Parsing:

10/3/18

Given a context-free grammar G and a string w, find a sequence of productions (or derivation tree or sequence of leftmost productions) of G that produces w, or find out that w can't be produced.

E.g.: Given

$$S \to (S) \mid SS \mid \lambda$$

find out if

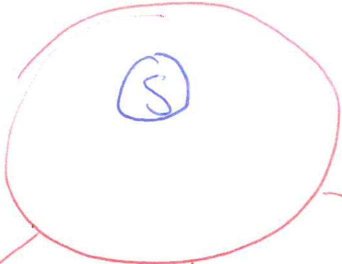
(()())((()))()(()

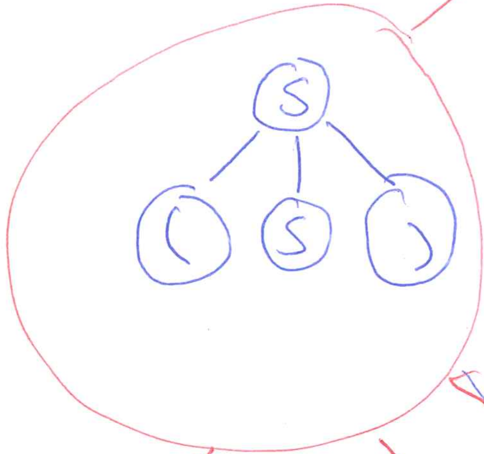
is produced by this grammar.

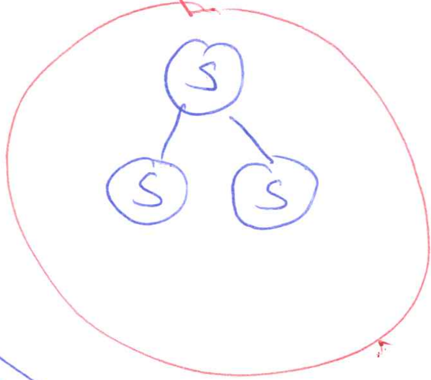
Stupid method: Try everything. Do a breadth-first search on the tree of possible derivation trees.

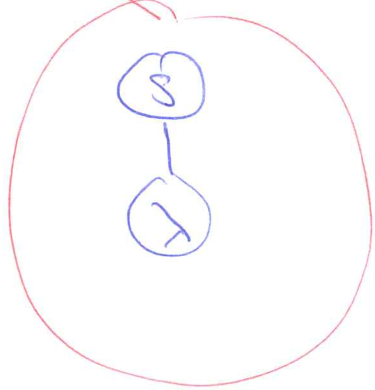
Problems: ① This takes a long time.

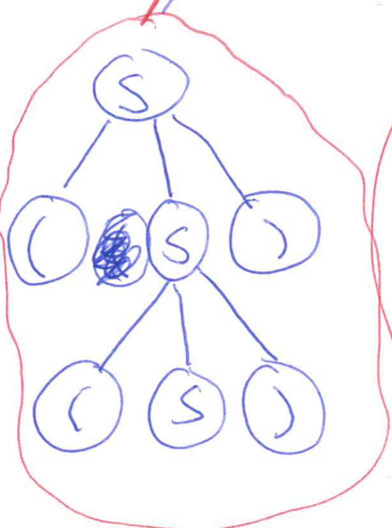
② If the answer is that w can't be produced, we never find out.

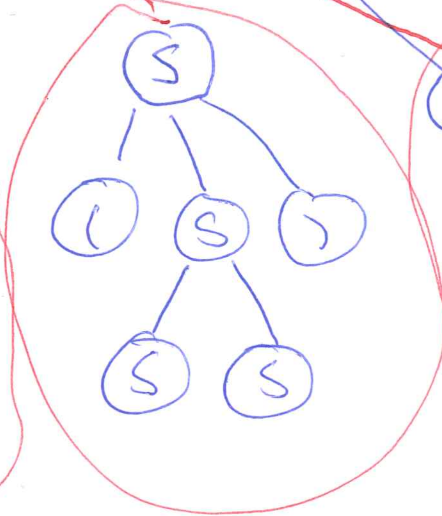

S

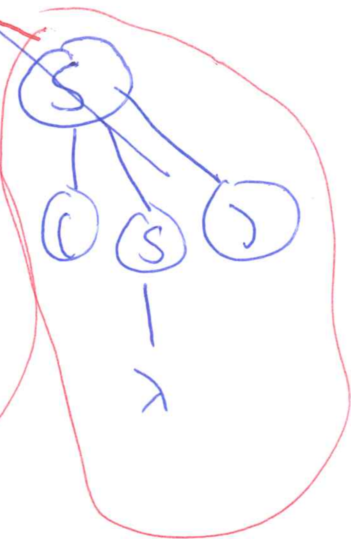

S
(S)


S
S S


S
λ


S
(S)
(S)


S
(S)
S S


S
(S)
λ

How can we solve (2)?

<u>Idea:</u> Modify the grammar so that ~~it~~ it has no productions of the form

$A \to \lambda$ (var $\to \lambda$)

or

$A \to B$ (var $\to$ single var only)

If I don't have productions of these forms, I must "make progress" at every production - I either

a) increase # of letters in the string

or b) increase # of variables. ↑ in this case I can lose 1 var.

i.e. to generate a string w/ 20 letters, the slowest possible method is to add 19 vars in 19 steps and convert vars 1 by 1 to letters in 20 steps

---

total at most 39 steps.

This solves ② since we know we don't need to search farther down than twice the length of the string. (i.e. once we have searched that much of the tree, we can declare failure)

Making ① a little better: pruning the search tree:

a) ~~Always~~ Always do leftmost derivations

b) Once the leftmost letters (before any variable) don't match the leftmost letters of your string, you can stop looking down that branch.

Some ~~to~~ restrictions on our grammar can make them easier to parse

1) Regular grammars are easy to parse

2) s-grammars:

~~a) "linear" every deriv.~~

a) every derivation looks like

$V \rightarrow l(stuff)$ ← first thing must be a letter (not var)

b) Given any variable V and letter ℓ, there is at most one derivation looking like

$$V \to \ell(\text{stuff})$$

i.e. if I have

$$A \to xyBC \mid xAyB,$$

this is <u>not</u> an s-grammar.

These are easy to parse b/c you can look at the first letter that is still in variable form an find out what derivation you use to get it (or find out none is possible)

<u>e.g.</u>: If my string is $w = xy\ldots\ldots$, then the first prod I use must be <u>the</u> production

$$S \to x(\text{stuff})$$

Suppose this was

$$S \to xABAy.$$

Then the next prod would have to be

$A \to y$ (stuff) (and there is at most one of these)

Another way to think of this: pruning takes off all but one branch.

Ambiguity

A grammar G is ambiguous if there is a string w produced by G in more than one way - i.e. w has two different derivation trees.

A language L is inherently ambiguous if every grammar for L ~~G~~ is ambiguous (there is no unambiguous grammar for L)

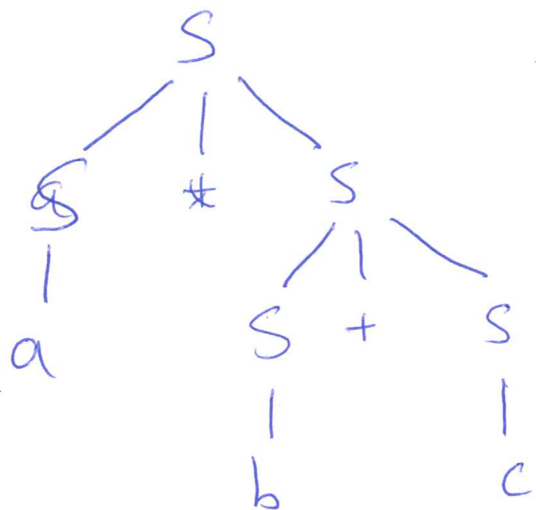
Example: Grammar for restricted arith. express.

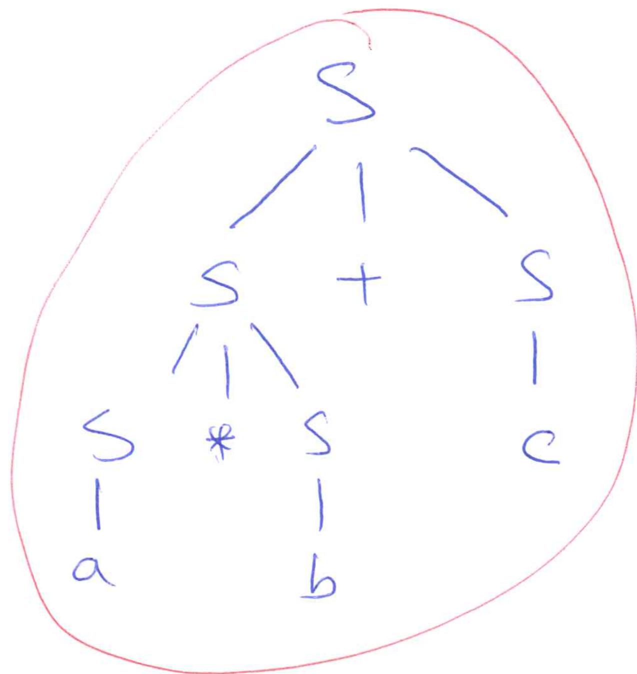
$\Sigma = \{ (, +, *,), a, b, c, d \}$

$$S \longrightarrow (S) \mid \lambda \mid a \mid b \mid c \mid d \mid S+S \mid S*S$$

This is ambiguous since

$a*b+c$ has derivations:





better b/c it reflects our order of operations.

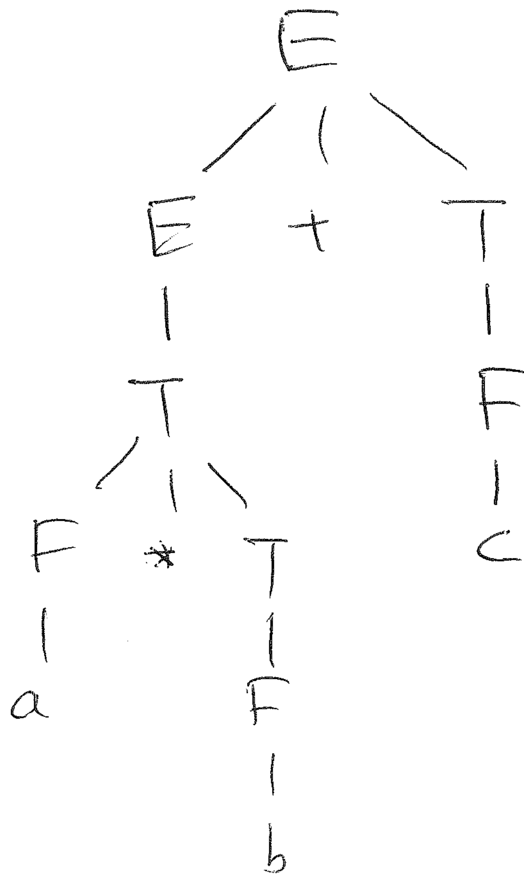
We can fix this at the parsing level - specify which deriv. to try first.

We can also try to make an unambiguous grammar:

$$E \longrightarrow T \mid E+T$$
$$T \longrightarrow F*T \mid F$$
$$F \longrightarrow a \mid b \mid c \mid d \mid (E)$$

E = expression ← start
T = term
F = factor

To parse a*b+c, we have to have



Something that is inherently ambig.

$L = \{ a^x b^y c^z \mid x = y \text{ or } y = z \}$

$S \to AZ \mid XB$

$Z \to cZ \mid \lambda$

$X \to aX \mid \lambda$

$A \to aAb \mid \lambda$

$B \to bBc \mid \lambda$

aabbcc