct groups
  - Lexical structure : the way in which individual characters are combined to form words or tokens.
  - Phrase structure : the way in which the words or tokens of the language are combined to form components of programs.

- Formally, a grammar is a four-tuple(N,$\sum$,P,S)
  - N : the set of non-terminal symbols, denoted by capital letters – Denotes the syntactic classes of the grammar.
  - $\sum$: the set of terminal symbols (or, simply, terminals), denoted by small letters
  - P : set of derivative productions, also called rules, or syntactic equations for generating permissible strings of terminals and non-terminals (sentential form), denoted as a whole by Greek letters.
  - S : A designated initial non-terminal from which all strings in the language are derived

Note :

– $\sum \cap N = \emptyset$ and $S \in N$

- The sentences of a language are generated by starting with the symbol S and applying productions from P to replace non-terminal symbols until a sentential form consisting only of terminals results.
- The set of all such sentences that can be derived in this way is the language defined by the grammar.
- An infinite number of grammars can be developed to generate any particular language.
- A sentence generated by starting with S and applying the productions is called a **derivation**.

- The productions are given in the form

$$\alpha \to \beta \text{ where } \alpha, \beta \in (\Sigma \cup N)^*$$

Indicates the sentential form α may be replaced by the sentential from β

| : read as or, is used to group alternative right parts for the same production.

Derivations

- $w1 \Rightarrow^* w2$
  - w1 can be converted to w2 by applying zero or more rules.
  - w1 derives w2
  - A partially derived string is called a *sentential form* and contains both terminals and non-terminals

$\omega 1 \Rightarrow^+ w2$

  - derive in one or-more steps
- Language of a Grammar G

$L(G) = \{w \mid w \in \Sigma^*, S \Rightarrow^* w\}$

Example :

$G = \{N, \sum, P, S\}$ where

$N = \{S\}$

$\sum = \{0,1\}$

Productions in P are

$S \rightarrow 0S1 | \in$

where $\in$ is the empty string.

The language generated by this grammar consists of all strings containing n $(0\geq)$ 0's followed by n 1's.

$\in$ Productions

Consider the productions of the form

$$L \rightarrow \in$$

results the erasure of the non-terminal L from the sentential form

Regular Grammars

- Regular grammars have rules of the form

<non terminal> ::= <terminal><non terminal> |
<terminal> | $\in$

Example :

A grammar to generate binary strings ending in 0

$A \rightarrow 0A|1A|0$

Regular expressions can also be used to define languages.

Classes of Grammars

Chomsky's scheme of classification

Based on the format of the productions

assume productions are of the form $\alpha_i \rightarrow \beta_i$

– Type 0 : Phrase structure grammars

no restrictions on form of productions $\alpha_i \rightarrow \beta_i$

for all i

- All formal grammars.
- Generates all languages recognizable by a Turing machine

Chomsky's scheme of classification

– Type 1 : Context-sensitive grammars

- $|\alpha_i| \leq |\beta_i|$ for all i , where | | denotes the length

  Note : null string would not be allowed as a right hand side of any production.

Example :

<sentence> ::= abc | a<thing>bc

<thing>b ::= b<thing>

<thing>c ::= <other>bcc

a<other> ::= aa | aa<thing>

b<other> ::= <other>b

– Type 2 : Context free grammars (BNF Grammars)

$\forall \alpha_I$ restricted to a single non-terminal symbol, for all i

– Can be recognized by *pushdown automata*

– Context free grammar is a common notation for specifying the syntax of programming languages.

Example :

In C if-else statement

Stmt $\rightarrow$ **if** (expr) stmt **else** stmt

– Type 3 : regular grammars

- all production of the form A $\rightarrow$ xB or A $\rightarrow$x where A and B are non-terminals and x is in $\sum$* - **right liner grammar.**
- all production of the form A $\rightarrow$ Bx or A $\rightarrow$x where A and B are non-terminals and x is in $\sum$* - **left liner grammar.**
- Can be recognized by finite automata .

- The syntax of a regular language can be expressed by a single EBNF expression.
  - only terminal symbols occur in the expression

Note : type t grammars are also type t-1 for all t > 0

- A language L(G) is said to be of type k if it can be generated by type k grammar.

Example :

- $G_1 = (\ \{0,1\},\{S\},\{S \rightarrow 0S1|\ \in\},S)$
- $G_2 = (\ \{0,1\},\{S,Z,U\},\ P,\ S)$

  $P = \{S \rightarrow ZU,\ Z \rightarrow 0Z|\ \in\ ,\ U \rightarrow 1U|\in\}$
- $G_3 = (\{0,1\},\{S,R\},P,S)$

  $P = \{S \rightarrow\ 0S|0|1|1R|\ \in\ ,\ R \rightarrow 1|1R\}$
  - G_1,G_2,G_3 are context free grammars
  - G_3 is regular
- The more restricted the grammar, the easier it is to construct a corresponding recognizer for the language generated by the grammar

- The BNF and context-free grammar forms are equivalent in power, the differences are only in notation.

- Definition

  Let G = (N, $\sum$,P,S), then the set

  $$L(G) = \{w \mid w \in \sum^*, S \overset{*}{\Rightarrow} w\}$$

  is the language generated by G

- Two grammars are **equivalent** if they generate the same language.
  - Important in designing parsers.
  - For some grammars it is hard/impossible to build practical parser – may be transformed into equivalent grammars that can be parsed.

Example

Let G1 = ({S},{a,b}, S, P1), with P1 given by

$$S \to aSb \mid \in$$

Let G2 = ({A,S},{a,b}, S, P2), with P2 given by

$$S \to aAb \mid \in$$

$$A \to aAb \mid \in$$

G1 is equivalent to G2

- Given a grammar of a language how can we prove that a given string is an element of the language?

  Example :

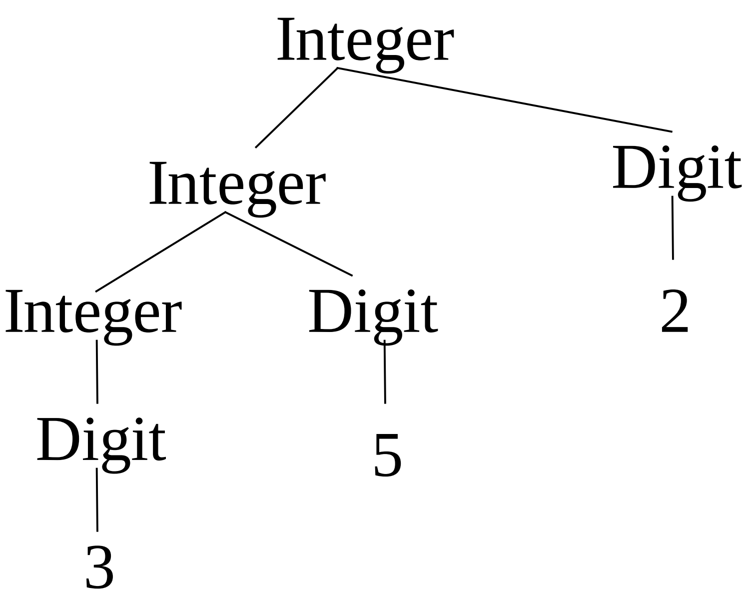
  Integer $\rightarrow$ Digit | Integer Digit

  Digit $\rightarrow$ 0|1|2|3|4|5|6|7|8|9

  Is the string 352 in the language?

- Two main methods
  - Build a parse tree for the string
  - Develop a derivation for the string

Parse Tree



Two different ways of building the parse tree

- Top-down
- Bottom-up

- Derivation

  Integer $\Rightarrow$ Integer Digit $\Rightarrow$ Integer Digit Digit

  $\Rightarrow$ Digit Digit Digit $\Rightarrow$ 3 Digit Digit $\Rightarrow$ 3 5 Digit

  $\Rightarrow$3 5 2

- Each string on the right hand side of a derivation is called a sentential form.
  - Generally contains terminal and non-terminal symbols.
- There are many possible derivation paths from the start symbol to the final sentence depending on the order in which productions are applied.

- **Parse Trees**

  A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

  Formally, given a context-free grammar, a parse tree is a tree with the following properties

  - The root is labeled by the start symbol
  - Each leaf is labeled by a token (terminal) or by $\in$
  - Each interior node is labeled by a non-terminal
  - If A is the nonterminal labeling some interior node and XYZ are the labels of the children of that node from left to right, then $A \rightarrow XYZ$ is a production

Canonical Derivations

Each derivation step requires two kinds of choices to be made.

- Selecting a non-terminal from the sentential form.
- Selecting a production for the non-terminal selected.

A canonical derivation is obtained by imposing some ordering rule for the selection of the next non-terminal to replace in a sentential form.

Two types of canonical derivations

- Left-most derivation
- Right-most derivation

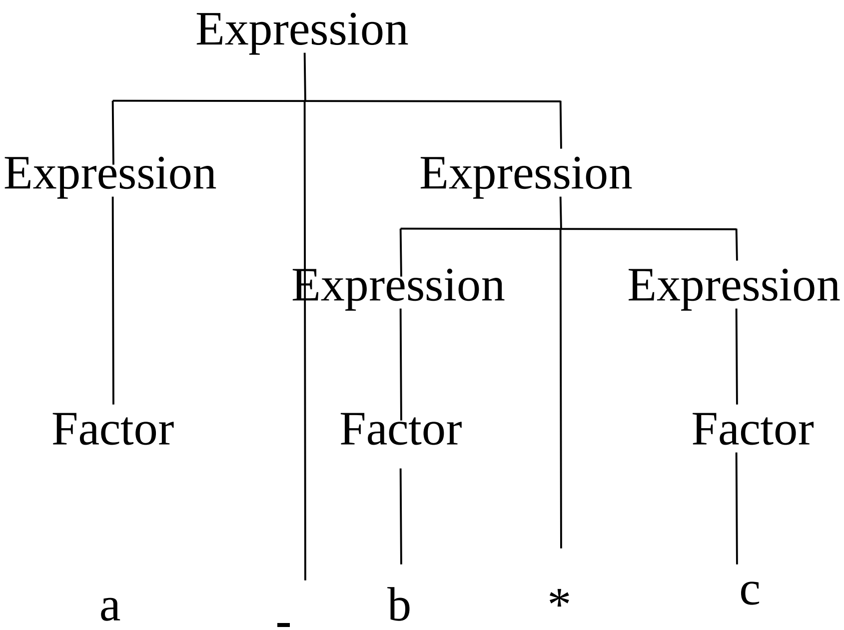
- **Ambiguous Grammars**

  A grammar is ambiguous **if at least one sentence in its language has more than one valid parse tree**. Since the parse tree of a sentence is often used to infer its semantics, an ambiguous sentence can have multiple meanings.

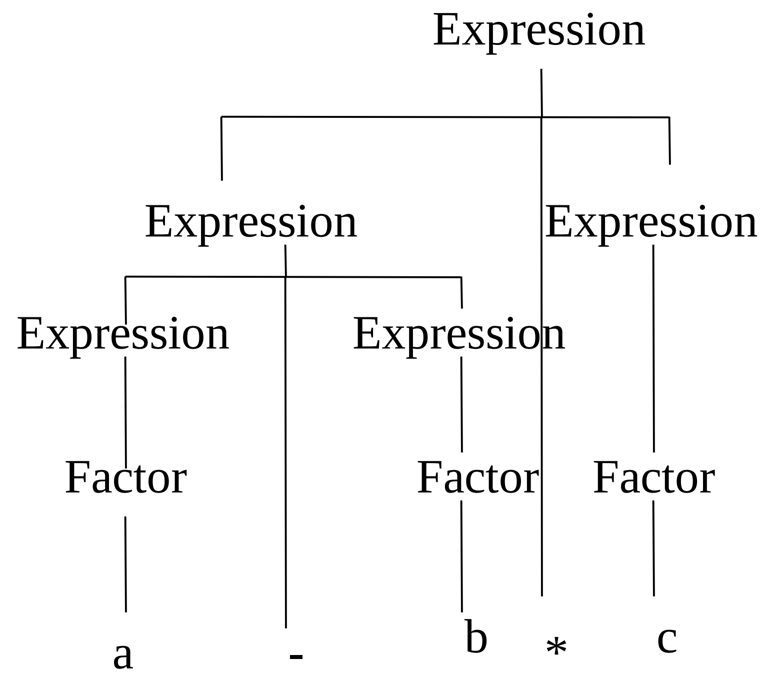
Example

$$
\begin{aligned}
Expression &\rightarrow Expression - Expression \\
&\mid Expression * Expression \\
&\mid Factor \\
Factor &\rightarrow a \mid b \mid c
\end{aligned}
$$

- Consider the sentence a – b * c



a- (b*c)



(a-b) * c

No algorithm exists that can take an arbitrary grammar and determine with certainty and in finite time whether it is ambiguous or not.

Ambiguous Grammars...

- In the previous example the grammar does not reflects the true semantics of the operators – results ambiguity.
- It is possible to embed some of the semantics of a language in its syntax.
- Example

  Expression $\rightarrow$ Expression – Term | Term

  Term $\rightarrow$ Factor | Term * Factor

  Factor $\rightarrow$ a | b | c

Ambiguous Grammars...

- Ambiguities can arise in recursive productions where a particular non-terminal can be replaced at two different locations in the definition.
- There exists no general method for determining whether an arbitrary BNF specification is ambiguous or not.