Context-free Languages and Grammars

Prof. (Dr.) K.R. Chowdhary
Email: kr.chowdhary@iitj.ac.in

Former Professor & Head, Department of Computer Sc. & Engineering
MBM Engineering College, Jodhpur

Saturday 30th January, 2021

Introduction

- CFLs and CFGs are fundamentals to computer science, because they help in describing the structure of programming languages. All the HLL are in the category of CFL. Though natural languages (NLs) are not CFL, but their analysis is possible only when they are treated as CFLs.
- Context-free languages(CFL) are more powerful than regular languages. The context-free grammars (CFG) are generators of CFL. Regular languages are subset of CFL.
- CFG is finite specification of rules to generate infinite context-free language.
- To generate all the strings of regex $a^*(b^* + c^*)b$, we follow the steps:
  1. write character a zero or more times
  2. arbitrarily choose b or c and write it arbitrary times
  3. write b.

Generating Language strings

- Let $L = \mathscr{L}(a^*(b^* + c^*)b)$ is language corresponding to regex. We can generate all the strings by substitution rules:

1. $S \rightarrow AMb$, 6. $B \rightarrow \varepsilon$
2. $A \rightarrow aA$, 7. $B \rightarrow bB$
3. $A \rightarrow \varepsilon$, 8. $C \rightarrow cC$
4. $M \rightarrow B$, 9. $C \rightarrow \varepsilon$
5. $M \rightarrow C$

Consider generating $w = aaccb$ using production rules.

$$
\begin{aligned}
S &\Rightarrow AMb \text{ ;by rule 1} \\
&\Rightarrow aAMb \text{ ;by rule 2} \\
&\Rightarrow aaAMb \text{ ;by rule 2} \\
&\Rightarrow aaMb \text{ ;by rule 3} \\
&\Rightarrow aaCb \text{ ;by rule 5} \\
&\Rightarrow aacCb \text{ ;by rule 8} \\
&\Rightarrow aaccCb \text{ ;by rule 8} \\
&\Rightarrow aaccb \text{ ;by rule 9}
\end{aligned}
$$

Therefore, $aaccb \in L$. Symbol "$\rightarrow$" stand for "can be substituted by", and "$\Rightarrow$" stand for "derives." Strings like $aacCb$ or AMb, during the derivation are called *sentential form*.

Generating language strings

Let us try to generate the strings of language $L = \{a^n b^n | n \geq 0\}$. The rules this time are: $S \rightarrow aSb$, $S \rightarrow \varepsilon$. Consider deriving $w = aaabbb$.

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaaSbbb \\ &\Rightarrow aaabbb \end{aligned}$$

Therefore, $S \Rightarrow^* aaabbb$. The generator of these languages, is CFG $G = (V, \Sigma, S, P)$, where:

- V is finite set of variables symbols, appearing in the process of derivation
- Σ is set of terminal symbols (appearing in the final generated sentence), $V \cap \Sigma = \phi$,
- S is start symbol,
- P is set of production/ substitution rules of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
- Symbols in upper case in the beginning of English alphabets are variables (non-terminal symbols), i.e. A, B, C, D, E.

Derivations

Definition

Context-free grammar: A CFG is regular grammar if productions are like: $A \to a, A \to aB, A \to \varepsilon$, where, $a \in \Sigma$, and $A, B \in V$. For the regular expression $a^*(b^* + c^*)b$, a CFG was used in the previous slides to generate the regular language.

Definition

Derivation: A derivation $\alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \cdots \Rightarrow_G \alpha_n$, can be written in short as $\alpha_1 \Rightarrow_G^* \alpha_n$. In a derivation: $\beta A \gamma \Rightarrow_G \beta \alpha \gamma$, the symbol A can be substituted by α, if there is production like $A \to \alpha$, irrespective of presence of substrings β and γ around the non-terminal symbol A. Languages having this property are called context-free. The substrings β and γ are called context of variable A. Relation $\Rightarrow$ is *reflexive, anti-symmetric, and transitive* (a partial ordering) relation.

Definition

Language acceptability: $L = L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\}$. Two grammars are equal if they generate the same language.

Derivations

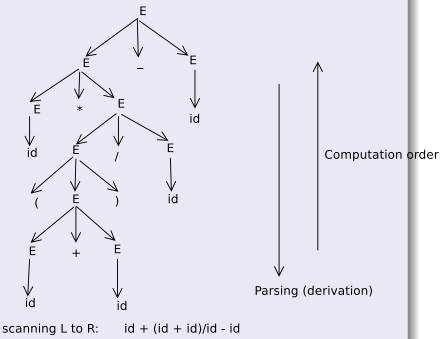
Example

Given grammar $G = \{V, \Sigma, S, P\}$, $\Sigma = \{+, -, *, /, (,), id\}$, $V = \{E\}$, $S = E$, and $P = \{E \rightarrow E + E \mid E - E \mid E/E \mid E * E \mid (E) \mid id\}$, find out the derivation and derivation tree for $id * (id + id)/id - id$.

Solution

The generating process:

$$
\begin{aligned}
E &\Rightarrow E - E \Rightarrow E * E - E \\
&\Rightarrow id * E/E - E \Rightarrow id * (E)/E - E \\
&\Rightarrow id * (E + E)/E - E \Rightarrow id * \\
&\qquad\qquad (id + E)/E - E \\
&\Rightarrow id * (id + id)/E - E \\
&\Rightarrow id * (id + id)/id - E \\
&\Rightarrow id * (id + id)/id - id
\end{aligned}
$$

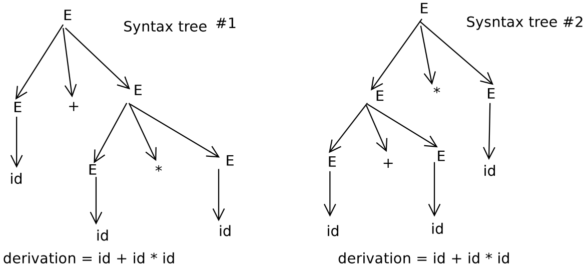


Derivations and ambiguity

- Compilers derive a given expression, if it succeed, the expression is syntactically correct, else not.
- In a derivation, e.g., $S \Rightarrow ABC$, we may start by first replacing left side variables (leftmost derivation) or the right side variable first (rightmost derivation). In both, the end result is the same.
- If a language $L = L(G)$ can be derived using two or more different derivation trees, the grammar G is ambiguous grammar.
- If G has maximum n number of derivation trees, then its degree of ambiguity n.
- It is recursively unsolvable, to find out if an arbitrary grammar is ambiguous. Thus, there does not exist an algorithm to find out if a given grammar is ambiguous.
- A grammar is un-ambiguous if every $w \in L(G)$ has a unique parse-tree. A grammar is called reduced, if every non-terminal appears in some derivation.

Ambiguity

- Show that grammar for $id + id * id$ is ambiguous.



- The two derivation trees have different semantics: first calculates $id + (id * id)$ while the second does $(id + id) * id$, hence the grammar is ambiguous.

Removing ambiguity

The general case of detection of ambiguity in a grammar is unsolvable. However, if it is found that the grammar is ambiguous, it can be made unambiguous by adding few more non-terminals in the grammar.

Example

Given $\Sigma = \{(,),+,*,id\}$, $P = \{E \to E + E|E * E|(E)|id\}$, which is an ambiguous grammar, find out its equivalent unambiguous grammar.

Solution

Let $V = \{E, T, F\}$, *and* $P = \{E \to E + T, E \to T, T \to T * F, T \to F, F \to (E), F \to id\}$. *Note that, you can derive a string in one way only.*

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \Rightarrow id + T \\
&\Rightarrow id + T * F \Rightarrow id + F * F \\
&\Rightarrow id + id * F \Rightarrow id + id * id
\end{aligned}
$$