CSC301 Lecture 9

1910456 Mir Shafayat Ahmed

August 17, 2021

Review so far

When we were talking about regular expressions, we talked about regular language. We minimized DFA. Which meant we can minimize NFAs as they can be converted to DFA. We made NFAs into Regex and vice versa. So we can also make DFAs from Regex. We discussed all of them in a "framework" known as regular language.
We were only allowed to use + (or), Concatenation and * (star).

A Shortcoming of Regex

A simple requirement of programming languages is making sure that the number of left brackets are equal to the number of right brackets.
But with Regex, there is no way to "count" exactly how many we need. When using 0* we have no control over how many 0s are produced.
What we want is something like 0^n1^n where the number of 0s is exactly the same as 1s.
To include strings as simple as 000111 and 0011, we use **Context Free language** which is expressed with **Context Free Grammar**.

Context Free Languages

Trees

Figure 1: Precedence Tree

When we want to calculate 2+3*5, we think from the bottom up approach (recursion trees). Hence, the tree on the right of Figure 1 is the method we follow (how we do math)
So we do 3*5 first and then, 2+15.

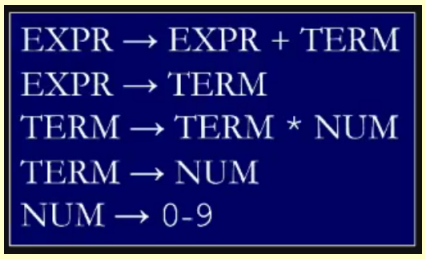

Figure 2: Rule Of Basic Arithmetic Operations

In Figure 2 we can say that an expression (EXPR) can be a summation of an expression and a term. An expression can also be just a term. A term can be the product of a term and a number. A number is anything between [0, 9].
Now we see if we can derive the previous Arithmetic operation from this rule:

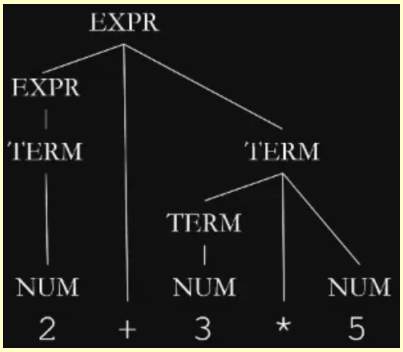

Figure 3: Tree From Expression

In Figure 3, we used the rules in Figure 2.

Context Free Grammar

$$A \to 0A1$$

$$A \to B$$

$$B \to \#$$

The Capital letters on the left are called *variables*. The different rows represent rules (or *productions*) denoted by "→". Anything to the right that are not

variables are called terminals. Here, 0, 1 and # are *terminals*. The variable in the first rule is called the *start variable*.

We can derive 00#11 from the rules above:

$$\begin{aligned} A &\to 0A1 \\ &\to 00A11 \\ &\to 00B11 \\ &\to 00\#11 \end{aligned} \tag{1}$$

Definition of CFG

A CFG is a quad-tuple, given by (V, Σ, R, S) where,
V is a finite set of *variables* or *non-terminals*
Σ is a finite set of *terminals*
R is a set of *productions* or *substitution rules* in the form $A \to \alpha$
S is the *start variable*

Figure 4: Identifying the CFG from the definition

In shorthand, we can write:

$$E \to E + E \mid (E) \mid N$$

$$N \to 0N \mid 1N \mid 0 \mid 1$$

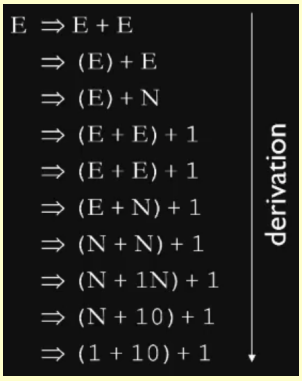

Figure 5: A Derivation based on CFG in Figure 4

We say that $E \overset{*}{\Rightarrow} (1+10)+1$. The '*' above the $\Rightarrow$ represents 'after many productions'.
The Language of a CFG is the set of all strings at the end of a derivation

$$L(G) = \{w : w \in \Sigma^* \; and \; S \overset{*}{\Rightarrow} w\}$$

Parse Trees

Given a CFG, $S \to SS \mid (S) \mid \varepsilon$. We can create a string "(()())".

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow ((S)S) \\ &\Rightarrow ((S)(S)) \\ &\Rightarrow (()(S)) \\ &\Rightarrow (()()) \end{aligned} \tag{2}$$

We can create a parse tree which gives us a more compact representation of a derivation.

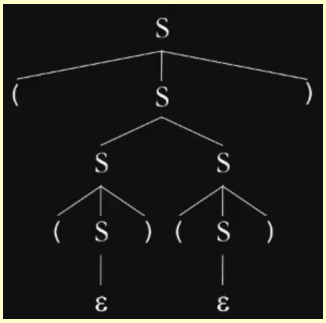

Figure 6: The Parse tree for the derivation 2

.

No matter what steps we do first in a specific derivation of a *specific* string, the parse tree looks the same for all.
When Drawing a parse tree, we can draw it bottom-up or top-to-bottom.

Examples

Some examples were discussed.A detailed discussion will be on the next lecture summary.