# CS 241 Lecture 13

Graham Cooper

June 17th, 2015

Recall:
$S \rightarrow$ S OP S | a | b | c
OP $\rightarrow +| - | * |/$

Leftmost, rightmost derivation

Derivations can be expressed naturally and succinctly as a tree structure

For every leftmost (or rightmost) derivation, there is a unique parse tree.

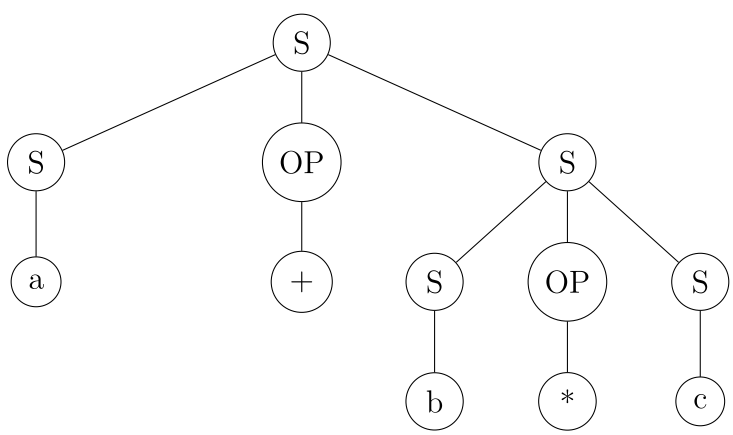
Example: Leftmost derivation for a+b*c

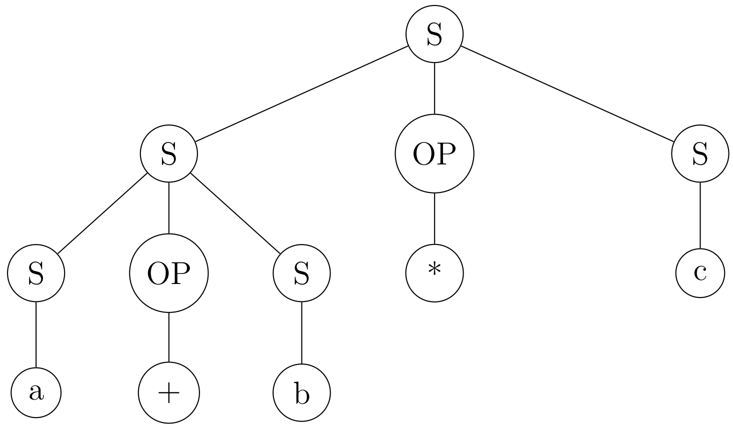
$S \implies$ S Op S $\implies$ a Op S $\implies$ a + S $\implies$ a + S Op S
$\implies$ a + b Op S $\implies$ a + b * S $\implies$ a + b * c

OR

$S \implies$ S Op S $\implies$ S Op S Op S $\implies$ a Op S Op S $\implies$ a + S Op S
$\implies$ a + b Op S $\implies$ a + b * S $\implies$ a + b * c

These correspond to different parse trees!






A grammar for whihc some word has more than one distinct leftmost derivation (equiv. $>$ 1 distinct parse tree) is called ambiguous.
$S \implies SOpS|a|b|c$
$Op \to +|-|*|/$
(The above is an ambiguous grammar)
If we only care whether w $\in$ L(G), ambiguity does not matter

But as compiler writers, we want to know why w $\in$ L(G), ie, the derivation (or parse tree) matters

<u>**WHY?**</u> The shape of the parse tree describes the meaning of the string with respect to the grammar.

So a word with $>$ 1 parse tree may have $>$ 1 meaning.

The first tree above means that a + (b * c), but the second one is suggesting we are doing the plus first (a+b)*c

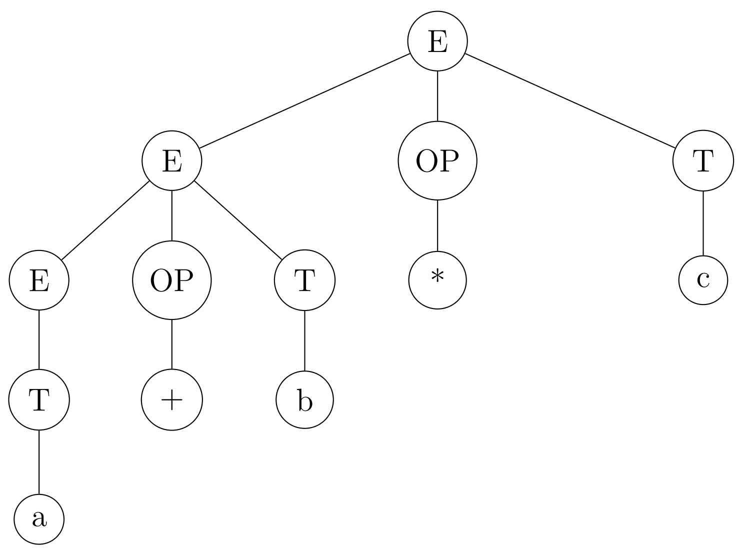
So a + b * c could mean (a + b) * c or a + (b*c)

<u>**What do we do?**</u>

1. Use heuristics ("precedence") to guide the derivation process
2. Make the grammar unambiguous

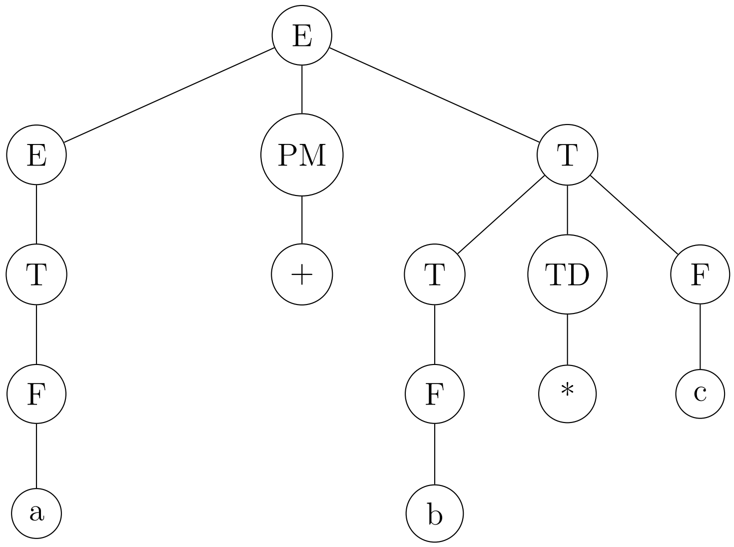
$E \rightarrow$ E OP T — T
$T \rightarrow$ a | b | c
$OP \rightarrow$ + | - | * | /

a + b * c:
E $\Longrightarrow$ E OP T $\Longrightarrow$ E OP T OP T $\Longrightarrow$ T OP T OP T $\Longrightarrow$
a OP T OP T $\Longrightarrow$ a + T OP T $\Longrightarrow$ a + b OP T
$\Longrightarrow$ a + b * T $\Longrightarrow$ a + b * c




Strict left-to-right precendence.

What if we want to give *, / precedence over +, - ?
E $\rightarrow$ E PM T | T
PM $\rightarrow$ + | -
T $\rightarrow$ T TD F | F
TD $\rightarrow$ * | /
This way we have plus and minus on the left and product/division on the right.
a + b * c
E $\Longrightarrow$ E PM T $\Longrightarrow$ T PM T $\Longrightarrow$ F PM T $\Longrightarrow$
a PM T $\Longrightarrow$ a + T $\Longrightarrow$ a + T TD F $\Longrightarrow$ a + F TD F
$\Longrightarrow$ a + b TD F $\Longrightarrow$ a + b * F $\Longrightarrow$ a + b * c




We were successful in making the b * c have precedence

**Question:** If L is context-free, is there always an unambiguous grammar G such that L = L(G)?

**Answer:** NO! There are Inherently ambiguous languages that only have ambiguous grammars

**Question:** Can we construct a tool that will tell whether a grammar is unambiguous.

**ANSWER:** NO! Undecidable

Equivalence of grammars $G_1$, $G_2$, ie: $L(G_1) =?L(G_2)$ is also undecidable

# Recognizers

Recognizers - what class of computer programs is needed to recognize a CFL?

- Regular Languages: DFA - essentially a program with finite memory

- Context-free languages - NFA + stack - infinite memory, but its use is limited to LIFO order

But we need more than just a yes/no answer!!
- Need the derivation (parse-tree) or an informative error message

Problem of finding the derivation is called parsing.
Given Grammar G, start symbol S, terminal string w,
Find: S $\Longrightarrow$ ... $\Longrightarrow$ w
- or report that there is no derivation

How can we do this?

2 Choices:

1. Forwards - "top down parsing"
   start at S, expand non-terminals until you produce w
2. Backwards - "bottom up parsing"
   Start at w, apply rules in reverse, produce S

Both options seem hard...

# Top-Down Parsing

Start at S, apply grammar rules, produce w

S $\Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow ... \Longrightarrow$ w

Use the stack to store intermediate steps $\alpha_i$ in reverse and match against characters in w.

Invariant: consumed input + reverse(stack contents) = $\alpha_i$ for some i