Unit 3.
Generative Grammars

Basic concepts of languages theory

- Parsing or syntactic analysis is the process of analyzing a string of symbols, either in natural languages or in computer languages, conforming to the rules of a formal grammar.
- The formal language theory considers a language as a mathematical object.
- A language is just a set of strings (sentences). To formally define a language we need to formally define what are the strings admitted by the language.

Alphabet

Symbol

A physical entity that we shall not formally define; we shall rely on intuition.

Alphabet

A finite, non-empty set of symbols

- We often use the symbol ∑ (sigma) to denote an alphabet
- Examples of alphabet
  - Binary: ∑ = {0,1}
  - All lower case letters: ∑ = {a,b,c,..z}
  - Alphanumeric: ∑ = {a-z, A-Z, 0-9}
  - DNA molecule letters: ∑ = {a,c,g,t}(guanine, adenine, thymine, and cytosine)
  - C character set
  - KPL token set.

C character set

<table>
<tr><th>Types</th><th>Character Set</th></tr>
<tr><td>Lowercase Letters</td><td>a –z</td></tr>
<tr><td>Uppercase Letters</td><td>A - Z</td></tr>
<tr><td>Digits</td><td>0-9</td></tr>
<tr><td>Special Characters</td><td>~! # $% ^ & *( )_ +| \’ - = { } [] : ”
; < > ? , . /</td></tr>
<tr><td>White Spaces</td><td>Tab Or New line Or Space</td></tr>
</table>

Token set of KPL

- Identifiers, numbers, character constants
- Keywords

  PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL, IF, ELSE, WHILE, DO, FOR, TO
- Operators

  := (assign), + (addition), - (subtraction), * (multiplication), / (division), = (comparison of equality), != (comparison of difference), > (comparison of greaterness), < (comparison of lessness), >= (comparison of greaterness or equality), <= (comparison of lessness or equality)

Separators:

:,;,(,),,,(.,.),.

String (sentence)

- ***A string is finite sequence of symbols chosen from some alphabet***
- Empty string is ε
- Examples of string:
  - 1000010101111
  - A C program is a string of tokens
  - A human DNA pattern

```
GGTGTGGGGACAGGGGTGTGGGGACAGGGGTCTGGGGACAGGGGTGTGGG
GACAGGGGTCCTGGGGACAGGGGTGTGGGGATAGGGGTGTGGGGACAGGG
GTGTGGGGACAGGGGTGTGGGGACAGGGGTCTGGGGACAGCAGCGCAAAG
AGCCCCGCCCTGCAGCCTCCAGCTCTCCTGGTCTAATGTGGAAAGTGGCC
CAGGTGAGGGCTTTGCTCTCCTGGAGACATTTGCCCCCAGCTGTGAGCAG
GGACAGGTCTGGCCACCGGGCCCCTGGTTAAGACTCTAATGACCCGCTGG
TCCTGAGGAAGAGGTGCTGACGACCAAGGAGATCTTCCCACAGACCCAGC
ACCAGGGAAATGGTCCGGAAATTGCAGCCTCAGCCCCCAGCCATCTGCCG
ACCCCCCCACCCCAGGCCCTAATGGGCCAGGCGGCAGGGGTTGAGAGGTA
GGGGAGATGGGCTCTGAGACTATAAAGCCAGCGGGGGCCCAGCAGCCCTC
```

Languages

A language over alphabet Σ is a set of strings over Σ

Examples of languages:

- The set of all words over {a, b},
- The set { $a^n \mid n$ is a prime number },
- Programming language C: the set of syntactically correct programs in C

Chomsky's Hierarchy

- Type-0 languages (recursive enumerable)

  instances of a problem.
- Type-1 languages (context-sensitive)

  natural languages, DNA languages
- Type-2 languages (context-free)

  programming language, natural languages
- Type-3 languages (regular)

  tokens of programming languages

A grammar to generate real numbers

<real number> ::= <sign><natural number> |
<sign><natural number>'.'<digit sequence> |
<sign>'.'<digit><digit sequence> |
<sign><real number>'e'<natural number>

<sign> ::= ‘’ | ‘+’ | ‘-‘

<natural number> ::= ‘0’ | <nonzero digit><digit sequence>

<nonzero digit> ::= ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<digit sequence> ::= ‘’ | <digit><digit sequence>

<digit> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

How a string of a context free language can be generate ?

A context free grammar can be used to generate strings in the corresponding language as follows:

let X = the start symbol s

while there is some nonterminal Y in X do

apply any one production rule using Y, e.g. Y -> w

Context Free Grammars (CFG)

A context free grammar G has:

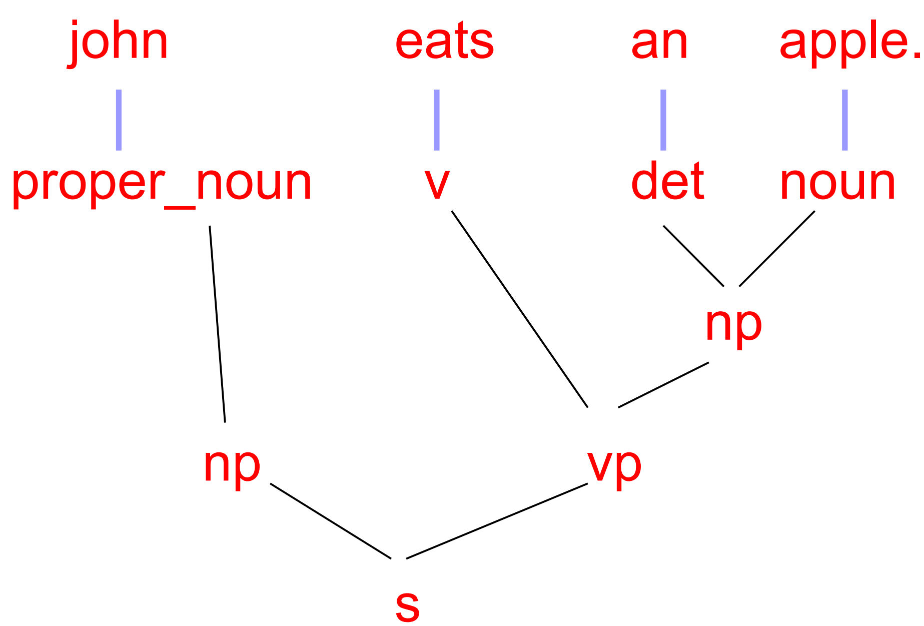
- A set of terminal symbols, Σ
- A set of nonterminal symbols, Δ
- A start symbol, S, which is a member of Δ
- A set P of production rules of the form A -> w, where A is a nonterminal and w is a string of terminal and nonterminal symbols or ε.

Parse Tree

s --> np, vp.
np --> det, noun.
np --> proper_noun.
vp --> v, ng.
vp --> v.

det -->[a]. det --> [an].
det --> [the].
noun --> [apple].
noun --> [orange].
proper_noun --> [john].
proper_noun --> [mary].
v --> [eats].
v --> [loves].

Eg.



Context Free Grammar Examples

- Grammar of nested parentheses
  $G = (\Sigma, \Delta, P, S)$ where

  $\Delta = \{S\}$

  $\Sigma = \{\ (,\ )\ \}$

  $P = \{\ S \rightarrow (S),\ S \rightarrow SS,\ S \rightarrow \varepsilon\ \}$

Context Free Grammar Examples

- The grammar of decimal numbers

S -> +A | -A |A

A -> B.B | B

B -> BC | C

C -> 0 | 1 | 2 |. . . .|9

Derivations

- When X consists only of terminal symbols, it is a string of the language denoted by the grammar.
- Each iteration of the loop is a derivation step.
- If an iteration has several nonterminals to choose from at some point, the rules of derviation would allow any of these to be applied.
- Example : S⇨-A⇨-B.B⇨-B.C⇨-C.C⇨-1.C⇨-1.5

Leftmost and Rightmost Derivations

- In practice, parsing algorithms tend to always choose the leftmost nonterminal, or the rightmost nonterminal, resulting in strings that are leftmost derivations or rightmost derivations
- Example:

Leftmost derivation:

S⇨-A⇨-B.B⇨-C.B⇨-1.B⇨-1.C⇨-1.5

Rightmost derivation:

S⇨-A⇨-B.B⇨-B.C⇨-B.5⇨-C.5⇨-1.5

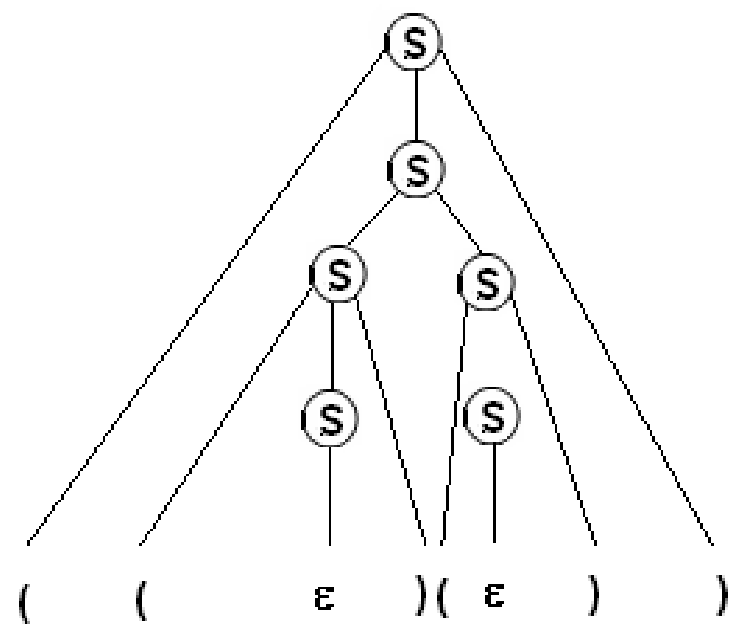
Derivation Tree (parse tree)

Derivation tree is constructed with

1) Each tree vertex is a variable (nonterminal) or terminal or epsilon
2) The root vertex is S
3) Interior vertices are from Δ, leaf vertices are from ∑ or epsilon
4) An interior vertex A has children, in order, left to right,

   $X_1, X_2, \ldots, X_k$ when there is a production in P of the

   form $A \text{->} X_1 X_2 \ldots X_k$
5) A leaf can be epsilon only when there is a production $A \to \varepsilon$ and the leaf's parent can have only this child.



Ambiguity

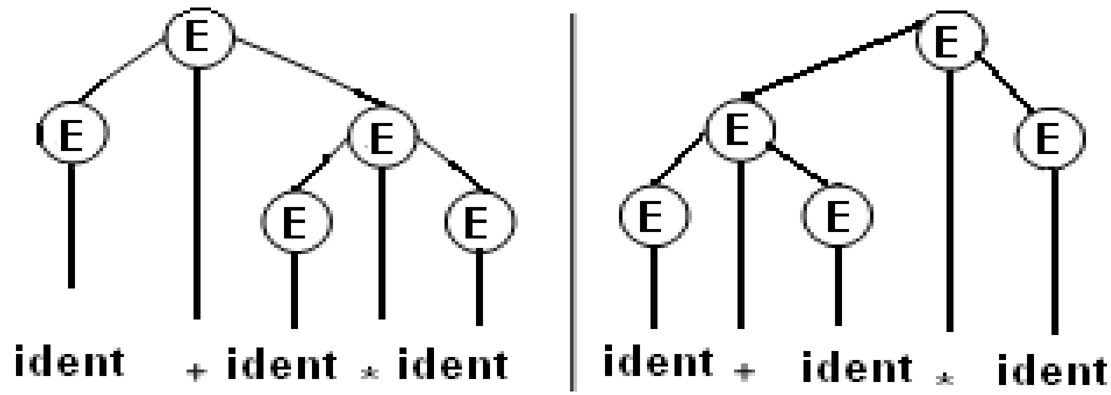
Grammar

E -> E + E

E -> E * E

E -> (E)

E -> ident



allows two different derivations for strings such as ident + ident * ident (e.g. x + y * z)

The grammar is ambiguous

Disambiguation

E -> E + T

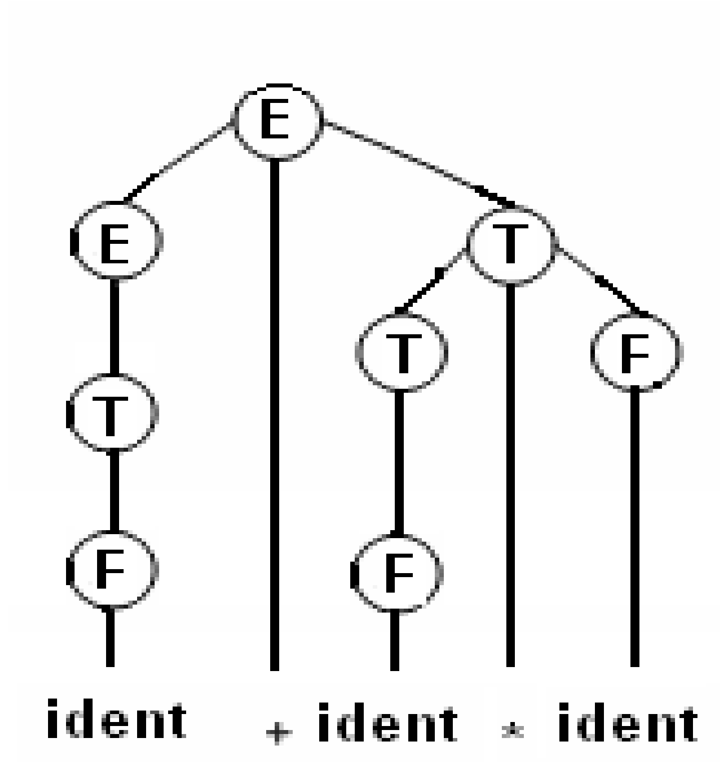
E -> T

T -> T * F

T -> F

F -> (E)

F -> ident



(by adding some nonterminals and production rules to force operator precedence)

Recursion

- **A production is recursive if** $X \Rightarrow^* \omega 1X\ \omega 2$ Can be used to represent repetitions and nested structures

Direct recursion $X \Rightarrow \omega_1 X\ \omega_2$

Left recursion X -> b | Xa. X ⇒X a ⇒X a a ⇒X a a a ⇒b a a a a a ...
Right recursion X -> b | a X. X ⇒a X ⇒a a X ⇒a a a X ⇒... a a a a a b
Central recursion X ->b | (X). X ⇒(X) ⇒((X)) ⇒(((X))) ⇒(((... (b)...)))

Indirect recursion $X \Rightarrow^* \omega_1 X\ \omega_2$
Example

Removing Left Recursion

Let the left-recursive productions in which A occurs as lhs be

$$A \rightarrow A\alpha_1$$

. . . .

$$A \rightarrow A\alpha_r$$

and the remaining productions in which A occurs as lhs be

$$A \rightarrow \beta_1$$

. . . .

$$A \rightarrow \beta_s$$

Removing Left Recursion

Let K_A denote a symbol which does not already occur in the grammar.

Replace the above productions by:

$$A \to \beta_1 K_A \mid . \ . \ . \mid \beta_s K_A$$
$$K_A \to \varepsilon \mid \alpha_1 K_A \mid \ldots \mid \alpha_r K_A$$

Clearly the grammar G' produced is equivalent to G.