CS 3313
Foundations of Computing:

Introduction to Context Free Grammars

http://gw-cs3313-2021.github.io

Recall: Three key concepts

- Languages
  - Set of sentences (strings/words) formed from some alphabet
  - How do we specify the property?
- Grammars
  - A formalism for mathematically defining the properties of a language
  - A set of rules for generating the sentences in a formal language
- Automata
  - Mathematical model of machines (of different capabilities)
  - Formal construct that accepts input, produces output and may have temporary storage and can make decisions

A better formalism to define language ?
Some questions

- Set notation works but does not specify a way to generate the words/strings in the language
- Regular expressions work for regular languages but many interesting/useful languages are not regular
- Ex: how do you define a syntactically valid C program ?
- Ex: how do you define the construction of a sentence in the English language ?

Need for formalism and Math rigor

- We would like to capture precisely, and logically, the properties (problems) in the language
- Ex.: actual and formal parameters (arguments) should match in a program
- Ex.: valid sentences in English ? How do we define the a language to be ambiguous…which is a bad thing in a programming language
  - How do we define, using a mathematical structure, what ambiguity means (in an unambiguous manner ☺)

Grammars: Definition

- Precise mechanism to describe the strings in a language
- Definition: A grammar G (V,T,P,S) consists of:

  V: a finite set of *variable* or non-terminal symbols

  T: a finite set of *terminal* symbols (the *alphabet*!)

  S: a variable called the *start* symbol

  P: a set of *productions* (also called production rules)
- Example 1:

  $V = \{ S \}$

  $T = \{ a, b \}$

  $P = \{ S \rightarrow aSb, S \rightarrow \lambda \}$

<sentence> = <noun phrase><verb phrase>

Grammars - comments

- Basic idea is to use “variables” to stand for sets of strings (i.e., languages)
  - Variable for <nouns>, <verbs>
- These variables are defined recursively, in terms of one another
- Recursive rules (Productions) involve only concatenation
  - Alternate rules for a variable allow union
- Production rule is of the form $x \rightarrow y$ where x,y are $(V \cup T)^+$
  - Production rules are akin to the transition function of an automaton

Grammars: Derivation of Strings

- Beginning with the start symbol, strings are derived by repeatedly replacing string on left hand side symbols with the expression on the right-hand side of any applicable production
- Any applicable production can be used, in arbitrary order, until the string contains no variable symbols.
- Sample derivation using grammar in Example:

$S \Rightarrow aSb$ (applying first production)

$\Rightarrow aaSbb$ (applying first production)

$\Rightarrow aabb$ (applying second production)

The Language Generated by a Grammar

- Definition: For a given grammar G, the language generated by G, L(G), is the set of all terminal strings derived from the start symbol
- If G is a CFG, then L(G), the *language of G*, is

  L(G)= *{w | S =>* w and w is a string over set T}*.
- Example: G has productions S -> ϵ and S -> 0S1.
- L(G) = $\{0^n 1^n \mid n \geq 0\}$.

- To show a language L is generated by G:
  - Show every string in L can be generated by G
  - Show every string generated by L is in G

Grammars and Language Classes

- By placing constraints on what type of productions are allowed, we define different language classes.

Regular Grammars

- In a right-linear grammar, at most one variable symbol appears on the right side of any production. If it occurs, it is the rightmost symbol.
- In a left-linear grammar, at most one variable symbol appears on the right side of any production. If it occurs, it is the leftmost symbol.
- A *regular grammar* is either right-linear or left-linear.
- Example: a regular (right-linear) grammar:

V = { S }, T = { a, b }, and productions S $\rightarrow$ abS | a

Context Free Grammars

- A context free grammar is a grammar G=(V,T,P,S) where all production rules are of the form: $V \rightarrow (V \cup T)^*$
  - Production rules have exactly one variable on the left and a string consisting of variables and terminals on the right.

$V = \{ S \}$

$T = \{ a, b \}$

$P = \{ S \rightarrow aSb, S \rightarrow \lambda \}$

Grammars for Programming Languages

- The syntax of constructs in a programming language is commonly described with grammars
  - Commonly referred to as Backus-Naur Form (BNF)
- Assume that in a hypothetical programming language,
  - Identifiers consist of digits and the letters a, b, or c
  - Identifiers must begin with a letter
- Productions for a sample grammar:

<id> $\rightarrow$ <letter> <rest>

<rest> $\rightarrow$ <letter> <rest> | <digit> <rest> | λ

<letter> $\rightarrow$ a | b | c

<digit> $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Right-Linear Grammars Generate Regular Languages

Theorem: It is always possible to construct a nfa to accept the language generated by a regular grammar G:

- Label the nfa start state with S and a final state V_f
- For every variable symbol V_i in G, create a nfa state and label it V_i
- For each production of the form $A \rightarrow aB$, label a transition from state A to B with symbol a
- For each production of the form $A \rightarrow a$, label a transition from state A to V_f with symbol a (may have to add intermediate states for productions with more than one terminal on RHS)

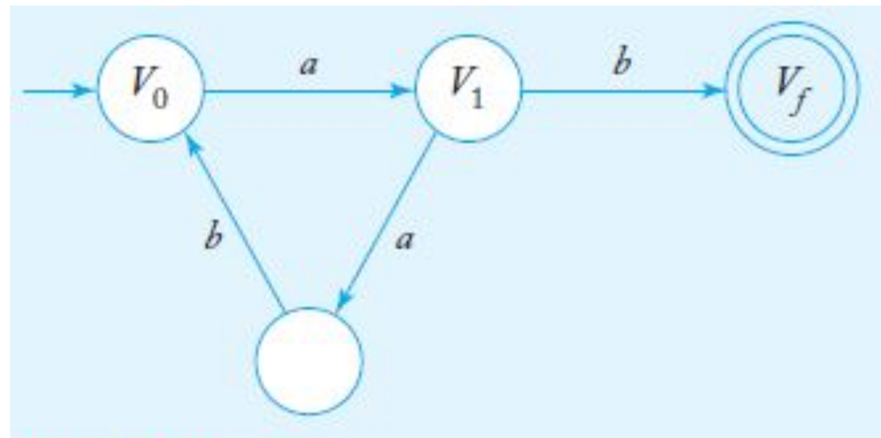
Example: Construction of a nfa to accept a language L(G)

Given the regular grammar G with productions

$V_0 \rightarrow aV_1$

$V_1 \rightarrow abV_0 \mid b$

a nondeterministic fa to accept L(G) can be constructed systematically



Grammars and Languages

- Regular Grammars = Regular Languages

- We have a context free grammar for L= $\{a^n b^n\}$ but we know that L is not regular:

  Context Free Grammars <> Regular Languages

  - DFAs cannot accept context free languages
  - Need a new model of automaton
    - Add simplest form of memory

Context Free Grammars

Context Free Grammars

- A context free grammar is a grammar $G=(V,T,P,S)$ where all production rules are of the form: $V \rightarrow (V \cup T)^*$
  - Production rules have exactly one variable on the left and a string consisting of variables and terminals on the right.

$V = \{ S \}$

$T = \{ a, b \}$

$P = \{ S \rightarrow aSb, S \rightarrow \lambda \}$

Grammars - comments

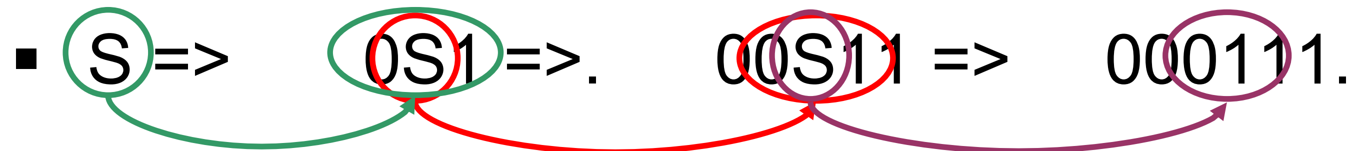
- Basic idea is to use “variables” to stand for sets of strings (i.e., languages)
  - Variable for <nouns>, <verbs>
- These variables are defined recursively, in terms of one another
- Recursive rules (Productions) involve only concatenation
  - Alternate rules for a variable allow union
- Production rule is of the form $x \rightarrow y$ where x is in V and y is in $(V \cup T)^*$
  - Production rules are akin to the transition function of an automaton

Recall: Derivations

- Beginning with the start symbol, strings are derived by repeatedly replacing a variable in string on left hand side with the expression on the right-hand side of any applicable production
- We say $\alpha A \beta$ => $\alpha\gamma\beta$ if A -> γ is a production.

Example: S -> 01; S -> 0S1.

- S => 0S1 =>. 00S11 => 000111.



- we are replacing the occurrence of variable A in string $\alpha A \beta$ on LHS with the RHS of the production A -> γ
- =>* means "zero or more derivation steps."

Recall: The Language Generated by a Grammar

- Definition: For a given grammar G, <u>the language generated by G</u>, L(G), is the set of all terminal strings derived from the start symbol
- If G is a CFG, then L(G), the *language of G*, is

  L(G) = *{w | S =>* w and w is a string over set T}*.
- Example: G has productions S -> ϵ and S -> 0S1.
  - $L(G) = \{0^n 1^n \mid n \geq 0\}$.
- To show a language L is generated by G:
  - Show every string in L can be generated by G
  - Show every string generated by L is in G
- Definition: a string α is in sentential form if S =>* α
  - The string α can derive a sentence in the language

Leftmost and Rightmost Derivations

- In a *leftmost derivation*, the leftmost variable in a sentential form is replaced at each step
- In a *rightmost derivation*, the rightmost variable in a sentential form is replaced at each step
- Consider the grammar :

  $V = \{ S, A, B \}$, $T = \{ a, b \}$, and productions

  $S \rightarrow aAB$

  $A \rightarrow bBb$

  $B \rightarrow A \mid \lambda$
- The string abb has two distinct derivations:
  - Leftmost: $S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abbB \Rightarrow abb$
  - Rightmost: $S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abb$

Derivations as Parse Trees

- Represent derivation of a string as (directed) Tree

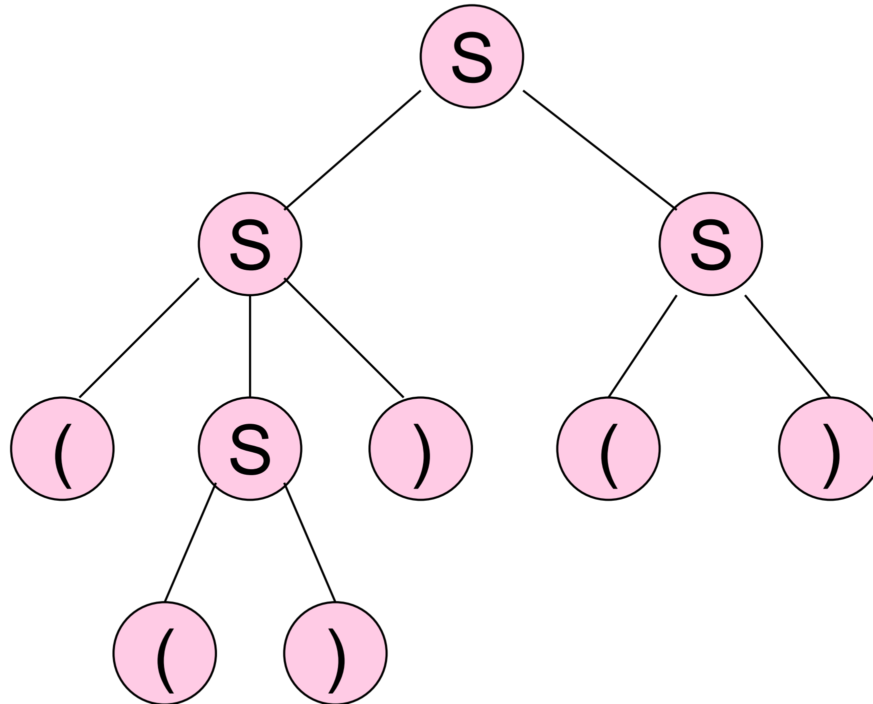
- Why is this useful ?

Parse Trees, also known as Derivation Trees

- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ϵ.
- Interior nodes: labeled by a variable (occurring on LHS of production).
  - Children are labeled by the RHS body of a production for the parent.
- Root: must be labeled by the start symbol.

Example: Parse Tree

S -> SS | (S) | ()

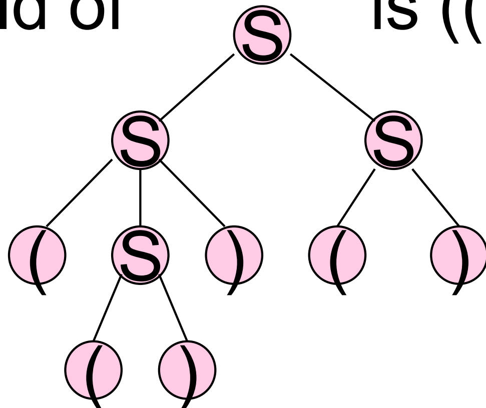


Yield of a Parse Tree

- The concatenation of the labels of the leaves in left-to-right order
  - That is, *in the order of a preorder traversal*.

  is called the *yield* of the parse tree.

- Example: yield of is (())()



Context-Free Grammar

- **Example**: $\{a^m b^n \mid m > n \geq 0\}$
- "Parallel" generation: generating two parallel parts; still from the "outside" to the "inside".
- Grammar ?

  $S \to AC$ S generates more a's than b's

  $C \to aCb \mid \lambda$ C generates equal number of a's and b's

  $A \to aA \mid a$ A generates at least one a

- Parse Tree for a^3b

Generalization of Parse Trees

- We sometimes talk about trees that are not exactly parse trees, but only because the root is labeled by some variable A that is not the start symbol.
- Call these *parse trees with root A*.

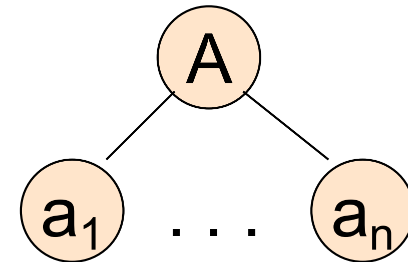
Equivalence of Parse Trees, Leftmost and Rightmost Derivations

- Trees, leftmost, and rightmost derivations correspond.
- We'll prove:
  1. If there is a parse tree with root labeled A and yield w, then

     $A =>^*_{lm} w$.
  1. If $A =>^*_{lm} w$, then there is a parse tree with root *A* and yield w.

Proof – Part 1

- Induction on the *height* (length of the longest path from the root) of the tree.
- Basis: height 1. Tree looks like
- $A \rightarrow a_1 \ldots a_n$ must be a production.
- Thus, $A \Rightarrow^*_{lm} a_1 \ldots a_n$.



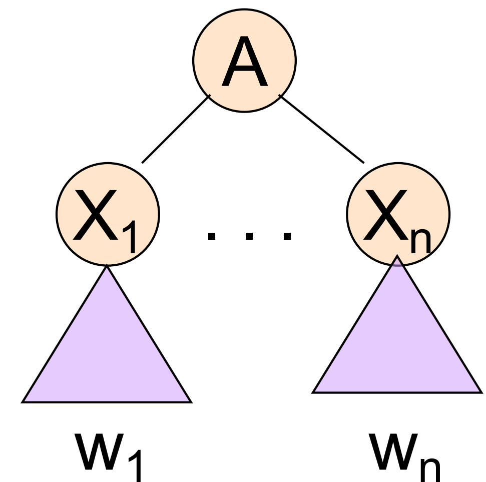
Part 1 – Induction

- Assume (1) for trees of height $< h$, and let this tree have height h:
- By IH, $X_i =>^*_{lm} w_i$.
  - Note: if X_i is a terminal, then $X_i = w_i$.
- Thus, $A =>_{lm} X_1...X_n$

$$=>^*_{lm} w_1X_2...X_n$$

$$=>^*_{lm} w_1w_2X_3...X_n$$

$$=>^*_{lm} ... =>^*_{lm} w_1...w_n.$$

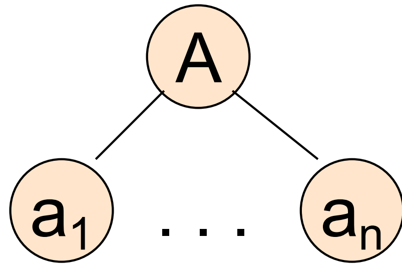


Proof: Part 2

- Given a leftmost derivation of a terminal string, we need to prove the existence of a parse tree.
- The proof is an induction on the length of the derivation.

Part 2 – Basis

- If $A =>^*_{lm} a_1...a_n$ by a one-step derivation, then there must be a parse tree

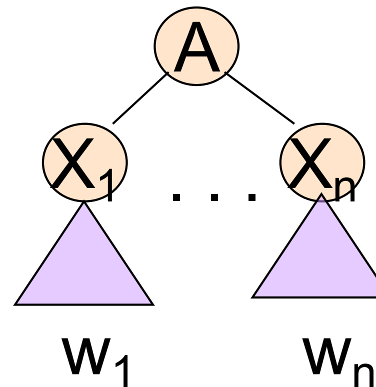


Part 2 – Induction

- Assume (2) for derivations of fewer than k > 1 steps, and let $A =>^*_{lm} w$ be a k-step derivation.
- First step is $A =>_{lm} X_1...X_n$.
- Key point: w can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.
  - If X_i is a terminal, then $w_i = X_i$.

Induction – (2)

- That is, $X_i =>^*_{lm} w_i$ for all i such that X_i is a variable.
  - And the derivation takes fewer than k steps.
- By the IH, if X_i is a variable, then there is a parse tree with root X_i and yield w_i.
- Thus, there is a parse tree



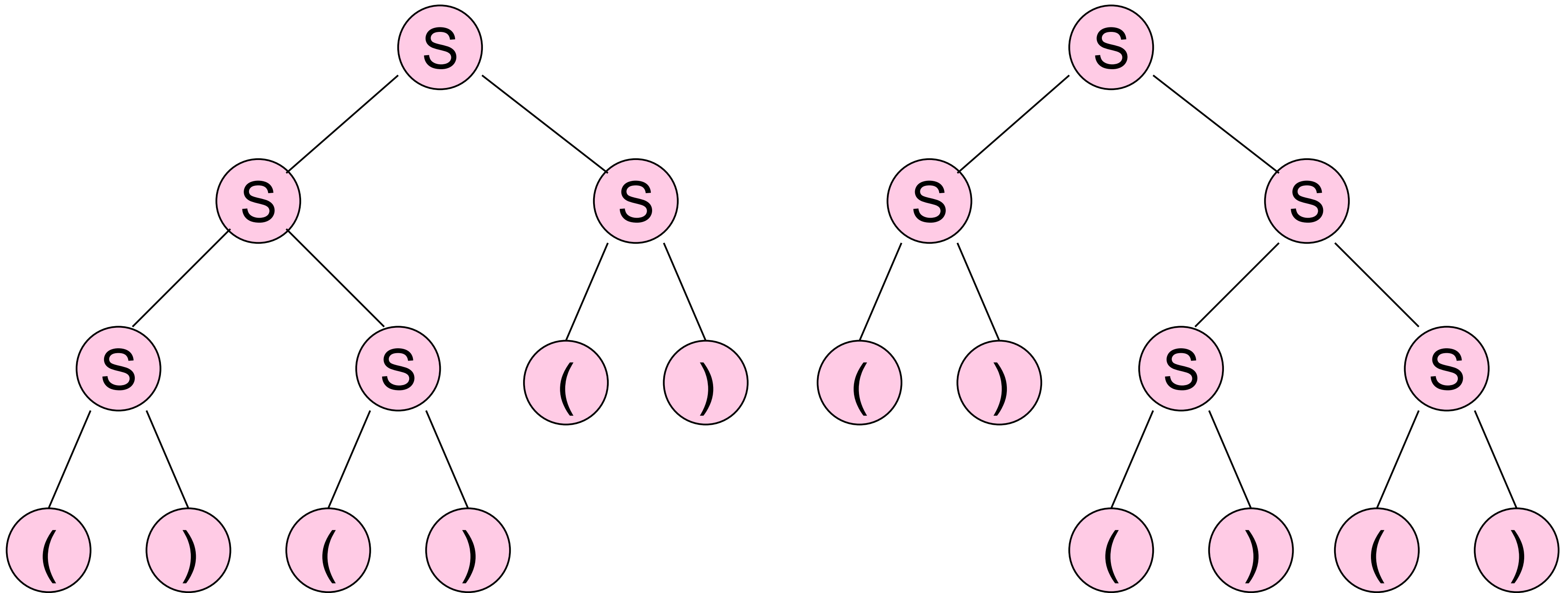
Parse Trees and Any Derivation

- The proof that you can obtain a parse tree from a leftmost derivation doesn't really depend on "leftmost."
- First step still has to be $A => X_1 ... X_n$.
- And w still can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.

Ambiguous Grammars: Definition

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.
- Example: S -> SS | (S) | $()$
- Two parse trees for ()()() on next slide.

Example – Continued



Ambiguity, Left- and Rightmost Derivations

- If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof.
- Conversely, two different leftmost derivations produce different parse trees by the other part of the proof.
- Likewise for rightmost derivations.

Ambiguity, etc. – (2)

- Thus, equivalent definitions of "ambiguous grammar'' are:
  1. There is a string in the language that has two different leftmost derivations.
  2. There is a string in the language that has two different rightmost derivations.

Ambiguity is a Property of Grammars, not Languages

- For the balanced-parentheses language, here is another CFG, which is unambiguous.

B -> (RB | ϵ

R ->) | (RR

B, the start symbol, derives balanced strings.

R generates certain strings that have one more right paren than left.

Example: Unambiguous Grammar

B -> (RB | ϵ   R ->) | (RR

- Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right.
  - If we need to expand B, then use B -> (RB if the next symbol is "("; use ϵ if at the end.
  - If we need to expand R, use R ->) if the next symbol is ")" and (RR if it is "(".

Inherent Ambiguity

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity, as we did for the balanced-parentheses grammar.
- Unfortunately, certain CFL's are *inherently ambiguous*, meaning that every grammar for the language is ambiguous.

Interesting Question on Ambiguity

- Given a CFG G, can we determine if the language is inherently ambiguous ?
  - Oops!

- Programming language syntax
  - Unambiguous grammars
  - Or semantic rules to resolve ambiguity
    - Ex: Precedence rules in expressions a+a*a ?

Next….the quest for ”automation”!

- We would like to answer the question “Does G derive string w” *i.e.,* Is w generated by the grammar?
  - Ex: Given the grammar of Python and a program in Python, does the program satisfy all the rules of the grammar.
- Design an algorithm that takes as input the grammar G and string w, and outputs the parse tree for w or returns “syntax error”
- How do we proceed ?
  - Grammars seem to be built “arbitrarily”
    - Algorithm should handle all possible representations
    - Complicates the algorithm

Simplification and Parsing

- provide a procedure to simplify a grammar such that:
  - Resulting grammar generates the same language
  - Resulting grammar has production rules in a specific format
    - Simplifies the data structures and the parsing algorithm

- Normal Forms: specifications on how the production rules must be defined
  - Chomsky Normal Form (CNF)
  - Greiback Normal Form (GNF)
- Parsing Algorithm: Design a parsing algorithm that takes a grammar in a standard form (CNF)

Exercises

- 1. Is this grammar ambiguous ?

  $S \rightarrow aSbS \mid bSaS \mid \lambda$

- 2. grammar G_2: $S \rightarrow aSa \mid bSb \mid \lambda$
  - Show parse tree for abba

- 3. Give a CFG for L= $\{a^m b^n \mid m \neq n,\ m,\ n \geq 0\}$

Grammars for Programming Languages

- The syntax of constructs in a programming language is commonly described with grammars
  - Commonly referred to as **Backus-Naur Form (BNF)**
- Assume that in a hypothetical programming language,
  - Identifiers consist of digits and the letters a, b, or c
  - Identifiers must begin with a letter
- Productions for a sample grammar:

  <id> → <letter> <rest>

  <rest> → <letter> <rest> | <digit> <rest> | λ

  <letter> → a | b | c

  <digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Example: Check the formal grammar that defines the syntax of Python at https://docs.python.org/3/reference/grammar.html