

Regular Languages continued

Context-Free Languages

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Wednesday, January 23, 2019


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

© 2017–2019 Glenn G. Chappell

Review

Formal Languages & Grammars

Grammar


1. $S \rightarrow xxSy$  Each line is a **production**.
2. $S \rightarrow a$
3. $S \rightarrow \varepsilon$

To use a grammar:

- Begin with the start symbol.
- Repeat:
 - Apply a production, replacing the left-hand side with the right-hand side.
- We can stop only when there are no more nonterminals.

Derivation of xxxxyy

\underline{S}
1 $xx\underline{S}y$
1 $xxxx\underline{S}yy$
3 $xxxxyy$

No " ε "
appears here. 

The result is a **derivation** of the final string.

The language **generated** by a grammar consists of all strings for which there is a derivation.

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

$$A \rightarrow \varepsilon$$

$$A \rightarrow b$$

$$A \rightarrow bC$$

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

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

This grammar is regular:

$$\begin{aligned} S &\rightarrow \varepsilon \\ S &\rightarrow t \\ S &\rightarrow xB \\ B &\rightarrow yS \end{aligned}$$

The language it generates is therefore a regular language:

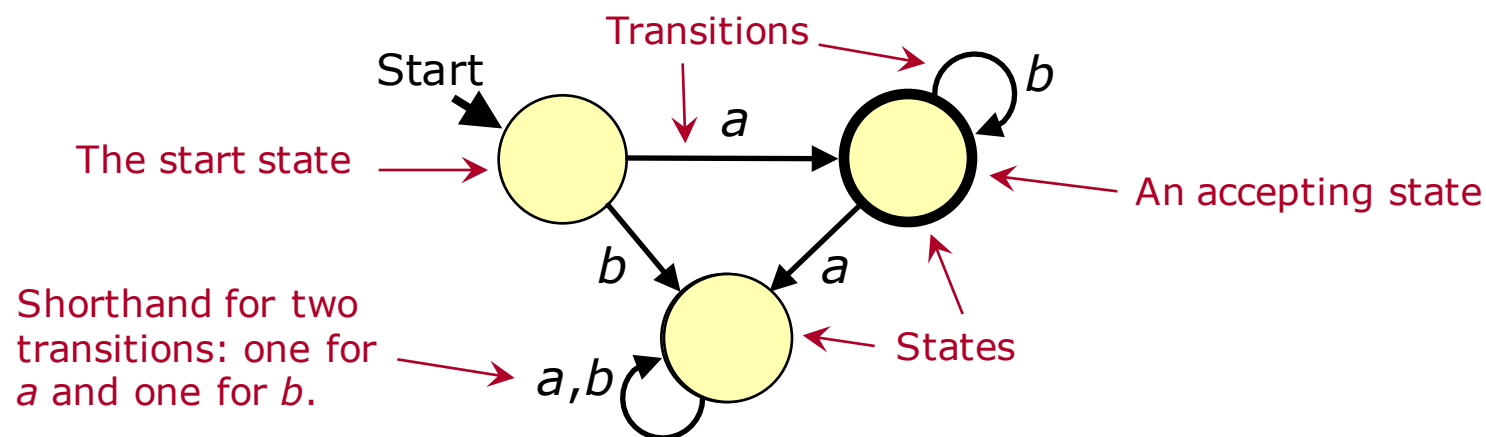
$$\{\varepsilon, xy, xyxy, xyxyxy, \dots, t, xyt, xyxyt, xyxyxyt, \dots\}$$

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

A DFA has:

- A finite collection of **states**. One is the **start state**. Some may be **accepting states**.
- **Transitions**, each beginning at a state, ending at a state, and associated with a character in the alphabet.

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



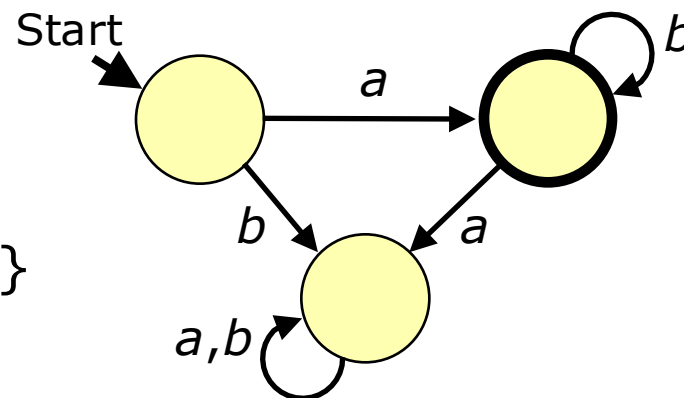
To use a DFA as a recognizer:

- Start in the start state; proceed in a series of steps.
- At each step, read a character from the input and follow the transition beginning at the current state and labeled with the character that was read.
- 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.

The DFA shown recognizes the following language:

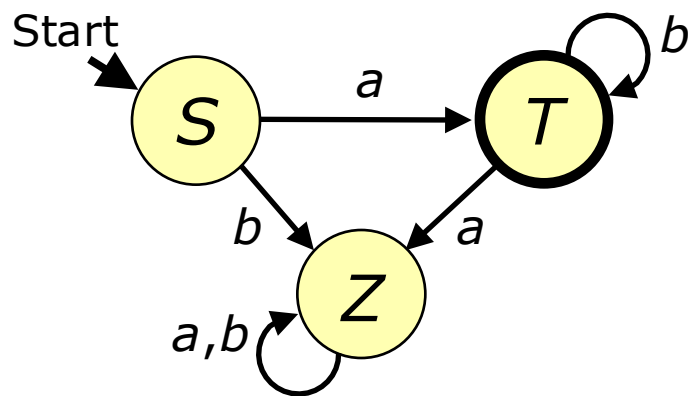
$\{a, ab, abb, abbb, abbbb, abbbbbb, \dots\}$



Given a DFA, we can construct a regular grammar that generates the same language the automaton recognizes.

- For each state, 1 nonterminal. Start state gives start symbol.
- For each transition, 1 production. A to B by x gives $A \rightarrow xB$.
- For each accepting state, 1 production. A accepting gives $A \rightarrow \varepsilon$.

DFA with states labeled



Regular Grammar

$$S \rightarrow aT$$

$$S \rightarrow bZ$$

$$T \rightarrow aZ$$

$$T \rightarrow bT$$

$$Z \rightarrow aZ$$

$$Z \rightarrow bZ$$

$$T \rightarrow \varepsilon$$

Productions
involving Z can
never be used
in a derivation,
so we may
ignore them.

Language Recognized/Generated

$\{a, ab, abb, abbb, abbbb, abbbbb, \dots\}$

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

We specify 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: $(a|x)^*cb$

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 or 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.

Fact. The languages that are generated by regular expressions are precisely the regular languages.

That is:

- For each regular expression, the language it generates is a regular language.
- For each regular language, there is regular expression that generates it.

Consider the language containing all strings consisting of zero or more x 's, followed by either y or z . That is,

$$\{ y, xy, xxy, xxxy, xxxxy, \dots, z, xz, xxz, xxxz, xxxxz, \dots \}$$

This is a regular language.

Exercises

1. Write a regular expression that generates the above language.
2. Draw the diagram of a DFA that recognizes this language.

Review

Regular Languages — TRY IT [2/2]

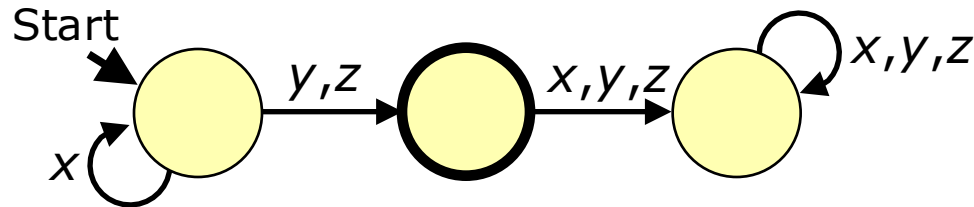
$\{ y, xy, xxy, xxxy, xxxxy, \dots, z, xz, xxz, xxxz, xxxxz, \dots \}$

Answers

1. Write a regular expression that generates the above language.

$x^*(y|z)$

2. Draw the diagram of a DFA that recognizes this language.



Most regular-expression libraries accept something like the syntax we have described, except that “ ϵ ” is replaced by an actual empty string. In addition, a number of common shortcuts are almost universally used.

First, “.” matches any single character, except possibly the end-of-line character.

Second, brackets with a list of characters between them will match any one of the characters in the list. The following two regular expressions match the same strings.

`[qwerty]`

`(q|w|e|r|t|y)`

Parentheses ()

Brackets []

Braces { }

Angle brackets < >

Regular Languages

Regular Expressions — In Practice [2/5]

With the bracket syntax, “-” specifies a range of consecutive characters. The following expressions match the same strings.

```
[ 0-9 ]
```

```
[ 0123456789 ]
```

```
( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )
```

The following will match any single letter.

```
[ A-Za-z ]
```

Placing “^” just after the opening bracket means that all characters *not* in the list are matched. So this regular expression

```
[ ^A-Za-z ]
```

matches any single character that is *not* an ASCII letter.

Third, “+” means one-or-more, in the same way that “*” means zero-or-more. So the following two expressions match the same strings.

`(abc) +`

`abc (abc) *`

Fourth, “?” means zero-or-one. So the following two expressions match the same strings.

`x (abc) ?`

`x | xabc`

Regular Languages

Regular Expressions — In Practice [4/5]

Last, the various special characters above are treated as ordinary characters when preceded by a backslash(`\`). For example, `"."` matches any character, while `"\."` matches only `"."`.

The rules for backslash escaping vary from one regular-expression library to another. Read your library's documentation!

To match a single backslash, use an escaped backslash: `"\\"`.

The extras we have mentioned so far are all just shortcuts. They make regular expressions more convenient, but they do not allow for the generation of any new languages.

In answers to assignments, quizzes, and exams in this class, I will allow any of the shortcuts we have mentioned so far.

Many programming languages & libraries include facilities that make their “regular expressions”—so called—decidedly non-regular. That is, they allow for the generation of languages that are not regular.

One way to do this is to allow a requirement that two sections of a string are the same. For example, the following, used in Perl or Python, matches strings `b`, `aba`, `aabaa`, `aaabaaa`, etc.

`(a*)b\1`

The language generated by this expression is the same language given earlier as an example of a language that is not regular. For the purposes of this class, we do *not* consider the above to be a regular expression.

Regular Languages

Wrap-Up

Regular languages form the smallest of the four classes of languages in the Chomsky Hierarchy.

Regular languages are used in text search/replace, and in lexical analysis (lexing).

A **regular language** is one that can be generated by a **regular grammar**, which is a grammar in which every production has one of the following three forms.

$$A \rightarrow \varepsilon$$

$$A \rightarrow b$$

$$A \rightarrow bC$$

Regular languages are precisely those languages that are recognized by some **DFA**.

They are also the languages that can be generated by a **regular expression**—in the strict sense of the term.

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.

These languages, and the associated grammars, are thus important in **parsing**: determining whether a program is syntactically correct, and, if so, finding its structure.

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.

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.

Here is a CFG.

$$S \rightarrow aSa$$
$$S \rightarrow b$$

This grammar generates the following language.

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

We can also write this language as follows.

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

This is not a regular language. But since it is generated by a CFG, it is a CFL.

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.

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

A CFG may have a number of productions with the same left-hand side. As a shortcut, we write 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.

$$S \rightarrow aSa \mid b$$

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

$$\begin{array}{l} S \rightarrow aSa \\ \quad \mid b \end{array}$$

Context-Free Languages

Parse Trees — Definition [1/3]

Parsing involves finding the structure of a program.
We can represent this structure using a **parse tree**, also called a **concrete syntax tree (CST)**.

Later, we will cover a related structure called an *abstract syntax tree (AST)*.

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*

$$\underline{S}$$
$$A\underline{B}$$
$$A\underline{y}$$
$$pA\underline{y}$$
$$ppA\underline{y}$$
$$ppy$$

Context-Free Languages

Parse Trees — Definition [2/3]

Grammar A

$S \rightarrow AB$

$A \rightarrow pA \mid \varepsilon$

$B \rightarrow x \mid y$

A **parse tree** is a rooted tree in which each node holds a single symbol.

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

Here is a parse tree based on the derivation.

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.

Derivation of ppy

S

A B

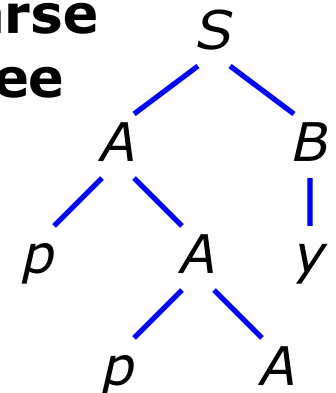
A y

p A y

pp A y

ppy

Parse Tree



Context-Free Languages

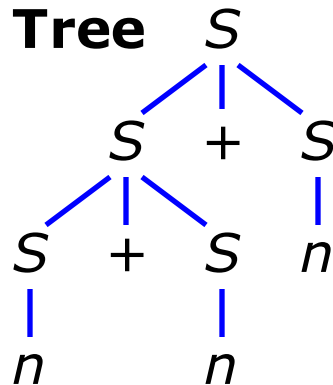
Parse Trees — Definition [3/3]

Another grammar, derivation, and associated parse tree.

Grammar B

$S \rightarrow S+S \mid n$

Parse Tree



Derivation of $n+n+n$

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

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 above derivation, draw a parse tree for the string *ac*.
2. Based on Grammar C, draw a parse tree for the string *bccc*.

Context-Free Languages

Parse Trees — TRY IT [2/3]

Grammar C

$S \rightarrow XY$

$X \rightarrow a \mid b$

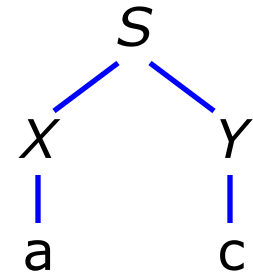
$Y \rightarrow cY \mid c$

Derivation of ac

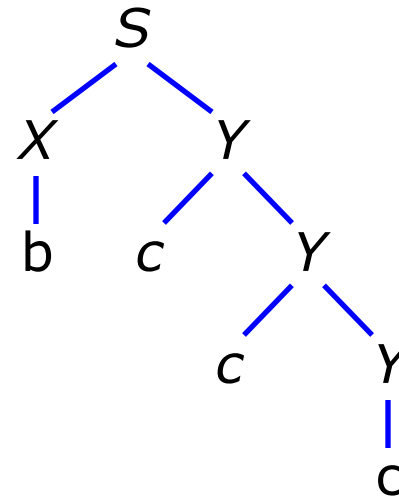
\underline{S}
 \underline{XY}
 $a\underline{Y}$
 ac

Answers

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



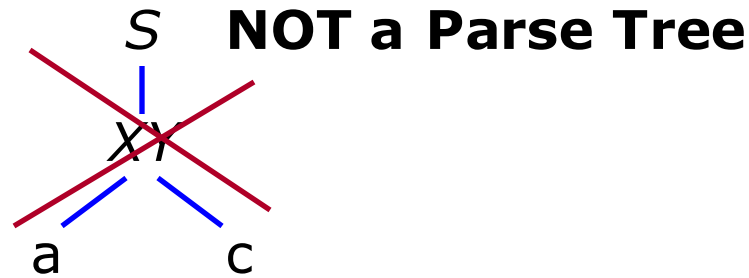
2. Based on Grammar C, draw a parse tree for the string $bccc$.



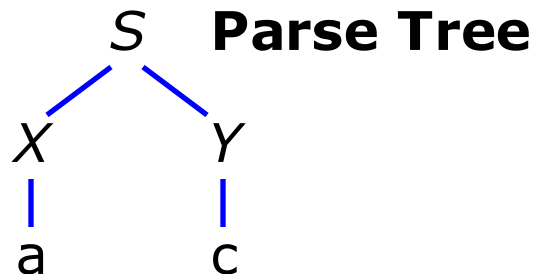
Context-Free Languages

Parse Trees — TRY IT [3/3]

If you drew something like this for Exercise 1 ...



... then you should be aware that the above is not a parse tree.
A parse tree has *one symbol in each node*.



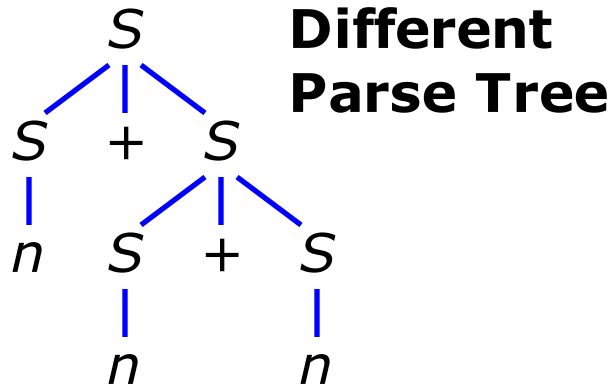
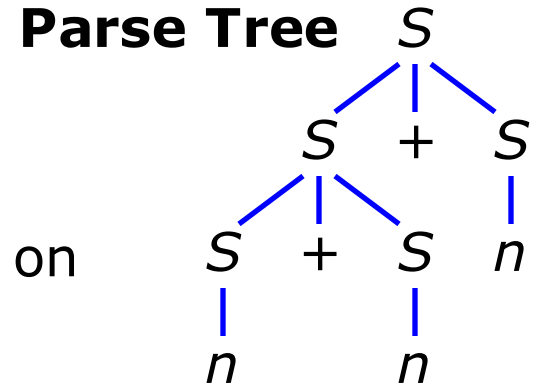
Context-Free Languages

Ambiguity — Definition

Grammar B

$S \rightarrow S+S \mid n$

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



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.

Context-Free Languages

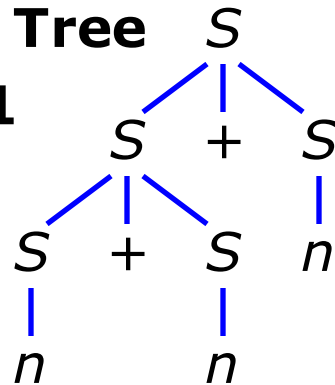
Ambiguity — Eliminating Ambiguity [1/3]

Grammar B

$S \rightarrow S+S \mid n$

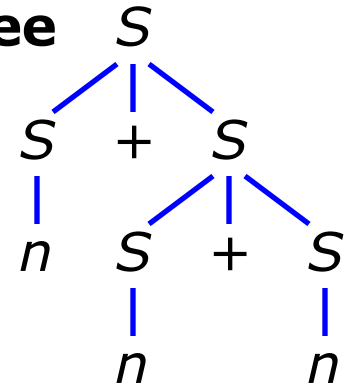
Parse Tree

#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:
 $n+n+n = (n+n)+n$.

Context-Free Languages

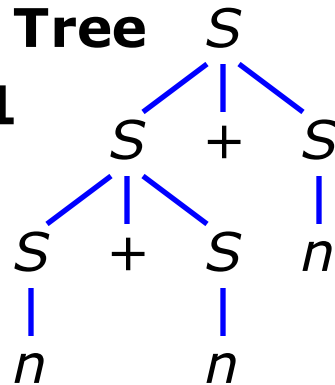
Ambiguity — Eliminating Ambiguity [2/3]

Grammar B

$S \rightarrow S+S \mid n$

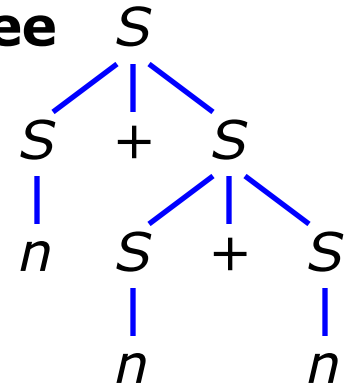
Parse Tree

#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'

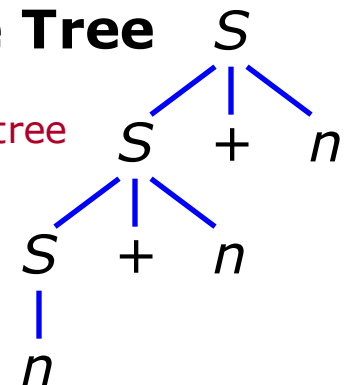
$S \rightarrow S+n \mid n$

Derivation

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

Parse Tree

This parse tree is unique!



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

Here is a standard example of an inherently ambiguous CFL.

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

The diagram shows two sets of strings. The first set is $\{ a^m b^m c^n d^n \mid m, n \geq 0 \}$. Red arrows point from the word 'Same' above to the 'a' and 'b' exponents, and from the word 'Same' below to the 'c' and 'd' exponents. The second set is $\{ a^m b^n c^n d^m \mid m, n \geq 0 \}$. Red arrows point from the word 'Same' above to the 'a' and 'd' exponents, and from the word 'Same' below to the 'b' and 'c' exponents.

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

Notes

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

Context-Free Languages

Ambiguity — Ramifications

Ambiguity in a CFG is not a good thing. In the context of parsing, it can mean that a program potentially has more than one possible structure—and thus likely more than one possible semantics.

Curiously, it is known to be *impossible* to design an algorithm that can determine, for an arbitrary given CFG, whether or not it is ambiguous.

However, it is certainly possible to determine the ambiguity (or lack thereof) for many specific CFGs.

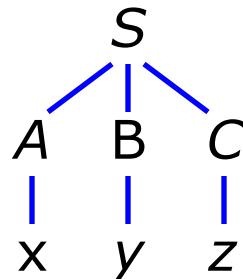
Context-Free Languages

Leftmost & Rightmost Derivations [1/3]

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

Grammar D	Derivation #1	Derivation #2	Derivation #3
$S \rightarrow ABC$	<u>S</u>	<u>S</u>	<u>S</u>
$A \rightarrow x$	<u>ABC</u>	<u>ABC</u>	<u>ABC</u>
$B \rightarrow y$	<u>xBC</u>	<u>ABz</u>	<u>AyC</u>
$C \rightarrow z$	<u>xyC</u>	<u>Ayz</u>	<u>Ayz</u>
	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, the three derivations of xyz differ significantly.

- 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

S
ABC
 x BC
 xy C
 xyz

#2: Rightmost derivation

S
 ABC
 AB z
 A yz
 xyz

#3: Neither

S
 AB C
 Ay C
 A yz
 xyz

Context-Free Languages

Leftmost & Rightmost Derivations [3/3]

#1: Leftmost derivation

S
ABC
xBC
xyC
xyz

#2: Rightmost derivation

S
ABC
ABz
Ayz
xyz

#3: Neither

S
ABC
AyC
Ayz
xyz

These concepts will come up in our study of parsing.

- A parser goes through the process of finding a derivation. Some parsers construct the derivation in forward order, with the leftmost nonterminal usually expanded first, producing a leftmost derivation.
- Other parsers construct a derivation in reverse, repeatedly **contracting** a substring to a nonterminal. Typically, the leftmost input is contracted first. Viewed in forward order, the rightmost nonterminal is expanded first, producing a rightmost derivation.

Context-Free Languages

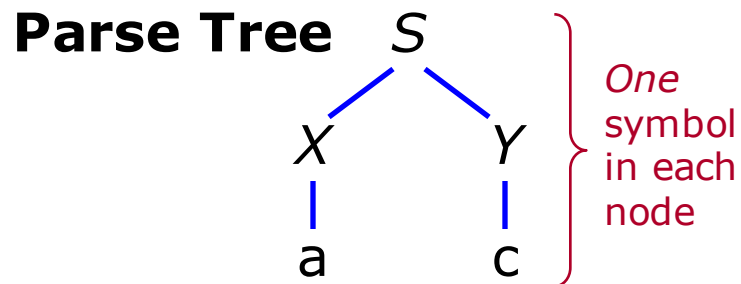
Notes

Do not confuse parse trees with derivations!

- A parse tree is a rooted tree.
- A derivation is a list of sequences of symbols.

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

Ambiguity is about multiple parse trees, not multiple derivations.



Leftmost Derivation

S
XY
aY
ac

Rightmost Derivation

S
XY
Xc
ac