CIS 425 - Week 1- Lecture 2

Alexander Eldemir

3 April 2019

1 Regular Grammar

Examples of regular expressions:
- a^* stands for a sequence, possibly empty, of a's.
- a^+ stands for a non-empty sequence of a's.
- {a,b,c}* stands for the set of all the strings generated over the alphabet a, b, and c, including the empty string.
- {a,b,c}$^+$ stands for all the nonempty strings generated over the alphabet a, b, and c.
- $a^*b^*c^*$ stands for a possibly empty sequence of a's, followed by a possibly empty sequence of b's, followed by a possibly empty sequence of c's. The same language can be generated by the following regular grammar:

S ::= aS | A
A ::= bA | U
U ::= cU | ε

Derivation

Remind you that the language generated by a grammar G is:

$$\text{L(G)} = \{\text{w} \in T^* \mid \text{S} \rightarrow^+ \text{w}\}$$

where T is the set of all terminals and S is the root symbol of the grammar.

In order to show that a string belong to a language, i.e. proving that w $\in$ L(G), we show a derivation. For example, for w being the string aabbbc we have:

S $\rightarrow$aS $\rightarrow$aaS $\rightarrow$aaA $\rightarrow$aabA $\rightarrow$aabbA $\rightarrow$aabbbA $\rightarrow$aabbbU $\rightarrow$aabbbcU $\rightarrow$aabbbc

The derivation shows that w does indeed belong to L(G).

Parse Tree

Another way to show that a string is legal is to build its parse tree. This is indeed what the compiler does. The parse tree for the string *aabbbc* is shown

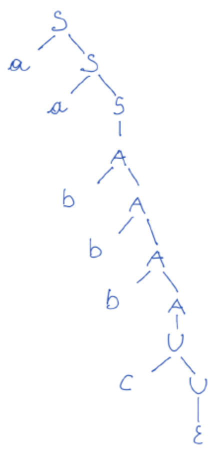

Figure 1: Parse tree for the string aabbbc

in Figure 1. Note that the tree depicted in Figure 2 is not a parse tree because the grammar does not contain the production

A ::= b U

Regular grammars correspond to Finite State Automata (FSA). The FSA accepting the language $a^*b^*c^*$ is given in Figure 3.

2 Context-Free Grammar

A contex-free grammar relaxes some restrictions of regular grammars. In particular, the right-hand side of a production can be any string over the alphabet $T \cup NT$:

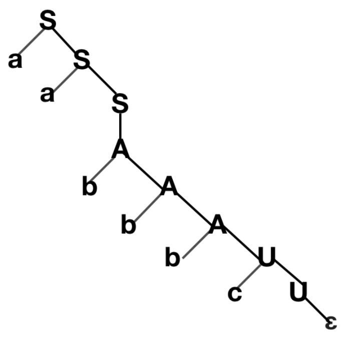

Figure 2: Wrong parse tree for the string aabbbc

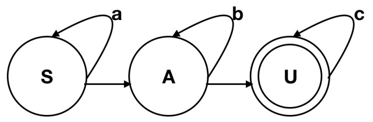

Figure 3: FSA for the language $a^*b^*c^*$

Definition 1 *A* context-free grammar *(CFG) has $l \in NT$ and $r = (T \cup NT)^*$.*

Example 2 The CFG grammar defining the language $\{a^n b^n \mid n \geq 1\}$ is

$$S ::= aSb \mid ab$$

. The CFG grammar defining the language $\{a^n b^n \mid n \geq 0\}$ is

$$S ::= aSb \mid \epsilon$$

. Note that there isn't a unique way to give a grammar. One can always introduce more non terminals as in the following grammar:

$$\begin{array}{lcl} S & ::= & aSb \mid X \\ X & ::= & ab \end{array}$$

Context-free grammars correspond to push down automata.

3 Context-sensitive grammars

Context-sensitive grammars do not impose any restrictions on the left and right-hand side of a production. Such a grammar is needed to recognize a language of the form

$$a^n b^n c^n$$

Intuitively, one could also argue that the above language cannot be context-free since one needs an additional memory to remember the number of b's. This is not possible in a PDA.

4 How to solve the ambiguity of grammars

Consider the following grammar of arithmetic expressions:

$$E ::= a \mid b \mid c \mid E + E$$

The expression $a+b+c$ belong to the language of arithmetic, as shown in Figure 4. However, note that the expression can be parsed in a different way. We call such a grammar ambiguous.

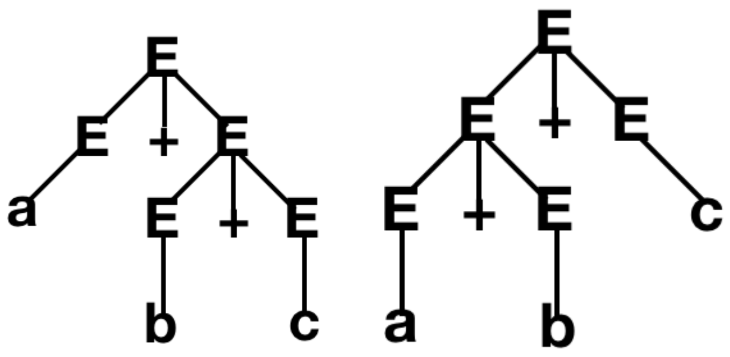

Figure 4: Parse trees for the string a+b+c

Definition 3 *A grammar is ambiguous is there is exists a string in the language with more than one parse tree.*

We want to avoid ambiguous grammars because the compiler will not know what code to generate.

To avoid this problem the grammar can be rewritten to impose the use of parentheses. The user would have to write either $(a + (b + c))$ or $((a + b) + c)$.

Left-associativity

Another solution is to rely on a predefined default case, which in this case would be $(a + b) + c$. In other words, we assume addition to be left associative. This can be directly expressed in the grammar by making use of an additional nonterminal:

$$\begin{array}{lcl} E & ::= & E + F \mid F \\ F & ::= & a \mid b \mid c \end{array}$$

Priority

Let us now extend the language with multiplication:

$$E ::= a \mid b \mid c \mid E + E \mid E * E$$

We assume that both + and * are left-associative, however, we still have one more decision to take. Consider the expression

a + b * c

do we read it as

a + (b * c) or (a + b) * c ?

The default case, used by a compiler, is that multiplication has higher priority over addition, therefore a + (b * c) will be the default reading.

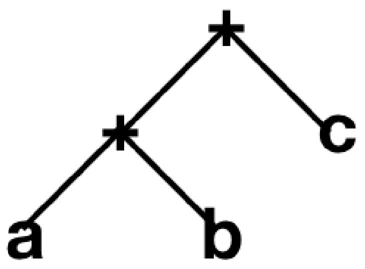

Figure 5: Abstract syntax tree for $a * b + c$

Non Ambiguous Grammar

We can express both association and priority in an non ambiguous grammar, as defined below:

$$\begin{array}{lcl} E & ::= & E + F \mid F \\ F & ::= & F * T \mid T \\ T & ::= & a \mid b \mid c \end{array}$$

Abstract Syntax Trees

After having determined that the program is syntactically correct, a compiler translates the parse tree into a structure containing only terminal symbols, called *abstract syntax tree*. For example, the abstract syntax tree representing the expression $a * b + c$ is given in Figure 5.

Infix, Prefix and Postfix

So far we mainly made use of infix notation for expressions, however languages use other notations:
Prefix : (+ a b), (+ (+ a b) c)
Postfix : (a b +)

References

The following have been suggested by students:

`http://kilby.stanford.edu/~rvg/154/handouts/grammars.html`

`http://www.cs.ru.nl/~herman/onderwijs/FLGA/lecture6.pdf`

`https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm`