San José State University
Department of Computer Science

Ahmad Yazdankhah

ahmad.yazdankhah@sjsu.edu
www.cs.sjsu.edu/~yazdankhah

Grammars

(Part 4)

Lecture 23

Day 25/31

CS 154

Formal Languages and Computability

Fall 2019

Agenda of Day 25

- Solution and Feedback of Quiz ++ and Quiz 8
- Summary of Lecture 22
- Quiz 9
- Lecture 23: Teaching ...
  – Grammars (Part 4)

Solution and Feedback of Quiz 8 (Out of 15)

Section	Average	High Score	Low Score
01 (TR 3:00 PM)	11.24	15	8
02 (TR 4:30 PM)	11.52	14	6
03 (TR 6:00 PM)	11.05	14	7

Solution and Feedback of Quiz ++ (Out of 15)

Section	Average	High Score	Low Score
01 (TR 3:00 PM)	54.16	60	38
02 (TR 4:30 PM)	56.4	60	49
03 (TR 6:00 PM)	56.06	60	42

Summary of Lecture 22: We learned ...

Context-Free Grammars (CFG)

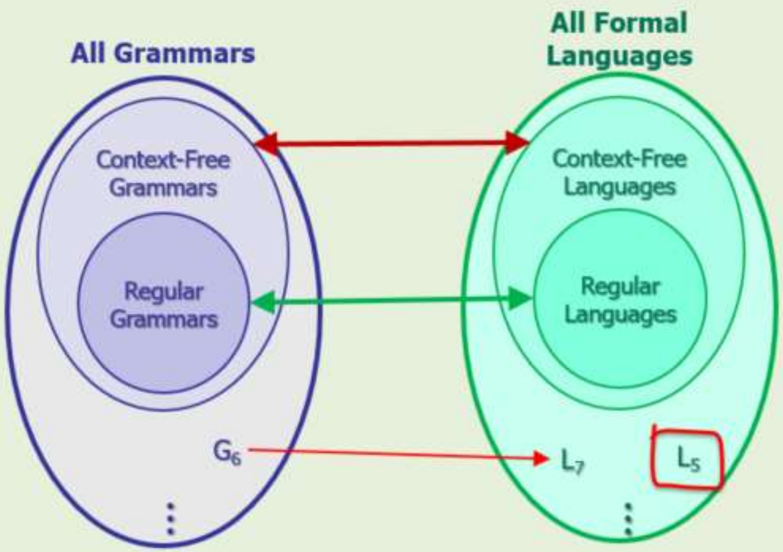
- A context-free grammar is ...

  ... a grammar whose production rules are of the form:

  $A \rightarrow v$

  Where $A \in V$ and $v \in (V \cup T)^*$

- A context-free language (CFL) is ...
  - ... a language produced by a CFG.



Any Question

Summary of Lecture 22: We learned ...

Context-Sensitive Grammars (CSG)

- A grammar G is context-sensitive if all production rules are of the form:

  $xAy \rightarrow xvy$

  Where $A \in V$ and $x, y, v \in (V \cup T)^*$ and $v \neq \lambda$

Unrestricted Grammars

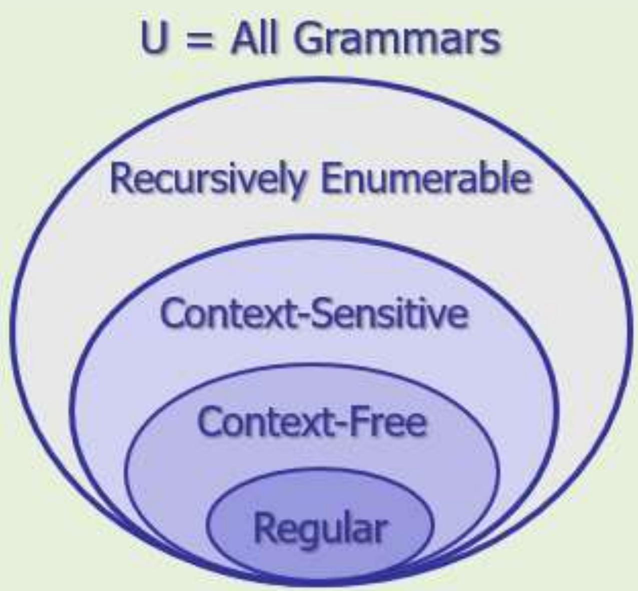
- A grammar G is recursively enumerable (aka unrestricted) if all production rules are of the form:

  $xAy \rightarrow z$

  where $A \in V$, $x, y, z \in (V \cup T)^*$

Chomsky's Hierarchy

- Type 0: Recursively-enumerable
- Type 1: Context-sensitive
- Type 2: Context-free
- Type 3: Regular



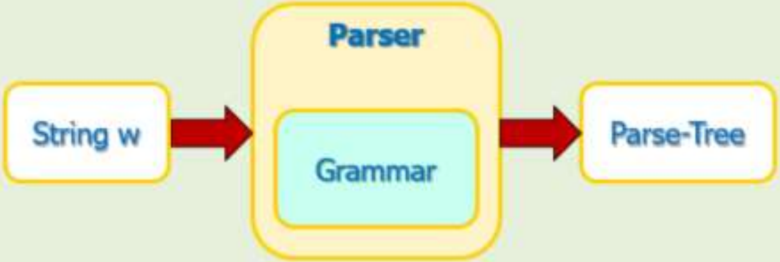
Summary of Lecture 22: We learned ...

Derivation Techniques

- There are two derivation techniques:
  - Leftmost and rightmost derivation.
  - Leftmost is the default method.

Parser

- Parser is ...
  - ... a program that gets a string as input and gives the sequence of derivation as the output.
  - We can construct parse-tree from that sequence.



- Every compiler has its own grammar and parser.

Any Question

NAME	Alan M. Turing		
SUBJECT	CS 154	TEST NO.	9
DATE	11/14/2019	PERIOD	1 / 2 / 3

TEST RECORD	
PART 1	123
PART 2	
TOTAL	

Your list # goes here!

Quiz 9

Use Scantron

Parse Trees

Parse Trees

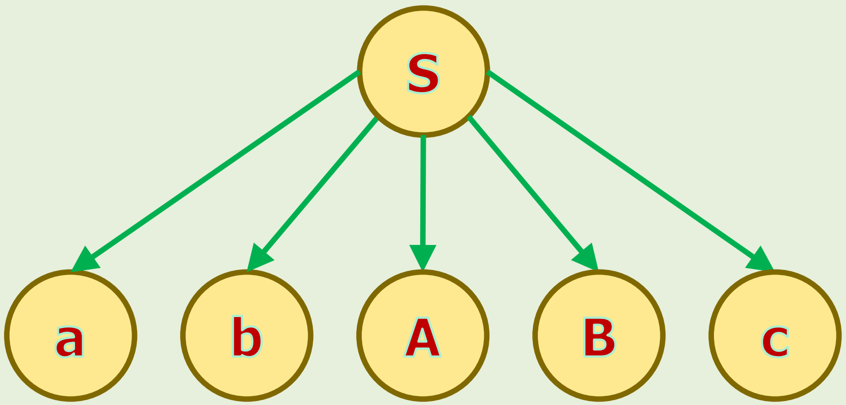
- Let's explain it through some examples.
- The first example shows how to construct a parse-tree for only one production-rule.

Example 25

- Construct a parse-tree for the following production rule.

  $S \rightarrow abABc$

- Note that the order of children matters.



Parse Trees

Example 26

- Given the following grammar:
  1. $S \rightarrow AB$
  2. $A \rightarrow aaA \mid \lambda$
  3. $B \rightarrow Bb \mid \lambda$
- Construct a parse-tree for the string aab.

Solution

- Note that every string has its own parse-tree.

Homework

- Given the following grammar:
  1. $S \rightarrow aAB$
  2. $A \rightarrow bBb$
  3. $B \rightarrow A \mid \lambda$

- Construct a parse-tree for the following strings:
  a. w = abbb
  b. w = abbbb
  c. w = abbbbb

Parsing Algorithms

Parsing Algorithms

- There are two main types of algorithms for parsers:
  1. Top-down
  2. Bottom-up

- To see the idea, we'll examine a top-down algorithm called "exhaustive search parsing" (aka "brute force parsing").
  - This algorithm checks all possibilities to derive a sting.

- We'll explain it through an example.

- For more information about other algorithms, you need to take Compiler Course!

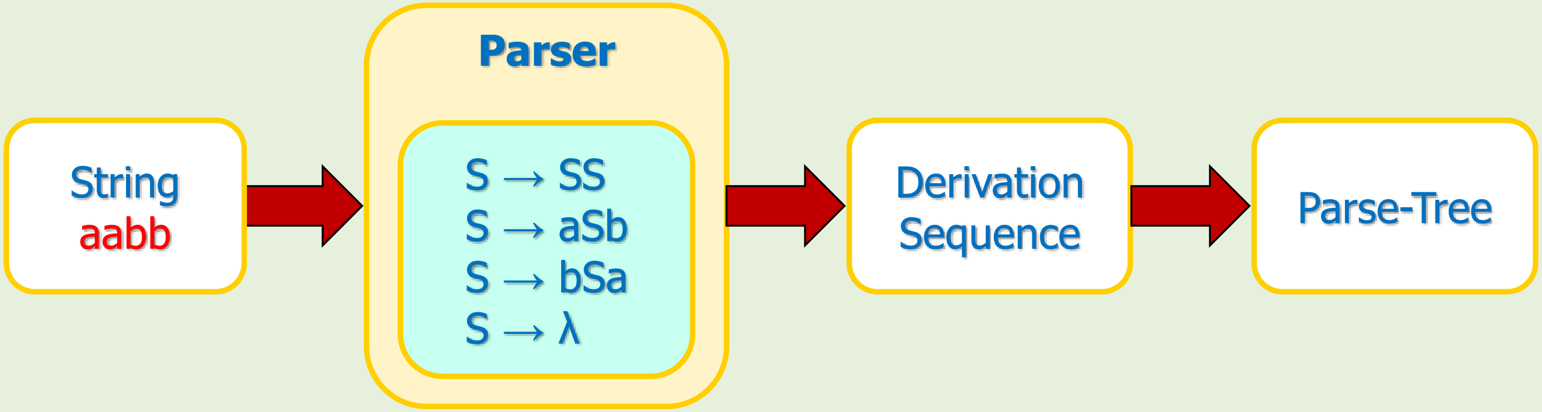
Exhaustive Search Parsing Algorithm: Example

Example 27

- Given the following grammar:

  $S \rightarrow SS \mid a\,S\,b \mid b\,S\,a \mid \lambda$

- Find a derivation sequence for w = aabb.
- Note that if we get the derivation sequence, then drawing the parse-tree would be simple.



Exhaustive Search Parsing Algorithm: Example

Example 27 (cont'd)

$S \to SS \mid aSb \mid bSa \mid \lambda$
$w = aabb$

- **Round One**
  1. $S \Rightarrow SS$
  2. $S \Rightarrow aSb$
  3. $S \Rightarrow bSa$
  4. $S \Rightarrow \lambda$
- Which production rules can be pruned?
- Number 3 and 4 can be pruned because they will never yield to w.

- **Conclusion of Round One**
  1. $S \Rightarrow SS$
  2. $S \Rightarrow aSb$
  3. ~~$S \Rightarrow bSa$~~
  4. ~~$S \Rightarrow \lambda$~~
- Therefore, 1 and 2 are our starters after the first round.

Exhaustive Search Parsing Algorithm: Example

Example 27 (cont'd)

$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
$w = aabb$

- **Conclusion of Round One**

Repeated

1. $S \Rightarrow SS$
2. $S \Rightarrow aSb$
3. ~~$S \Rightarrow bSa$~~
4. ~~$S \Rightarrow \lambda$~~

- In round 2, we substitute all possibilities for leftmost S in #1 and #2.

- **Round Two**
- Substitute leftmost S of #1 with all possible options:

1.1. $S \Rightarrow SS \Rightarrow SS\ S$

1.2. $S \Rightarrow SS \Rightarrow aSb\ S$

1.3. $S \Rightarrow SS \Rightarrow bSa\ S$

1.4. $S \Rightarrow SS \Rightarrow \lambda\ S$

- Substitute leftmost S of #2 with all possible options:

2.1. $S \Rightarrow a\ S\ b \Rightarrow a\ SS\ b$

2.2. $S \Rightarrow a\ S\ b \Rightarrow a\ aSb\ b$

2.3. $S \Rightarrow a\ S\ b \Rightarrow a\ bSa\ b$

2.4. $S \Rightarrow a\ S\ b \Rightarrow a\ \lambda\ b$

Exhaustive Search Parsing Algorithm: Example

Example 27 (cont'd)

$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
$w = aabb$

- **Conclusion of Round Two**

Repeated

1.1. $S \Rightarrow SS \Rightarrow SSS$

1.2. $S \Rightarrow SS \Rightarrow aSbS$

~~1.3. $S \Rightarrow SS \Rightarrow bSaS$~~

1.4. $S \Rightarrow SS \Rightarrow S$

2.1. $S \Rightarrow aSb \Rightarrow aSSb$

2.2. $S \Rightarrow aSb \Rightarrow aaSbb$

~~2.3. $S \Rightarrow aSb \Rightarrow abSab$~~

~~2.4. $S \Rightarrow aSb \Rightarrow ab$~~

- We continue this process …
- **Round 3**
- … (after a little bit cheating!)
- Substitute leftmost S of #2.2 with all possible options:

2.2.1. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\ SS\ bb$

2.2.2. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\ aSb\ bb$

2.2.3. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\ bSa\ bb$

2.2.4. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

- So, we got the derivation sequence to derive $w = aabb$

Exhaustive Search Parsing Algorithm: Complexity

- Exhaustive parsing has two serious problems:
  1. It is extremely inefficient: $O(|P|^{2|w|+1})$
     - Where $|P|$ is the number of production rules, and $|w|$ is the size of the string.
  2. It is possible that it never terminates if we don't put the appropriate controls in our program.
     - For example, try to find the derivation sequence for $w = abb$ in the previous example.

- How horrible do you think this efficiency is?

- Later, we'll take a practical example under the "Complexity" topic.

Parsing Algorithm: Good News

1. **Theorem**

   For every CFG G, there exists an algorithm that parses any $w \in L(G)$ in $O(|w|^3)$ steps.

2. **Using S-Grammar**

   If the grammar is s-grammar, then the efficiency of parsing would be: $O(|w|)$

- First, let's see what s-grammar is, then we'll take some examples.

Simple Grammars (S-Grammars)

Definition

- A context-free grammar G is said to be simple grammar (aka s-grammar) if the following two conditions are satisfied:

Condition #1

All production rules are of the form:

$A \rightarrow av$ Where $A \in V$, $a \in T$, $v \in V^*$

- Means: One terminal as prefix and any number of variables as suffix.

Condition #2

Any pair (A, a) occurs only once in all production rules.

S-Grammars Examples

Example 28

- Is the following grammar s-grammar?
  $S \to aS \mid bSS \mid c$

Solution

- Did you notice that λ is not part of S-grammar?

S-Grammars Examples

Example 29

- Is the following grammar s-grammar?
  $S \rightarrow bSS \mid aS \mid c \mid aSS$

Solution

Exhaustive Search Parsing Algorithm: S-Grammar

Example 30

- Given the following grammar:
  1. $S \to aS$
  2. $S \to bSS$
  3. $S \to c$
- Is this an s-grammar?
- Derive w = abcc

- Yes, both conditions of s-grammars are satisfied.
- Derivation of abcc:

$$S \overset{1}{\Rightarrow} aS \overset{2}{\Rightarrow} abSS \overset{3}{\Rightarrow} abcS \overset{3}{\Rightarrow} abcc$$

- Note that we are still using "exhaustive search parsing".
- The point is that each string has a unique derivation.
- That's why s-grammar is extensively used in the programming languages.

Exhaustive Search Parsing Algorithm: S-Grammar

FYI

Theorem

- If G is an s-grammar, then any string $w \in L(G)$ can be parsed in $O(|w|)$.

Proof

- Let's assume $w = a_1 a_2 \dots a_n$
- There can be at most one rule with S on the left and starting with a_1 on the right: $S \Rightarrow a_1 A_1 A_2 \dots A_m$
- Again, there can be at most one rule with A_1 on the left and starting a_2 on the right: $A_1 \Rightarrow a_2 B_1 B_2 \dots B_k$
- So, $S \Rightarrow a_1 a_2 B_1 B_2 \dots B_k A_2 \dots A_m$
- It means that after $|w|$ we can derive w.

Ambiguity in Grammars

Introduction

- We learned that parsers produce a parse-tree for every $w \in L(G)$.

- But the point is that the parse-tree is NOT always UNIQUE.
  - In other words, in some cases, for some $w \in L(G)$, there are more than one parse-tree.

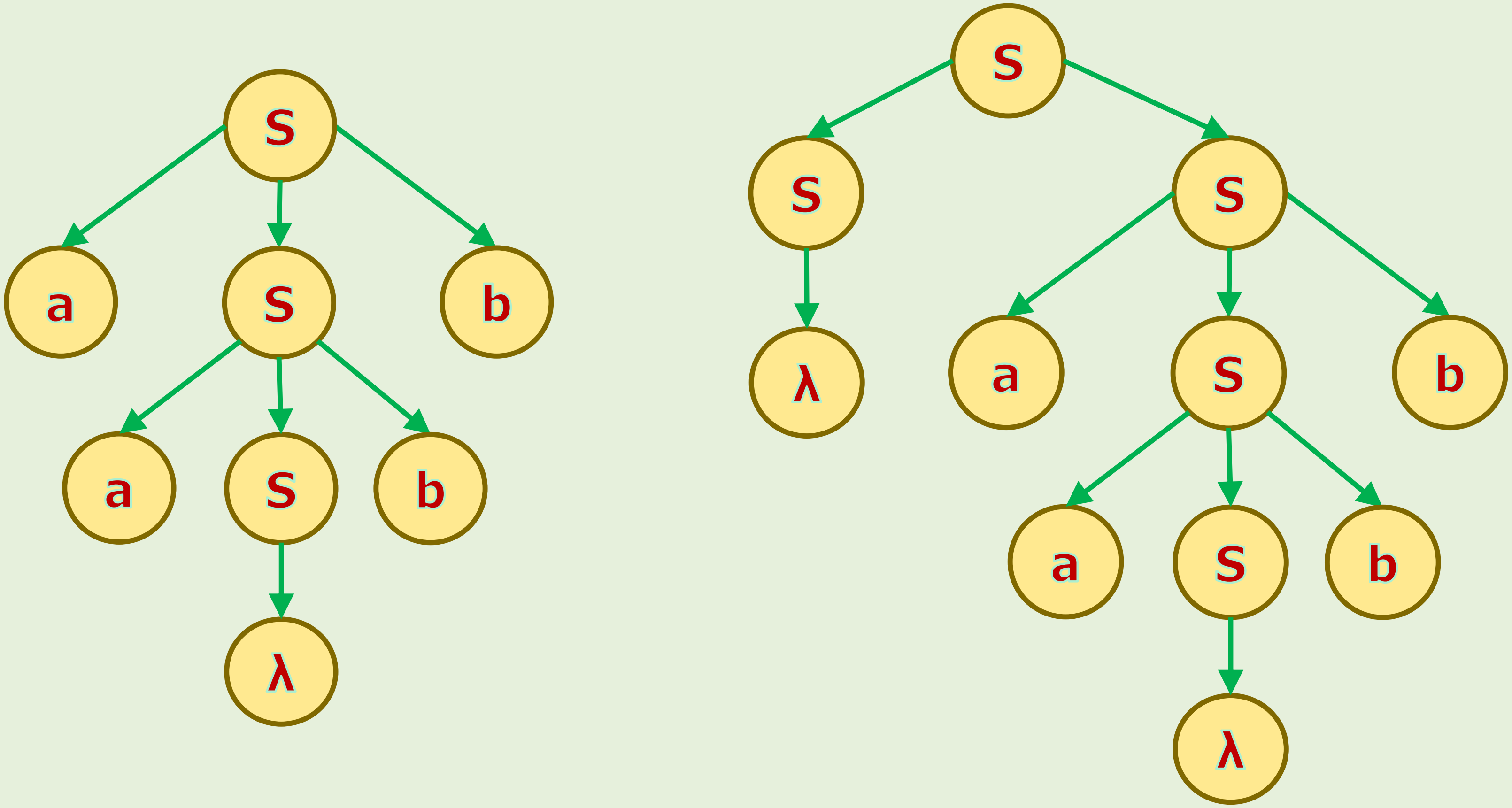
- First, let's see this through an example!
- Then, we show what could be the consequence of this non-uniqueness in practice!

When Parse-Tree is NOT Unique

Example 31

Given grammar G as: $S \rightarrow aSb \mid SS \mid \lambda$

Draw possible parse-trees for driving w = aabb.

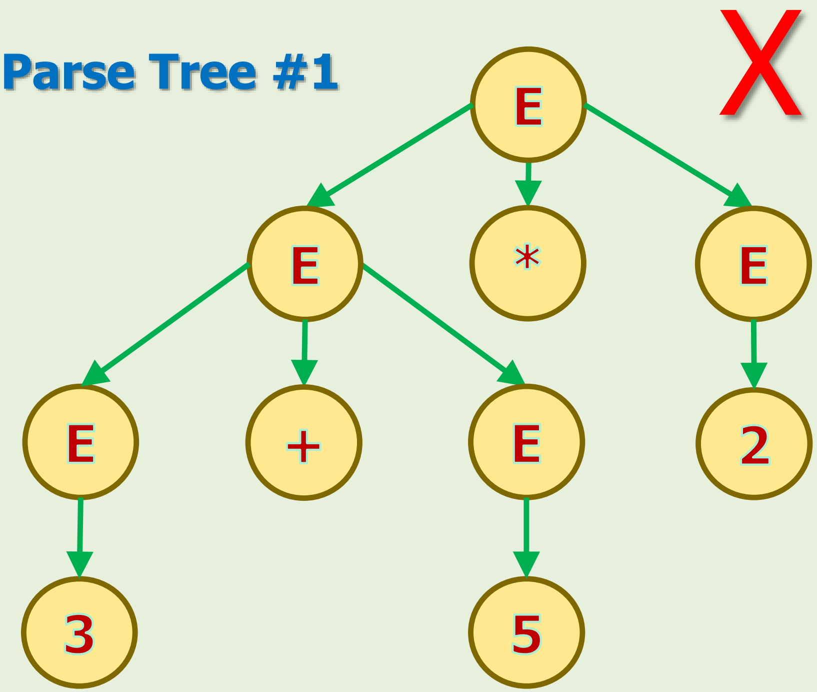


Non-Uniqueness of Parse-Trees Problem in Practice

Example 32

- Given grammar G as:
  1. $E \rightarrow E * E$
  2. $E \rightarrow E + E$
  3. $E \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- E is starting variable.
- Construct a parse-tree for the mathematical expression: 3 + 5 * 2
  - Note that this expression is just a string.
- This grammar is a simplified version of arithmetic expressions in the programming languages.

Parse Tree #1



- Is this a good parse-tree?
- No, because '*' should have more priority than + but this parse-tree is calculating (3 + 5) * 2.

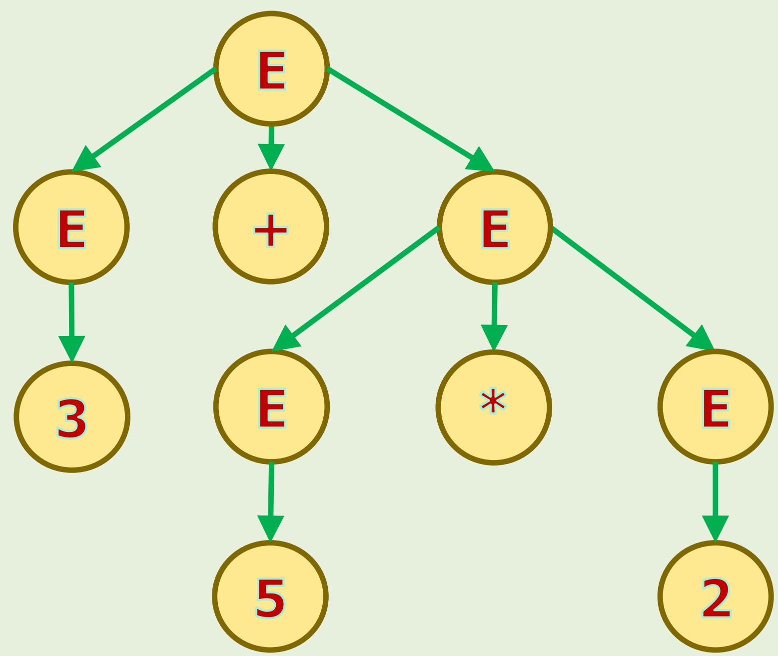
Non-Uniqueness of Parse Trees Problem in Practice

Example 32 (cont'd)

Repeated

- Given grammar G as:
  1. $E \rightarrow E * E$
  2. $E \rightarrow E + E$
  3. $E \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- E is starting variable.
- Construct a parse-tree for the mathematical expression: 3 + 5 * 2

Parse Tree #2



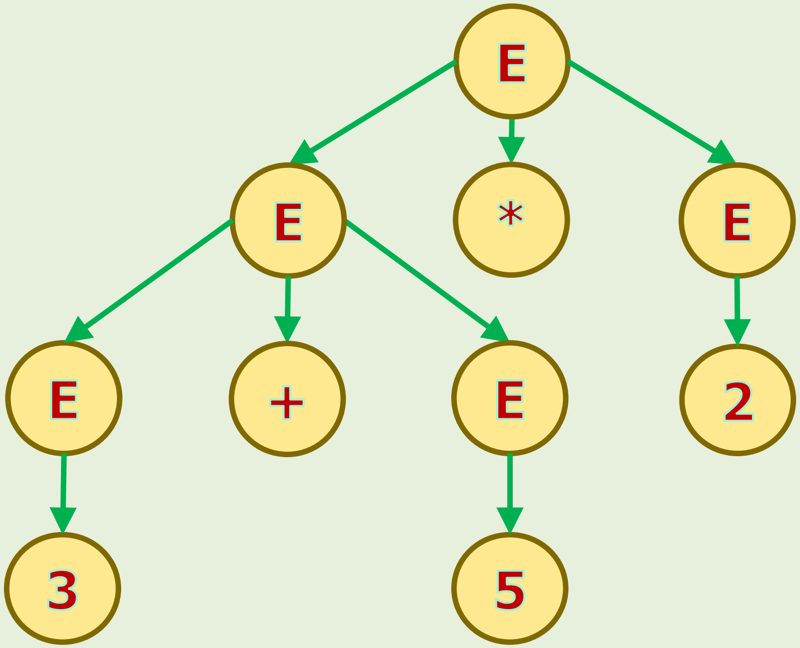
- Is this a good parse-tree?
- Yes! It's calculating 3 + (5 * 2)

Non-Uniqueness of Parse Tree Problem in Practice

Example 32 (cont'd)

Parse Tree #1

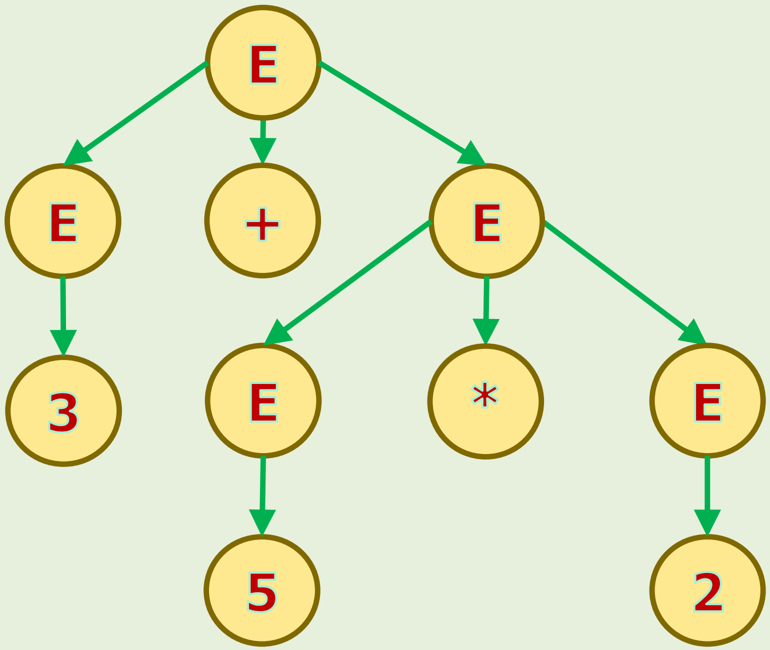
X



- Bad Parse Tree

Parse Tree #2

✓



- Good Parse Tree

Ambiguity in Grammars

Definition

- A grammar G is said to be ambiguous if there exists some $w \in L(G)$ that has at least two different parse-trees.

- In some cases, we can convert an ambiguous grammar to non-ambiguous one.
- But most of times, it is hard and needs compiler knowledge.
- You might learn these skills in "Compiler Course".

- Let's rewrite the grammar of our previous example and remove the ambiguity.

Ambiguity in Grammars

Example 33

- Convert the following grammar to an unambiguous grammar.

  1. $E \rightarrow E * E$
  2. $E \rightarrow E + E$
  3. $E \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

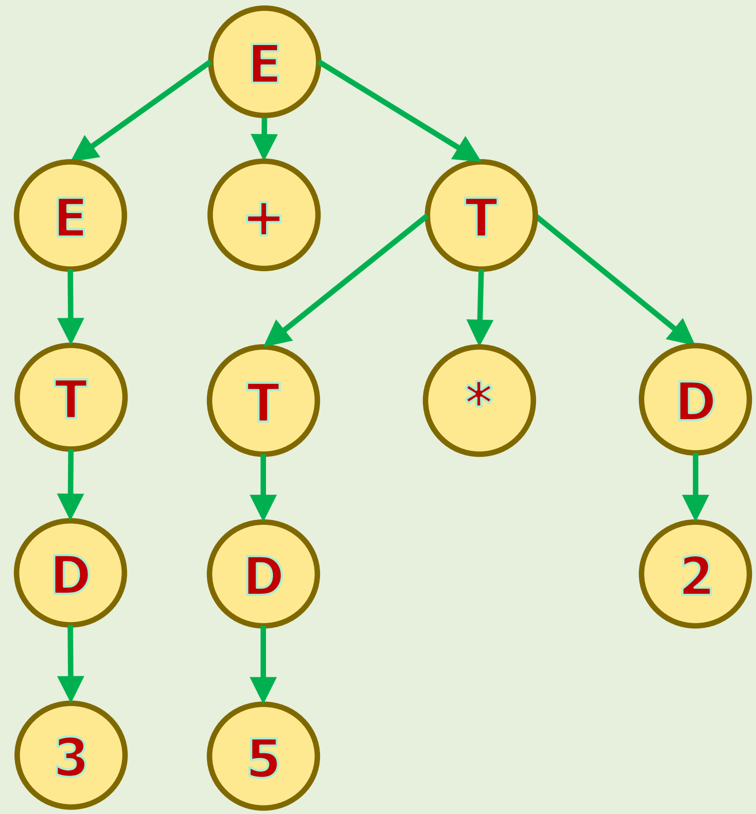
- E is starting variable.

Solution

  1. $E \rightarrow E + T \mid T$
  2. $T \rightarrow T * D \mid D$
  3. $D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Construct a parse-tree for:
  3 + 5 * 2

Parse Tree

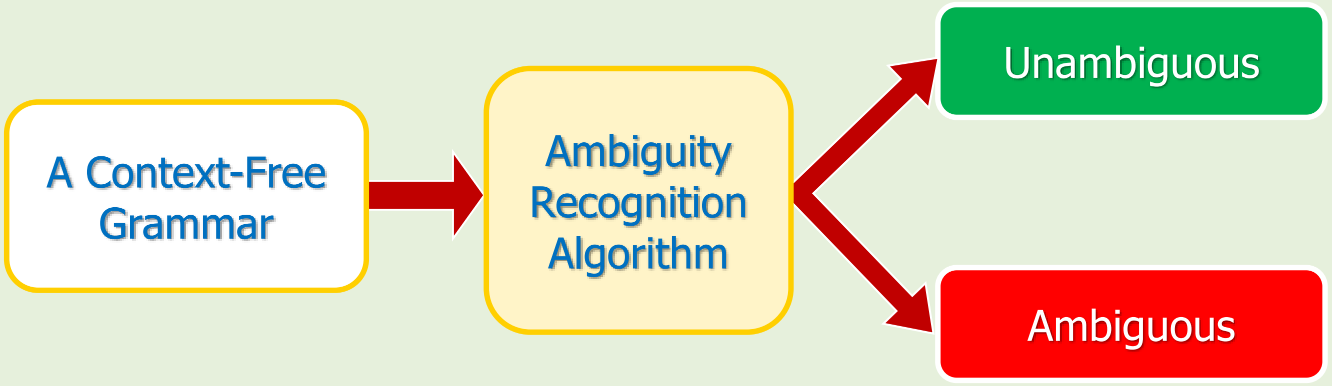


- There is no other parse-tree for this string.

Two Open Questions

1. Given a context-free grammar G.

- Is there an efficient algorithm to find out whether G is ambiguous or not?

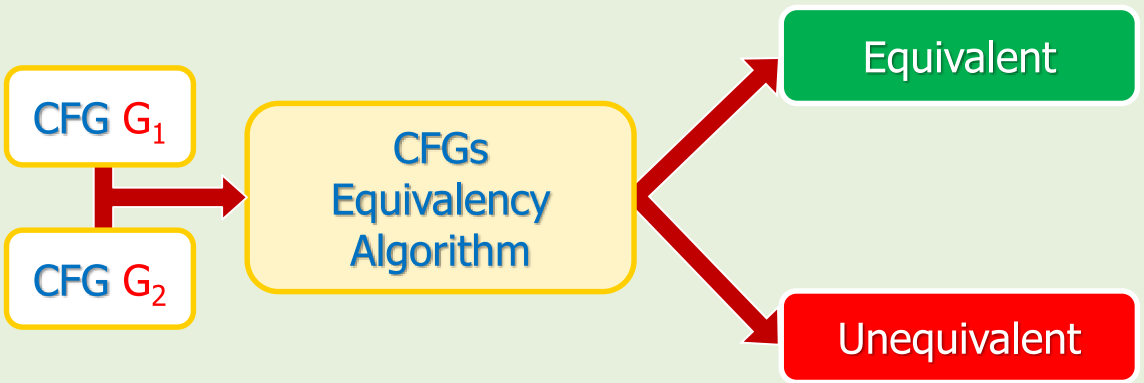


- As of this moment, there is no general algorithm to answer this question.

Two Open Questions

2. Are two given context-free grammars G_1 and G_2 equivalent?

- Is there an efficient algorithm to answer this question?

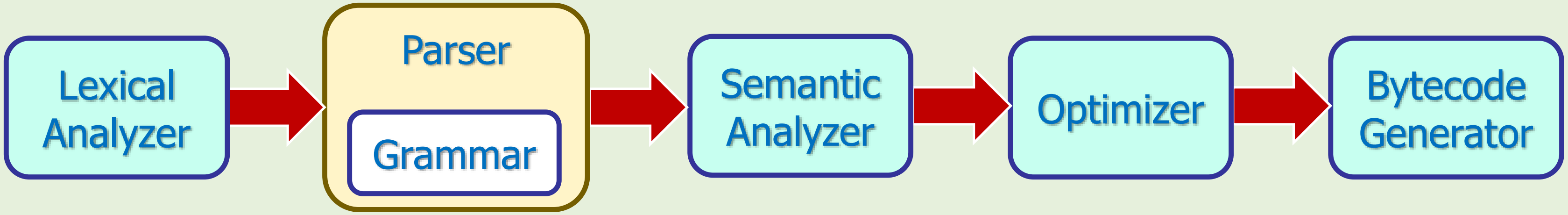


- Again, as of this moment, there is no general algorithm to answer this question.

Java Compiler (From Compiler Course!)

FYI

1. **Lexical Analyzer** (aka Lexer or scanner): breaks the entire code up into words (tokens)
2. **Parser**: by using the grammar, generates the parse-tree, checks the syntax of the sentences
3. **Semantic Analyzer**: checks the sentences meaning
4. **Optimizer**: optimizes the sentences to be more efficient
5. **Code Generator**: produces the bytecode



References

1. Linz, Peter, "An Introduction to Formal Languages and Automata, 5th ed.," Jones & Bartlett Learning, LLC, Canada, 2012
2. Michael Sipser, "Introduction to the Theory of Computation, 3rd ed.," CENGAGE Learning, United States, 2013
   ISBN-13: 978-1133187790
3. The ELLCC Embedded Compiler Collection, available at: http://ellcc.org/