Context-Free Languages
Thoughts on Assignment 1

CS 331 Programming Languages

Lecture Slides

Wednesday, January 22, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
Formal Languages & Grammars

Topics

- ✓ Basic concepts
- ✓ Introduction to formal languages & grammars
- ✓ The Chomsky hierarchy
- ✓ Regular languages
- ✓ Regular languages & regular expressions
- Context-free languages
- Programming language syntax specification

Review

Regular Languages — Regular Grammars & Languages

A **regular grammar** is a grammar, each of whose productions looks like one of the following.

$A \to \varepsilon$ $\quad$ $A \to b$ $\quad$ $A \to bC$

We allow a production using the same nonterminal twice: $A \to bA$

A **regular language** is a language that is generated by some regular grammar.

This is a regular grammar:

$$S \to \varepsilon$$
$$S \to t$$
$$S \to xB$$
$$B \to yS$$

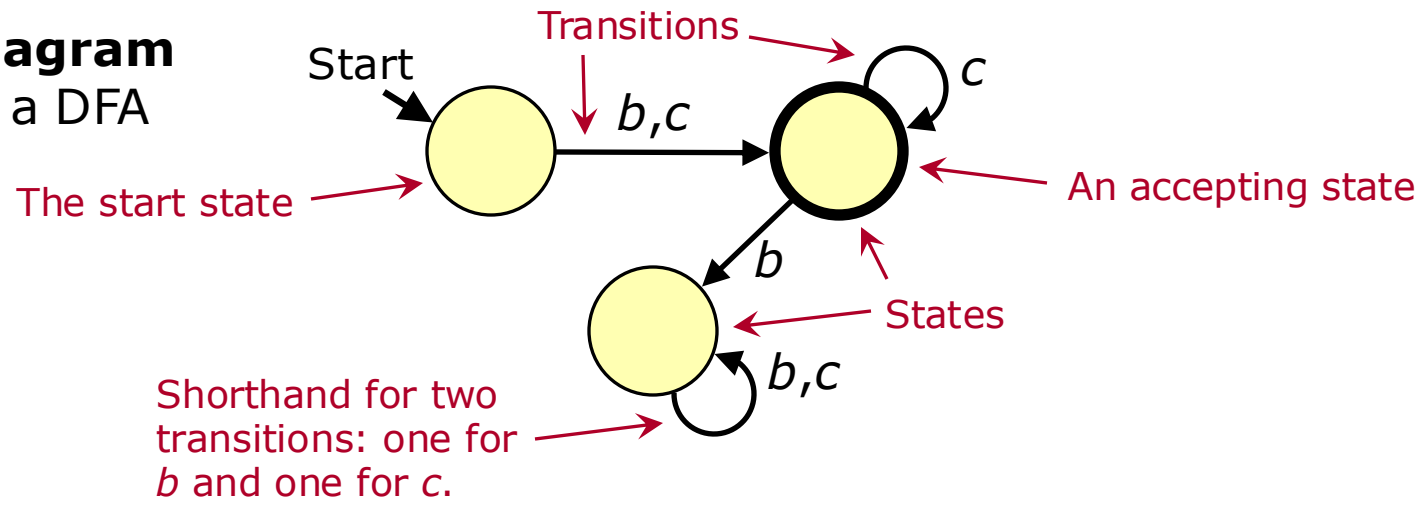
Therefore, the language it generates is a regular language:

$$\{\varepsilon, xy, xyxy, xyxyxy, \ldots, t, xyt, xyxyt, xyxyxyt, \ldots\}$$

Regular Languages — Finite Automata [1/3]

A **deterministic finite automaton** (Latin plural "**automata**"), or **DFA**, is a kind of recognizer for regular languages.

Diagram of a DFA



Rule. For each character in the alphabet, each state has *exactly one* transition leaving it that is associated with that character.

Regular Languages — Finite Automata [2/3]

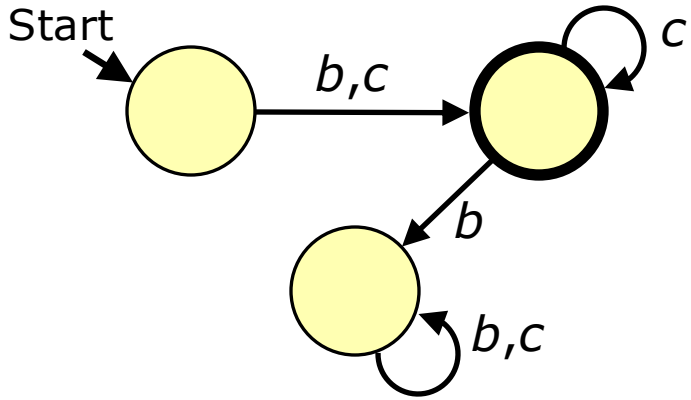
To use a DFA as a recognizer:

- Start in the start state; proceed in a series of steps.
- At each step, read a character from the input and follow the transition from at the current state, labeled with that character.
- If, when we reach the end of the input, we are in an accepting state, then we **accept** the input.

The set of all inputs that are accepted is the language **recognized** by the DFA.

Exercise

4. What language is recognized by the DFA whose diagram is shown?



Regular Languages — Finite Automata [3/3]

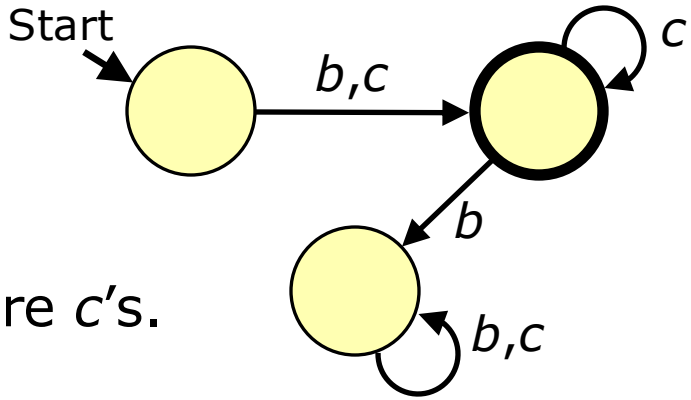
To use a DFA as a recognizer:

- Start in the start state; proceed in a series of steps.
- At each step, read a character from the input and follow the transition from at the current state, labeled with that character.
- If, when we reach the end of the input, we are in an accepting state, then we **accept** the input.

The set of all inputs that are accepted is the language **recognized** by the DFA.

Answer

4. What language is recognized by the DFA whose diagram is shown?



The set of strings consisting either of *b* then zero or more *c*'s, or of one or more *c*'s.

{*b*, *bc*, *bcc*, *bccc*, …, *c*, *cc*, *ccc*, …}

Regular Languages & Regular Expressions — Regexes: Syntax

A **regular expression** is a generator for a regular language.

We specified the syntax of regular expressions by showing how to build them up from small pieces.

- A *single character* is a regular expression: *a*.
- The *empty string* is a regular expression: ε.

If *A* and *B* are regular expressions, then so are the following.

- *A** ← **Kleene star** (say KLAY-nee)
- *AB*
- *A*|*B*

The above are listed from high to low precedence. All are left-associative. Override precedence using parentheses.

- (*A*)

Here is a regular expression: (*a*|*x*)**cb*

Regular Languages & Regular Expressions — Regexes: Semantics

A regular expression is said to **match** certain strings.

Semantics of regular expressions:

- A single character matches itself, and nothing else.
- The empty string matches itself, and nothing else.
- *A** matches the concatenation of zero of more strings, each of which is matched by *A*.
  - Note that *A** matches the empty string, no matter what *A* is.
- *AB* matches the concatenation of any string matched by *A* and any string matched by *B*.
- *A*|*B* matches all strings matched by *A* and also all strings matched by *B*.
- (*A*) matches the same strings that are matched by *A*.

The language **generated** by a regular expression consists of all strings that it matches.

Regular Languages & Regular Expressions — Regexes in Practice

Regular expression libraries typically include shortcuts in their syntax. The ones below do *not* change which languages can be generated. They may be used this semester in answers to assignments/quizzes/exams.

`.`	Matches any single character
`[abcd]`	Matches a single `a`, `b`, `c`, or `d`, like `a|b|c|d`
`[a-d]`	Same as above
`[^a-d]`	Matches anything except `a`, `b`, `c`, or `d`
`x+`	Matches one or more `x` characters: `x`, `xx`, `xxx`, `xxxx`, etc.
`x?`	Matches zero or one `x` characters—an optional `x`
`\.`	Matches a dot: "."
`\\`	Matches a backslash: "\"

Using a backslash in this way is called **escaping**.

Slashes might be delimiters: `/[0-9]/` matches any ASCII digit.

Regular Languages & Regular Expressions — Wrap-Up

The regular languages are precisely:

- The languages generated by regular grammars.
- The languages recognized by DFAs.
- The languages generated by regular expressions (in the strict sense, with no features beyond those covered).

That is, these three classes of languages are identical.

We will use ideas about regular languages when we do **lexical analysis** (**lexing**): breaking up a program into **lexemes** (words, roughly).

Context-Free Languages

Context-Free Languages
Introduction

We now turn to the second smallest class of languages in the Chomsky hierarchy: the *context-free languages*.

Context-free languages are important because, for most programming languages, the set of all syntactically correct programs forms a context-free language.

And for those PLs that do not have this property, it is still common for the *techniques* used in dealing with context-free languages to be useful.

Context-free languages, and the associated grammars, are thus important in **parsing**: determining whether input (for example, a program) is syntactically correct, and, if so, finding its structure.

Context-Free Grammars & Languages — Definitions

A **context-free grammar** (**CFG**) is a grammar, each of whose productions has a left-hand side consisting of a single nonterminal.

All of the grammars we have looked at have been CFGs. In particular, every regular grammar is a CFG.

A **context-free language** (**CFL**) is a language that is generated by some context-free grammar.

Every regular language is a CFL. But there are context-free languages that are not regular.

Context-Free Grammars & Languages — Examples [1/2]

Here is a CFG.

$S \rightarrow aSa$

$S \rightarrow b$

"Context-free" refers to the fact that a nonterminal can be expanded at any time. Chomsky defined a larger class of grammars, *context-sensitive grammars*, in which productions can sometimes only be applied if a nonterminal has certain characters around it, that is, only in a certain *context*. We will not study context-sensitive grammars this semester.

This grammar generates the following language.

$$\{b, aba, aabaa, aaabaaa, aaaabaaaa, \ldots\}$$

We can also write this language as follows.

$$\{ a^k b a^k \mid k \geq 0 \}$$

As we have noted, this is not a regular language. But since it is generated by a CFG, it is a CFL.

Context-Free Grammars & Languages — Examples [2/2]

Regular grammars are not powerful enough to handle things like matching parentheses. But CFGs are powerful enough.

Consider the following grammar—where "(" and ")" are terminal symbols.

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow \varepsilon$

The language generated by the above grammar consists of all sequences of properly matched parentheses. For example, here is one string in this language.

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

Context-Free Grammars & Languages — Concise Notation

It is common for a CFG to have multiple productions with the same left-hand side. As a shortcut, we allow writing the left-hand side and the arrow only once, with the various right-hand sides separated by vertical bars ("|").

For example, our first CFG can be rewritten as follows.

$S \to aSa$
$S \to b$

$S \to aSa \mid b$

We might place the right-hand sides on separate lines.

$S \to aSa$
$\quad \mid b$

Parse Trees — Definition [1/3]

Parsing involves finding the structure of a program. One way to represent this structure is to use a **parse tree**.

We introduce parse trees using *Grammar A*, below. To the right is a derivation for the string *ppy* based on this CFG.

Grammar A

$S \rightarrow AB$

$A \rightarrow pA \mid \varepsilon$

$B \rightarrow x \mid y$

Derivation of *ppy*

$\underline{S}$

$A\underline{B}$

$\underline{A}y$

$p\underline{A}y$

$pp\underline{A}y$

ppy

There are no parse trees on this slide! See the next slide for a drawing of a parse tree.

Parse Trees — Definition [2/3]

Grammar A

$S \rightarrow AB$

$A \rightarrow pA \mid \varepsilon$

$B \rightarrow x \mid y$

Derivation of *ppy*

$\underline{S}$

$A\underline{B}$

$\underline{A}y$

$p\underline{A}y$

$pp\underline{A}y$

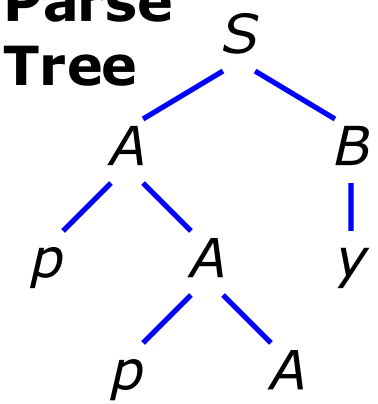
ppy

A **parse tree** is a rooted tree with one symbol in each node, based on a derivation.

- The root node holds the start symbol.
- The symbols a nonterminal is expanded into become its children—left to right, one symbol per tree node.

Here is a parse tree based on the above derivation.

Parse Tree



Every terminal symbol is in a leaf of the parse tree. We can read off the final string by looking at the leaves that contain terminal symbols.

Parse Trees — Definition [3/3]

Another grammar, derivation, and associated parse tree. Here, "+" is a terminal symbol.

Grammar B

$S \rightarrow S+S \mid n$

Derivation of *n*+*n*+*n*

$\underline{S}$

$\underline{S}+S$

$\underline{S}+S+S$

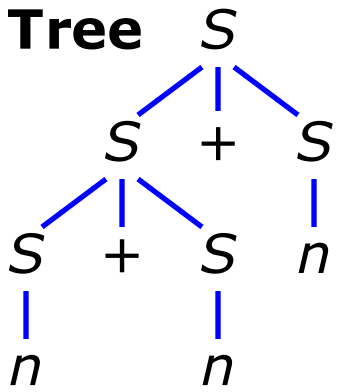
$n+\underline{S}+S$

$n+n+\underline{S}$

$n+n+n$

Remember: a *derivation* is a list of strings; a *parse tree* is a tree.

Parse Tree



Again, we can read off the final string by looking at the leaves that contain terminal symbols.

Parse Trees — TRY IT (Exercises)

Grammar C

$S \rightarrow XY$

$X \rightarrow a \mid b$

$Y \rightarrow cY \mid c$

Derivation of *ac*

$\underline{S}$

$\underline{X}Y$

$a\underline{Y}$

ac

Exercises

1. Based on Grammar C and the given derivation, draw a parse tree for the string *ac*.
2. Based on Grammar C, draw a parse tree for the string *bcc*.

Parse Trees — TRY IT (Answers)

Grammar C

$S \rightarrow XY$

$X \rightarrow a \mid b$

$Y \rightarrow cY \mid c$

Derivation of *ac*

$\underline{S}$

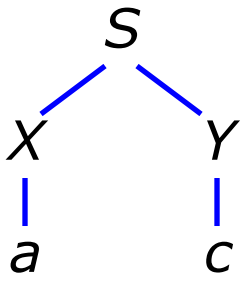
$\underline{X}Y$

$a\underline{Y}$

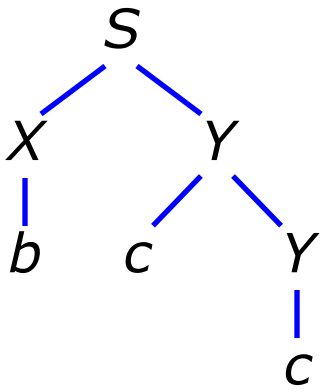
ac

Answers

1. Based on Grammar C and the given derivation, draw a parse tree for the string *ac*.



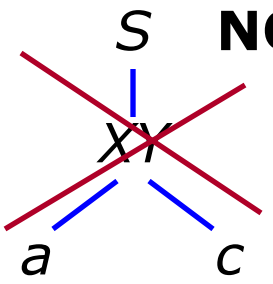
2. Based on Grammar C, draw a parse tree for the string *bcc*.



In both exercises, the answer is unique.

Parse Trees — TRY IT (Note)

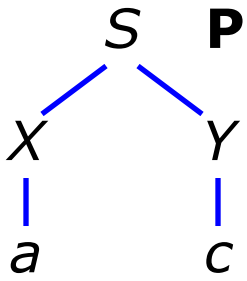
If you drew something like this for Exercise 1 …



NOT a Parse Tree

… then be aware that the above is not a parse tree.

A parse tree has *one symbol in each node*.

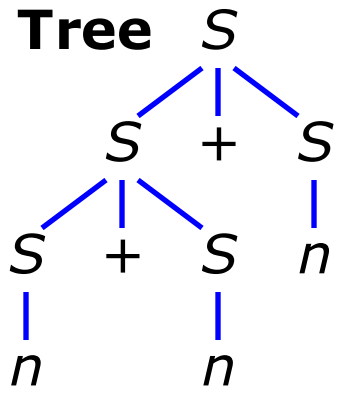


Parse Tree

Grammar B

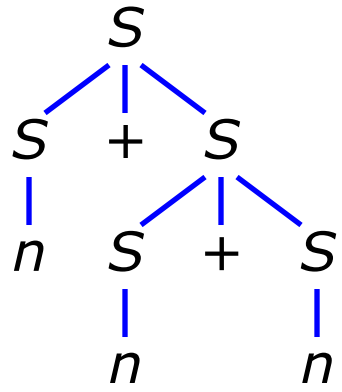
$S \rightarrow S+S \mid n$

Parse Tree



There is another parse tree for $n+n+n$ based on the above grammar. It is shown below.

Another Parse Tree



This means that the string $n+n+n$ has two possible structures.

A CFG in which a single string has more than one parse tree, is said to be **ambiguous**.

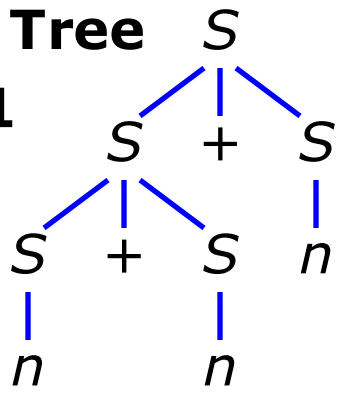
So Grammar B is ambiguous.

Ambiguity — Eliminating Ambiguity [1/3]

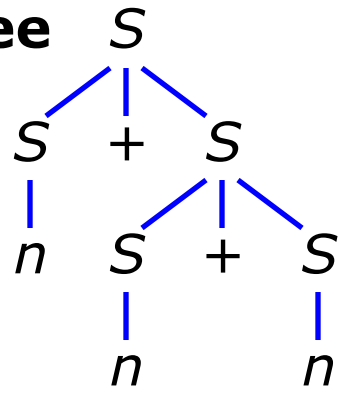
Grammar B

$S \rightarrow S{+}S \mid n$

Parse Tree #1



Parse Tree #2



Ambiguity is a property of *grammars*, not of languages. And it is generally a property that we do not like.

Grammar B is ambiguous; however, in this case we can actually find a non-ambiguous CFG that generates the same language.

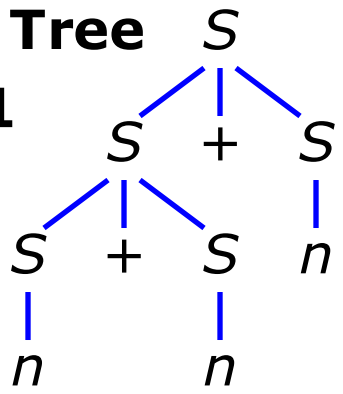
Before finding such a grammar, we first note that, assuming "+" represents addition, we prefer parse tree #1, since it expresses the left associativity that we usually want addition to have: $n{+}n{+}n = (n{+}n){+}n$.

Ambiguity — Eliminating Ambiguity [2/3]

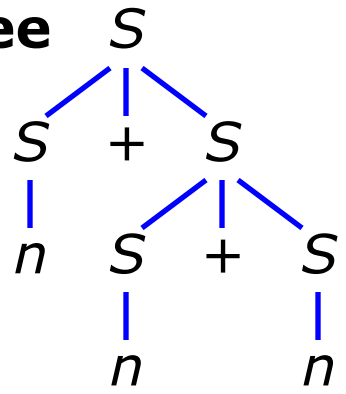
Grammar B

$S \rightarrow S{+}S \mid n$

Parse Tree #1



Parse Tree #2



Below is a non-ambiguous grammar that generates the same language and expresses the left-associativity of "+". Also shown: a derivation of $n{+}n{+}n$ and the unique parse tree.

Grammar B′

$S \rightarrow S{+}n \mid n$

Derivation

$\underline{S}$

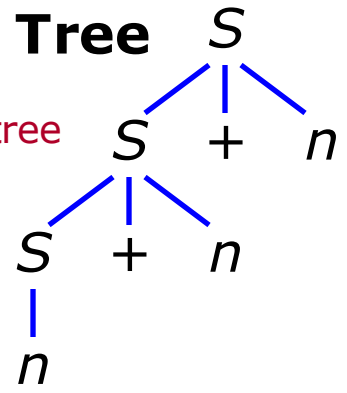
$\underline{S}{+}n$

$\underline{S}{+}n{+}n$

$n{+}n{+}n$

Parse Tree

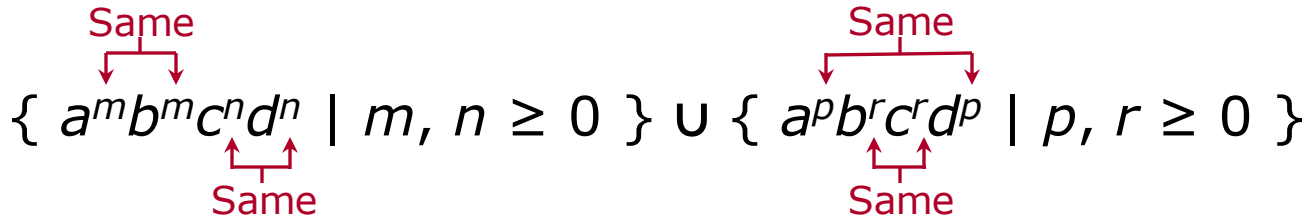
This parse tree is unique!



Ambiguity — Eliminating Ambiguity [3/3]

Sometimes ambiguity cannot be eliminated. There are CFLs that are only generated by ambiguous CFGs. Such a CFL is **inherently ambiguous**.

Here is a standard example of an inherently ambiguous CFL.



$$\{\, a^m b^m c^n d^n \mid m, n \geq 0 \,\} \cup \{\, a^p b^r c^r d^p \mid p, r \geq 0 \,\}$$

It can be demonstrated that, no matter how we write a CFG for this language, there will be some string that has two different parse trees.

Remember:

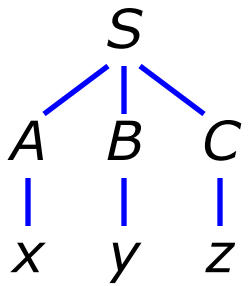
- **Ambiguity** is a property of *grammars* (CFGs).
- **Inherent ambiguity** is a property of *languages* (CFLs).

Leftmost & Rightmost Derivations [1/3]

The CFG below generates only *xyz*. There are multiple derivations.

Grammar D	**Derivation #1**	**Derivation #2**	**Derivation #3**
$S \rightarrow ABC$	$\underline{S}$	$\underline{S}$	$\underline{S}$
$A \rightarrow x$	$\underline{A}BC$	$AB\underline{C}$	$A\underline{B}C$
$B \rightarrow y$	$x\underline{B}C$	$A\underline{B}z$	$Ay\underline{C}$
$C \rightarrow z$	$xy\underline{C}$	$\underline{A}yz$	$\underline{A}yz$
	xyz	xyz	xyz

But there is only one parse tree. Grammar D is *not* ambiguous.



Context-Free Languages
Leftmost & Rightmost Derivations [2/3]

Even though they correspond to the same parse tree, these three derivations of *xyz* differ in a noteworthy way.

- In derivation #1, the leftmost nonterminal it expanded at each step. We call this a **leftmost derivation**.
- In derivation #2, the rightmost nonterminal it expanded at each step. We call this a **rightmost derivation**.
- Derivation #3 is neither leftmost nor rightmost.

#1: Leftmost derivation	**#2: Rightmost derivation**	**#3: Neither**
S	*S*	*S*
ABC	*ABC*	*ABC*
xBC	*ABz*	*AyC*
xyC	*Ayz*	*Ayz*
xyz	*xyz*	*xyz*

Context-Free Languages
Leftmost & Rightmost Derivations [3/3]

#1: Leftmost derivation	#2: Rightmost derivation	#3: Neither
$\underline{S}$	$\underline{S}$	$\underline{S}$
$\underline{A}BC$	$AB\underline{C}$	$A\underline{B}C$
$x\underline{B}C$	$A\underline{B}z$	$Ay\underline{C}$
$xy\underline{C}$	$\underline{A}yz$	$\underline{A}yz$
xyz	xyz	xyz

These concepts will come up later, in our study of parsing.

- A parser goes through the steps required to find a derivation. Some parsers go through the derivation in forward order, **expanding** the leftmost nonterminal first, producing a leftmost derivation.
- Other parsers go though the derivation in reverse, repeatedly **contracting** a substring to a nonterminal. Typically, the left part of the input is contracted first. Viewed in forward order, the rightmost nonterminal is expanded first, producing a rightmost derivation.

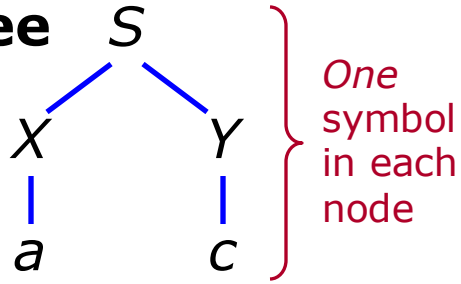
Do not confuse parse trees with derivations!

- A parse tree is a *rooted tree*.
- A derivation is a *list of strings*.

For *every parse tree*, there is a corresponding leftmost derivation and rightmost derivation.

Ambiguity is about multiple parse trees, not multiple derivations.

Parse Tree



Leftmost Derivation

$\underline{S}$

$\underline{X}Y$

$a\underline{Y}$

ac

Rightmost Derivation

$\underline{S}$

$X\underline{Y}$

$\underline{X}c$

ac

Thoughts on Assignment 1

Thoughts on Assignment 1

Three quick notes:

- Turn in your work as a **PDF** file.
- The point of the first exercise is making sure you are able to execute Lua code. You will be writing a lot of Lua this semester. Now is the time to be sure you can execute it.
- We have covered the required material for all parts of the assignment, except the last exercise. This concerns something called *BNF grammars*, which we will discuss next time.