CS 208 : Automata Theory and Logic 14-03-2024

Lecture - 28

Topic: Context-Free Grammars

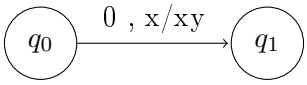
Scribed by: Geet Singhi (22B1035)
Checked and compiled by:

Recap

In the last lecture, we analyzed how to understand languages accepted by a Pushdown Automaton by way of empty stack. We essentially converted a geometric representation of a Pushdown Automaton to an algebraic representation. We shall give a brief description of the method.
We essentially defined a family of languages as follows -

$$L_{q_i x q_j} = \{w \mid \text{starting at } q_i \text{ with the stack having only } x \text{ we end up at } q_j \text{ with an empty stack}\}$$

Now, for every transition, we can write a recursive algebraic equation. For example, let us consider the following example:



Let us assume that the states in the Pushdown Automaton are labeled as $q_0, q_1 \cdots q_n$. Then the recurrence relation is as follows -

$$L_{q_0 x q_2} \supseteq \bigcup_{i=0}^{n} 0 \cdot L_{q_1 x q_i} \cdot L_{q_i y q_2}$$

Here $\cdot$ represents concatenation. A short, informal proof of the above relation is as follows -
Let $w \in \bigcup_{i=0}^{n} 0 \cdot L_{q_1 x q_i} \cdot L_{q_i y q_2} \implies \exists i$ such that $w \in 0 \cdot L_{q_1 x q_i} \cdot L_{q_i y q_2}$
Therefore, we can write w as $0 \cdot w_1 \cdot w_2$ where $w_1 \in L_{q_1 x q_i}$ and $w_2 \in L_{q_i y q_2}$. Therefore, on reading w, when we are at the state q_0 and have x on the stack, using the first 0, we move to the state q_1 and have xy on the stack. Now, on reading w_1, as $w_1 \in L_{q_1 x q_i}$, we move to q_i and have y on the stack. Now, on reading w_2, as $w_2 \in L_{q_i y q_2}$, we move to q_2 and have nothing on the stack. Hence $w \in L_{q_0 x q_2}$ and with that we conclude our demonstration.
Now, if there are n states and k stack symbols in our Pushdown Automaton, there are $n^2 k$ such languages (of the form $L_{q_1 x q_2}$) that can be defined.
Since the language that the Pushdown Automaton represents (by acceptance by empty stack) is described completely by the recurrence relations, let us try and see if we can analyze the recurrence relations by themselves.
Now, we have a bunch of languages (of the form $L_{q_1 x q_2}$) (let us call them $L_1, L_2 \cdots$ for brevity) and some recurrence relations between them.
We define a new notation for the recurrence relations as follows -
Let's suppose that we have the following: $L_1 \supseteq \{1\}, L_1 \supseteq 0 \cdot L_2 \cdot L_3, L_1 \supseteq 0 \cdot L_4 \cdot L_1$
We write this as $L_1 \to 1 \mid 0 \cdot L_2 \cdot L_3 \mid 0 \cdot L_4 \cdot L_1$

Defining Context-Free Grammars

First, we shall define the components of a context-free grammar. A context-free grammar consists of equations that are of the following form - $L \to A \cdot B \mid L \cdot C \cdot L \mid 0$
Every context-free grammar has some terminology which we shall define as follows -

1. *Non-terminals* : Non-terminals are the set of variables that are written to the left of the $\to$. Each of the symbols represents a language (i.e. a set of words).
2. *Terminals* : Terminals are the set of symbols other than the non terminals. They form the alphabet of the language.
3. *Start Symbol* : This is one of the non-terminals that represents the language being defined.
4. *Production Rules* : This is the set of rules that help recursively define the language. Each production rule consists of -
    - A variable that is being (partially) defined by the rules of the production.
    - The production symbol $\to$
    - A string of zero or more terminals and non terminals. This string, also called the *body* of the production represents one of the possible ways to construct the language of the variable to the left of the $\to$

The reason it's called Context-Free Grammar is as follows -

- The Context-Free part means that the words that are generated using the production rules are independent of what they are concatenated with, as opposed to *Context-Dependent* Grammars where the words that are generated are dependent on what they are concatenated with.
- The Grammar signifies that these are rules that we are using to construct the language.

Note: There is a subtle but important point when it comes to what we mean by the language of a variable. For example, let's take the language of the variable L as defined by this production rule -

$$L \to 01 \mid 0 \cdot L \cdot 1$$

Now, we can see that $L = \Sigma^\star$ trivially satisfies this production rule. In fact, for any context free grammar, setting all the non-terminals to $\Sigma^\star$ will always satisfy all production rules. This is trivial and hence it's not of too much interest to us.
So then, what we mean by language of L (Context-Free Language) is the *minimal language* of L. Informally, what we mean by minimal language is the set of words that always belong to the language of L for all possible languages that satisfy the production rules. Defining it a bit more formally, the language of L is the set of words S such that $L = S$ satisfies the production rules and for any language S' such that $L = S'$ satisfies the production rules, then $S \subseteq S'$. Here, in this case for example we can prove by induction that $L = \{0^n 1^n \mid n \in N, n \geq 1\}$ is the minimal language that satisfies this particular production rule.

An Example

Let us take the following example -

$$S \to A \cdot S \mid \epsilon$$
$$A \to A \cdot 1 \mid 0 \cdot A \cdot 1 \mid 01$$

Here the different components are as follows -

1. Non-terminal symbols - S, A
2. Terminal symbols - 0,1
3. Start Symbol - S (needs to be separately specified)
4. Production rules -

$$S \to A \cdot S \tag{1}$$
$$S \to \epsilon \tag{2}$$
$$A \to A \cdot 1 \tag{3}$$
$$A \to 0 \cdot A \cdot 1 \tag{4}$$
$$A \to 01 \tag{5}$$

Formally, a CFG can be represented as follows -

$$G = (V, \Sigma \cup \{\epsilon\}, P, S)$$

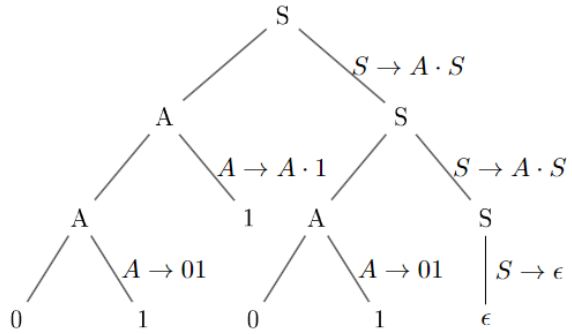
where
V - is the set of non-terminals
$\Sigma \cup \{\epsilon\}$ - is the set of terminals
P - are the production rules
S - is the start symbol ($S \in V$)
The language of the grammar G which is a Context-Free Language is the language corresponding to the start symbol.

Solving the Production Rules

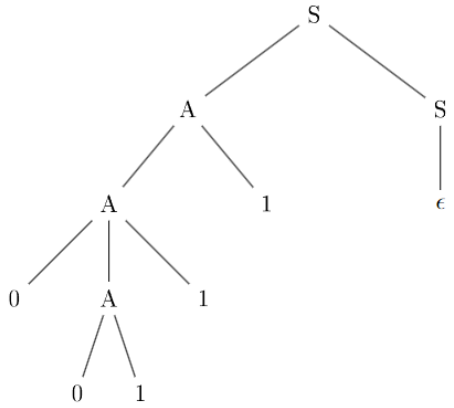
We note that, in the above production rules, A recurses back onto itself and S recurses onto A, so it makes sense to try and find out the language of A first.
We can prove by induction that language of $A = \{0^i 1^k \mid k \geq i, i \geq 1\}$
On analyzing the rule $S \to A \cdot S \mid \epsilon$, we can deduce that the language of $S = L(A)^\star$ (Note that we are using the Kleene star notation here, even though the language of A is not regular (we are simply borrowing the meaning)).

Constructing Strings using the Production Rules

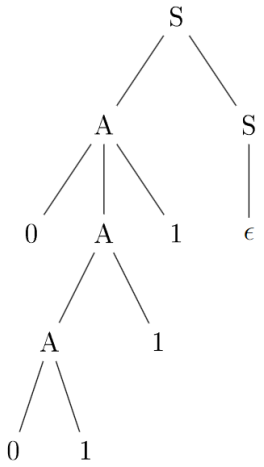
Now, we note that 01101 belongs to the language of G, and thus, we endavour to find a way of constructing it using the production rules.
This method of construction can be represented by a tree. We note that the root of the tree is always the start symbol S (since the language of the grammar is the language of the start symbol) and every time we use a production rule, we split a node into its children.
For example, a possible construction of the word 01101 looks like as follows-



This tree is also called a *derivation tree* or a *parse tree*.The rules used for each splitting are written on the right edge. On reading the leaves from left to right, we can reconstruct the original word.
A natural question to ask now is, for all words in the language of G is there a unique parse tree for each word. We shall show that this is untrue.
Let us consider the string 00111. We see that it belongs to the language of G and we shall show two parse trees that can reconstruct it.



There is another parse tree that gives us the same word 00111. We shall draw it as follows -



Ambiguity of Context-Free Grammars

We saw, in the last section, that the word 00111 which belongs to the language has at least 2 different parse trees. Grammars which have words in their language which do not have a unique parse tree are called *ambiguous grammars*. Hence, our grammar is an ambiguous grammar.
The reason studying ambiguity of grammars is important is because programming languages are based on context-free grammars and the grammars should not be ambiguous, otherwise there would be more than 1 way to interpret some words, defeating the entire point of programming languages. There are Context-Free Languages such that all grammars that we write for them are ambiguous. These languages are called *inherently ambiguous*. Programming languages should not be inherently ambiguous.
Now, we have noticed that the grammar provided to us in the above example is ambiguous, but the language that is represented by it is not inherently ambiguous.
We shall provide another CFG for the same language that is not ambiguous.

$$A \to 0 \cdot A \cdot 1 \mid 01$$
$$B \to 1 \cdot B \mid \epsilon$$
$$C \to A \cdot B$$
$$S \to C \cdot S \mid \epsilon$$

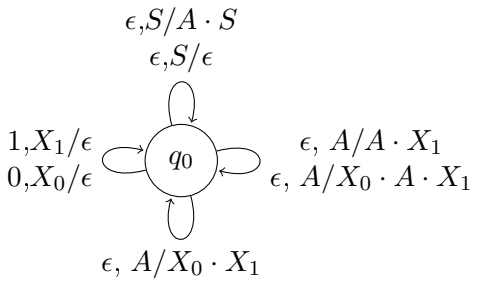
The start symbol here is S. We shall provide some intuition for why this is true. In our original CFG, A was handling both the right concatenation of 1s and adding 0s and 1s on both sides, which led to different parse trees. Here, we have restricted A to handle only 0s and 1s and B handles only appending 1s, which provides a definite path to C and by extension S.

Relation between PDAs and CFGs

In the last lecture and at the beginning of this one, we noted that all languages that are accepted by PDAs by Empty Stack can be constructed using CFGs. And, we also previously showed that all languages that are accepted by PDAs by Final State are also accepted by PDAs by Empty Stack. Hence all languages that are accepted by PDAs can be represented by CFGs. In this section, we aim to show that all languages that can be represented by CFGs are accepted by a suitable PDA by empty stack.
We go back to our example of the CFG being

$$S \to A \cdot S \mid \epsilon$$
$$A \to A \cdot 1 \mid 0 \cdot A \cdot 1 \mid 01$$

Note that the start symbol is S
Here, in constructing our non deterministic pushdown automaton we define our alphabet to be $\Sigma = \{0, 1\}$ and we define one stack symbol for each of the non-terminal states and each of the terminal states.
So $\Gamma = \{S, A, X_0, X_1\}$. Here S, A represent the non-terminal symbols S, A respectively and X_0, X_1 represent the terminal symbols $0, 1$ respectively. The stack starts out with the start symbol S on it. Our pushdown automaton basically follows the exact decision procedure that we might take to see whether a word belongs to the language or not. We have only one state and various transitions on it. We have one ϵ transition for every production rule that we have.
We also have two transitions that basically 'read' the terminal symbols off the stack.
We draw our Non deterministic Pushdown Automaton as follows -



An example for the loops that the automaton takes to parse the word 01101 is -

1. The automaton starts at q_0 with the stack having S
2. The automaton traverses the edge ϵ,$S/A \cdot S$ and the stack now has AS
3. The automaton traverses the edge ϵ,$A/A \cdot X_1$ and the stack now has AX_1S
4. The automaton traverses the edge ϵ,$A/X_0 \cdot X_1$ and the stack now has $X_0X_1X_1S$
5. The automaton traverses the edge 0,X_0/ϵ and the stack now has X_1X_1S
6. The automaton traverses the edge 1,X_1/ϵ and the stack now has X_1S
7. The automaton traverses the edge 1,X_1/ϵ and the stack now has S
8. The automaton traverses the edge ϵ,$S/A \cdot S$ and the stack now has AS
9. The automaton traverses the edge ϵ,$A/X_0 \cdot X_1$ and the stack now has X_0X_1S
10. The automaton traverses the edge 0,X_0/ϵ and the stack now has X_1S
11. The automaton traverses the edge 1,X_1/ϵ and the stack now has S
12. The automaton traverses the edge ϵ,S/ϵ and the stack now is empty and the word is accepted