Assignment 5: Context-Free Grammars, Parsing, and Ambiguity

Cameron Brooks
CS321 Introduction to Theory of Computation
Assignment 5

Wednesday, October 31, 2025

Contents

Section 5.1: Context-Free Grammar Construction

Problem 1: CFG for $L = \{a^n b^m : 2n \leq m \leq 3n\}$

Context-Free Grammar:

$$S \to aSbb \mid aSbbb \mid \lambda$$

Design Approach and Reasoning:

The key insight is that we need to maintain the ratio $2n \leq m \leq 3n$ throughout the derivation. The grammar achieves this by:

- **Rule** $S \to aSbb$: For each a added, we add exactly 2 b's (minimum case)
- **Rule** $S \to aSbbb$: For each a added, we add exactly 3 b's (maximum case)
- **Rule** $S \to \lambda$: Base case for the empty string (when $n = 0$, $m = 0$)

By allowing a choice between adding 2 or 3 b's for each a, we can generate any value of m in the range $[2n, 3n]$ for a given n.

How the Grammar Works:

- Start with S
- Apply $S \to aSbb$ or $S \to aSbbb$ recursively n times
- Each application adds one a and either 2 or 3 b's
- Finish with $S \to \lambda$ to terminate
- The total number of b's will be between $2n$ (all minimum choices) and $3n$ (all maximum choices)

Test Cases and Correctness:

Accept Cases (strings in L):

- λ (empty string): $n = 0, m = 0$, satisfies $2(0) \leq 0 \leq 3(0)$ ✓
- abb: $n = 1, m = 2$, satisfies $2(1) \leq 2 \leq 3(1)$ ✓
- $abbb$: $n = 1, m = 3$, satisfies $2(1) \leq 3 \leq 3(1)$ ✓
- $aabbbb$: $n = 2, m = 4$, satisfies $2(2) \leq 4 \leq 3(2)$ ✓
- $aabbbbb$: $n = 2, m = 5$, satisfies $2(2) \leq 5 \leq 3(2)$ ✓
- $aabbbbbb$: $n = 2, m = 6$, satisfies $2(2) \leq 6 \leq 3(2)$ ✓
- $aaabbbbbb$: $n = 3, m = 6$, satisfies $2(3) \leq 6 \leq 3(3)$ ✓

Reject Cases (strings not in L):

- a: $n = 1, m = 0$, violates $2(1) \leq 0$ ×
- ab: $n = 1, m = 1$, violates $2(1) \leq 1$ ×
- $abbbb$: $n = 1, m = 4$, violates $4 \leq 3(1)$ ×
- aab: $n = 2, m = 1$, violates $2(2) \leq 1$ ×
- $aabbb$: $n = 2, m = 3$, violates $2(2) \leq 3$ ×
- $aabbbbbbb$: $n = 2, m = 7$, violates $7 \leq 3(2)$ ×
- ba: Invalid order (b before a) ×
- $aaabbbb$: $n = 3, m = 4$, violates $2(3) \leq 4$ ×

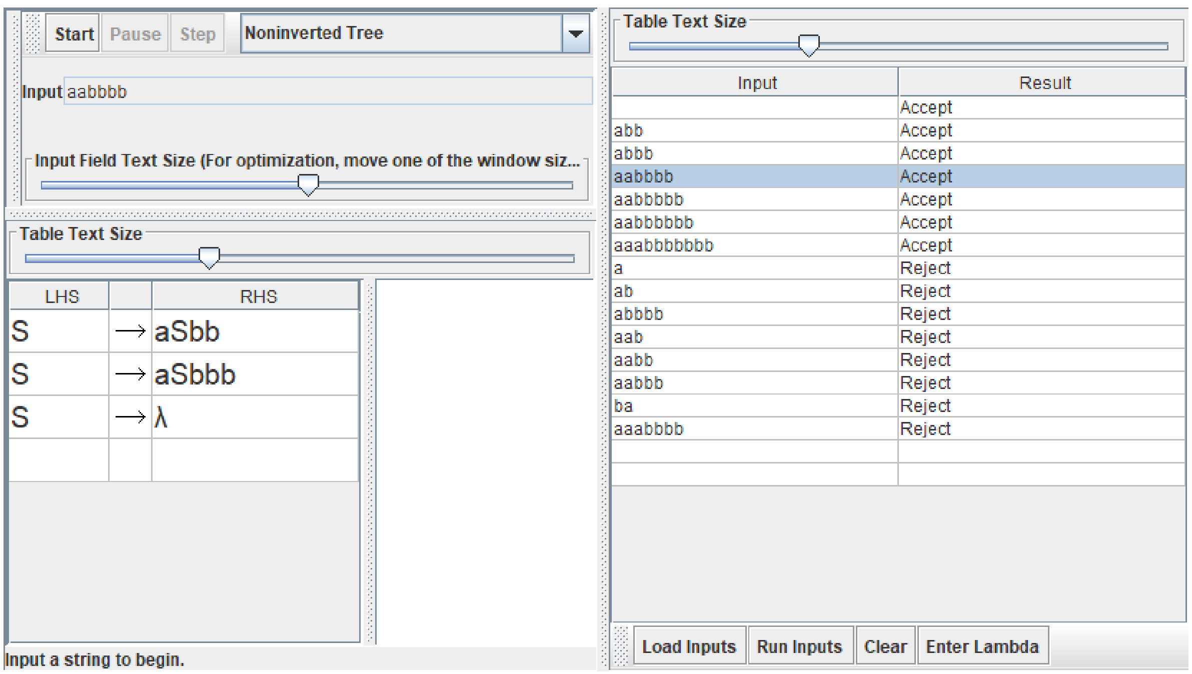

Figure 1: Grammar productions and test results for Problem 1. Accept: λ, abb, abbb, aabbbb, aabbbbb, aabbbbbb, aaabbbbbbb — Reject: a, ab, abbbb, aab, aabbb, aabbbbbbb, ba, aaabbbb

Problem 2: CFG for $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w)\}$

Context-Free Grammar:

$$S \to aabS \mid abaS \mid baaS \mid \lambda$$

Design Approach and Reasoning:

The challenge is that the language allows arbitrary interleaving of a's and b's, as long as the final count maintains $n_a = 2n_b$. The grammar handles this by:

- **Rule** $S \to aabS$: Adds the pattern aab (2 a's, 1 b)
- **Rule** $S \to abaS$: Adds the pattern aba (2 a's, 1 b)
- **Rule** $S \to baaS$: Adds the pattern baa (2 a's, 1 b)
- **Rule** $S \to \lambda$: Base case for the empty string

Each recursive rule adds exactly 2 a's and 1 b in different orderings, covering all possible local arrangements while maintaining the global 2:1 ratio.

How the Grammar Works:

- Start with S
- Apply any combination of the three recursive rules
- Each application adds a "block" of 2 a's and 1 b in some order
- The choice of which rule to apply determines the local ordering
- Finish with $S \to \lambda$ to terminate
- The result is a string where $n_a = 2n_b$ with arbitrary interleaving

Test Cases and Correctness:

Accept Cases (strings in L):

- λ (empty string): $n_a = 0, n_b = 0$, satisfies $0 = 2(0)$ ✓
- aab: $n_a = 2, n_b = 1$, satisfies $2 = 2(1)$ ✓
- aba: $n_a = 2, n_b = 1$, satisfies $2 = 2(1)$ ✓
- baa: $n_a = 2, n_b = 1$, satisfies $2 = 2(1)$ ✓
- $aabaab$: $n_a = 4, n_b = 2$, satisfies $4 = 2(2)$ ✓
- $aabbaa$: $n_a = 4, n_b = 2$, satisfies $4 = 2(2)$ ✓
- $baaaab$: $n_a = 4, n_b = 2$, satisfies $4 = 2(2)$ ✓
- $aaabaabaab$: $n_a = 6, n_b = 3$, satisfies $6 = 2(3)$ ✓

Reject Cases (strings not in L):

- a: $n_a = 1, n_b = 0$, violates $1 = 2(0)$ $\times$
- b: $n_a = 0, n_b = 1$, violates $0 = 2(1)$ $\times$
- aa: $n_a = 2, n_b = 0$, violates $2 = 2(0)$ $\times$
- ab: $n_a = 1, n_b = 1$, violates $1 = 2(1)$ $\times$

- *aaab*: $n_a = 3, n_b = 1$, violates $3 = 2(1)$ $\times$
- *aabb*: $n_a = 2, n_b = 2$, violates $2 = 2(2)$ $\times$
- *aaabb*: $n_a = 3, n_b = 2$, violates $3 = 2(2)$ $\times$
- *aaabbb*: $n_a = 3, n_b = 3$, violates $3 = 2(3)$ $\times$

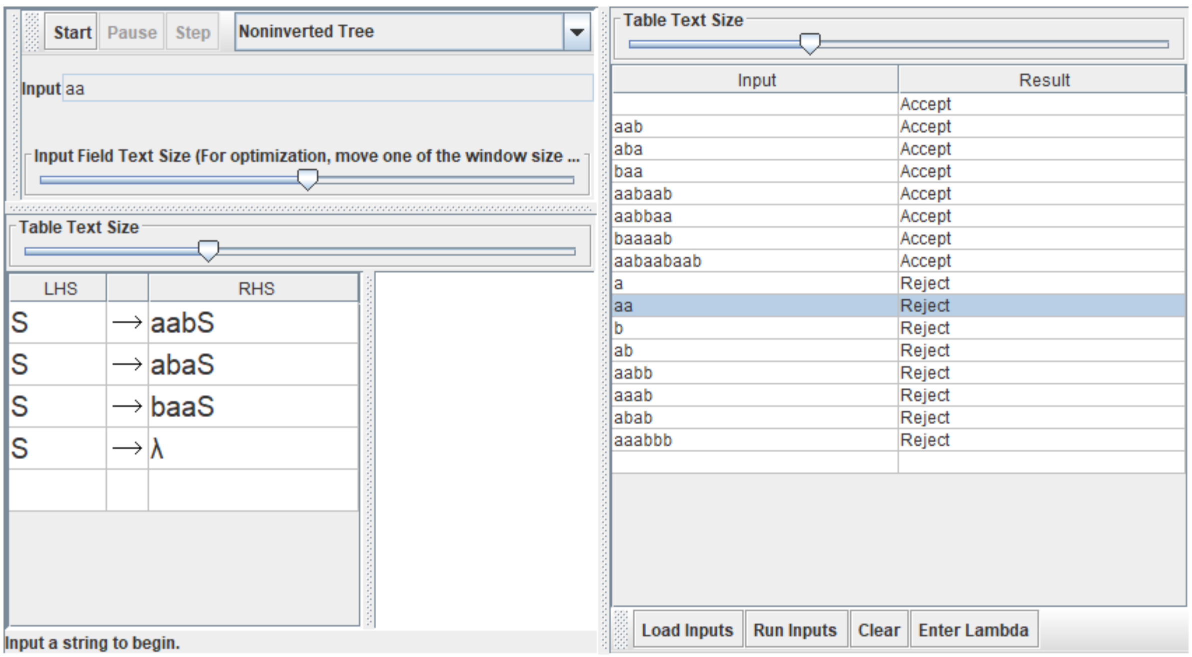

Figure 2: Grammar productions and test results for Problem 2. Accept: aab, aba, baa, aabaab, aabbaa, baaaab, aaabaabaab — Reject: a, b, aa, ab, aaab, aabb, aaabb, aaabbb

Section 5.2: Derivation Trees and Ambiguity

Problem 3: Derivation Tree for $(a + b) * c + d$

Grammar Rules (Modified Example 5.12):

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid I \\ I &\to a \mid b \mid c \mid d \end{aligned}$$

Precedence Hierarchy:

- **Lowest Precedence:** Addition (+) - handled by E rules
- **Medium Precedence:** Multiplication ($*$) - handled by T rules

- **Highest Precedence:** Parentheses and identifiers - handled by F and I rules

Design Approach and Reasoning:
This grammar enforces correct operator precedence through its hierarchical structure:

1. **Expression level (E):** Handles addition, the lowest precedence operator
2. **Term level (T):** Handles multiplication, which binds tighter than addition
3. **Factor level (F):** Handles parentheses (highest precedence) and atomic identifiers
4. **Identifier level (I):** Represents the terminal symbols (variables)

The grammar ensures that multiplication is performed before addition, and parenthesized expressions are evaluated first.

Leftmost Derivation Sequence:
For the string $(a + b) * c + d$:

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow T * F + T \\
&\Rightarrow F * F + T \\
&\Rightarrow (E) * F + T \\
&\Rightarrow (E + T) * F + T \\
&\Rightarrow (T + T) * F + T \\
&\Rightarrow (F + T) * F + T \\
&\Rightarrow (I + T) * F + T \\
&\Rightarrow (a + T) * F + T \\
&\Rightarrow (a + F) * F + T \\
&\Rightarrow (a + I) * F + T \\
&\Rightarrow (a + b) * F + T \\
&\Rightarrow (a + b) * I + T \\
&\Rightarrow (a + b) * c + T \\
&\Rightarrow (a + b) * c + F \\
&\Rightarrow (a + b) * c + I \\
&\Rightarrow (a + b) * c + d
\end{aligned}
$$

Derivation Tree Structure:
The parse tree demonstrates:

1. **Root:** E (the start symbol)
2. **First split:** $E \to E + T$ (outermost addition)
3. **Left subtree:** The expression $(a + b) * c$ is derived from the left E

- This E derives to T (no addition at this level)
- $T \to T * F$ (multiplication)
- Left T derives $(a + b)$ through parentheses
- Right F derives c

4. **Right subtree:** The term d is derived from the right T

The tree structure correctly shows that:

- The outermost operation is addition (+)
- The left operand of this addition is $(a + b) * c$ (multiplication has higher precedence)
- The right operand is d
- Inside the parentheses, $a + b$ is correctly parsed as addition

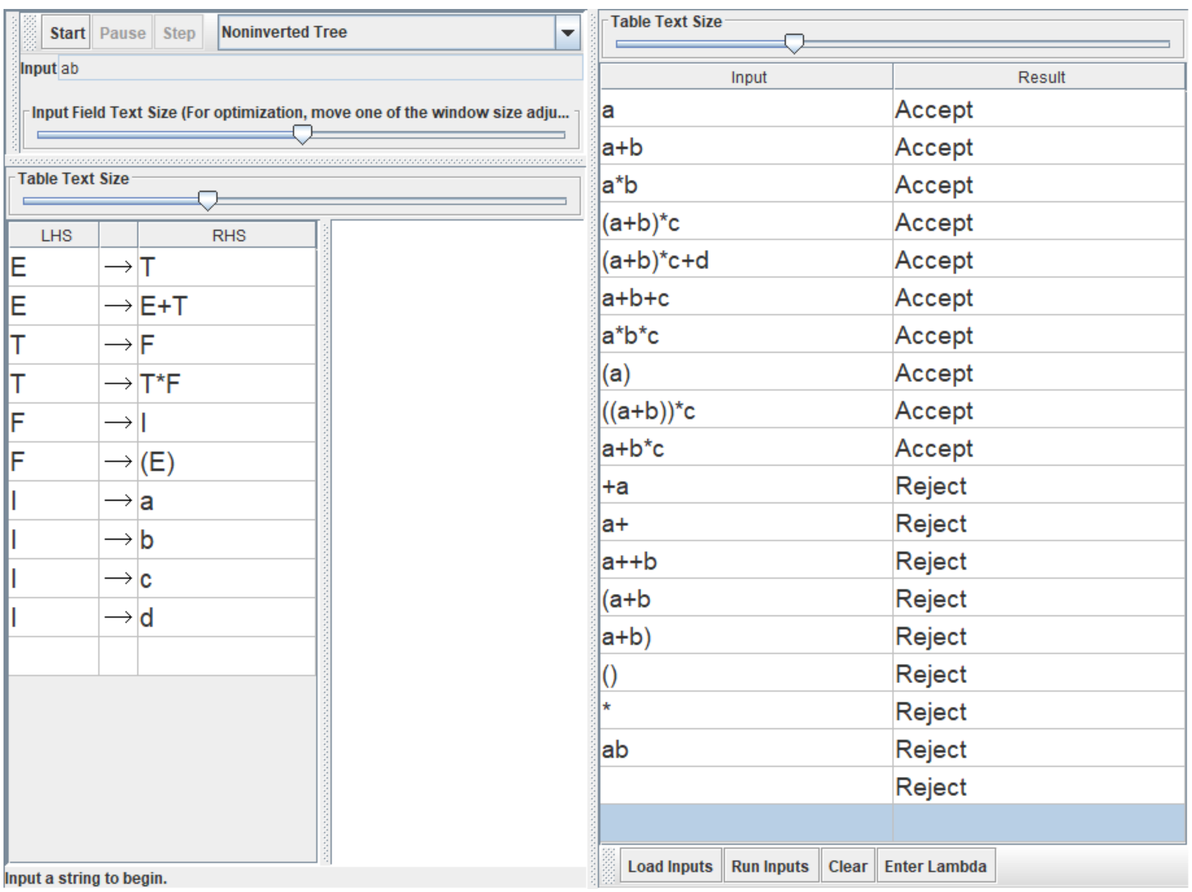

Figure 3: Expression grammar productions and test results. Accept: a, a+b, a*b, (a+b)*c, (a+b)*c+d — Reject: +a, a+, (), *, empty string

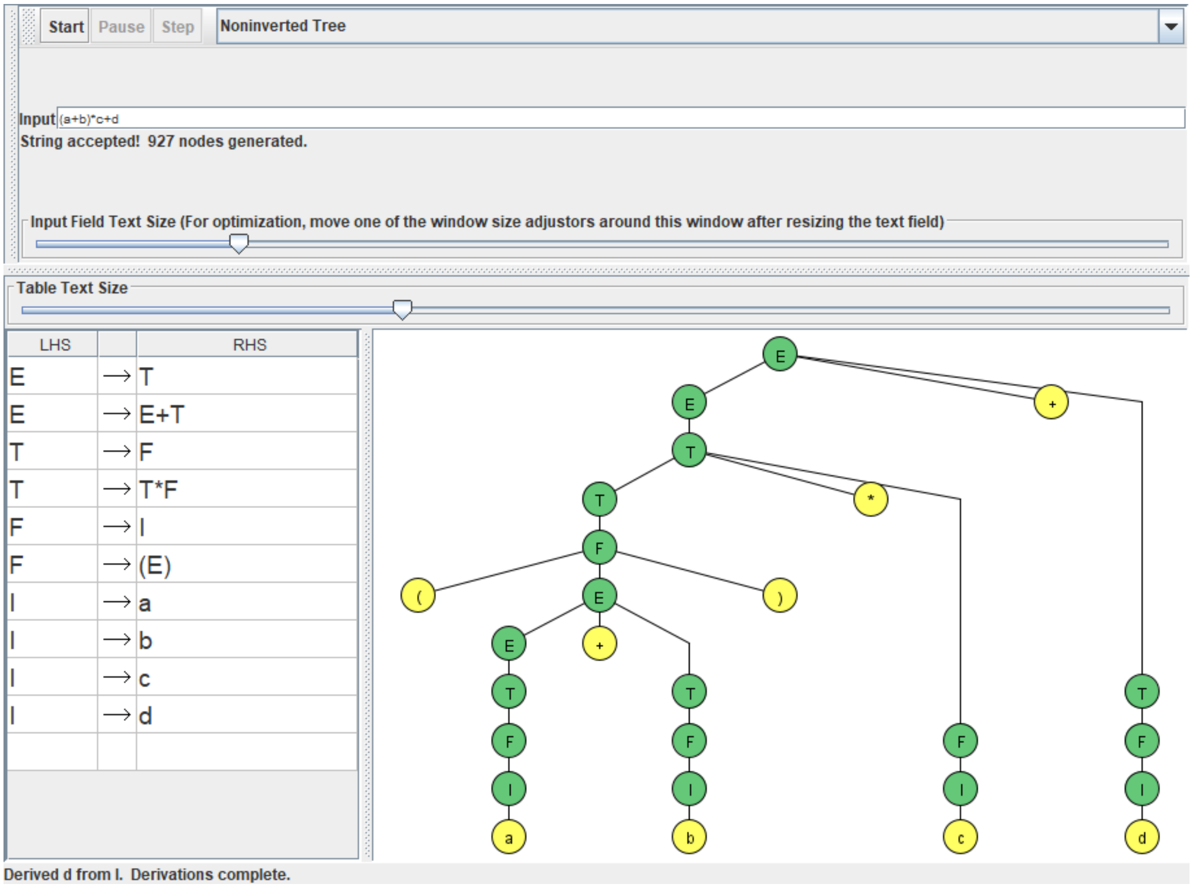

Figure 4: Complete derivation tree for $(a + b) * c + d$ showing correct operator precedence. The tree enforces: $((a + b) * c) + d$

Problem 4: Ambiguity Proof

Grammar: $S \to aSb \mid SS \mid \lambda$

Part I: Proving the Grammar is Ambiguous

Definition: A grammar is ambiguous if there exists at least one string in the language that has two or more distinct parse trees (or equivalently, two or more distinct leftmost derivations).

Proof Strategy: We demonstrate ambiguity by exhibiting a string with two distinct parse trees.

Witness String: $w = aabb$

Derivation 1 (Using $S \to aSb$ first):

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow a\lambda bb \\ &\Rightarrow aabb \end{aligned}$$

Parse Tree 1 Structure:
This tree represents a **nested structure**: the string is parsed as $a(ab)b$, where the inner ab is wrapped by an outer a and b.

Derivation 2 (Using $S \to SS$ first):

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow aSbS \\ &\Rightarrow a\lambda bS \\ &\Rightarrow abS \\ &\Rightarrow abaSb \\ &\Rightarrow aba\lambda b \\ &\Rightarrow aabb \end{aligned}$$

Parse Tree 2 Structure:
This tree represents a **concatenation structure**: the string is parsed as $(ab)(ab)$, where two independent ab substrings are concatenated.

Conclusion: Since the string $aabb$ has two distinct parse trees with different structural interpretations, the grammar $G_A : S \to aSb \mid SS \mid \lambda$ is **ambiguous**.

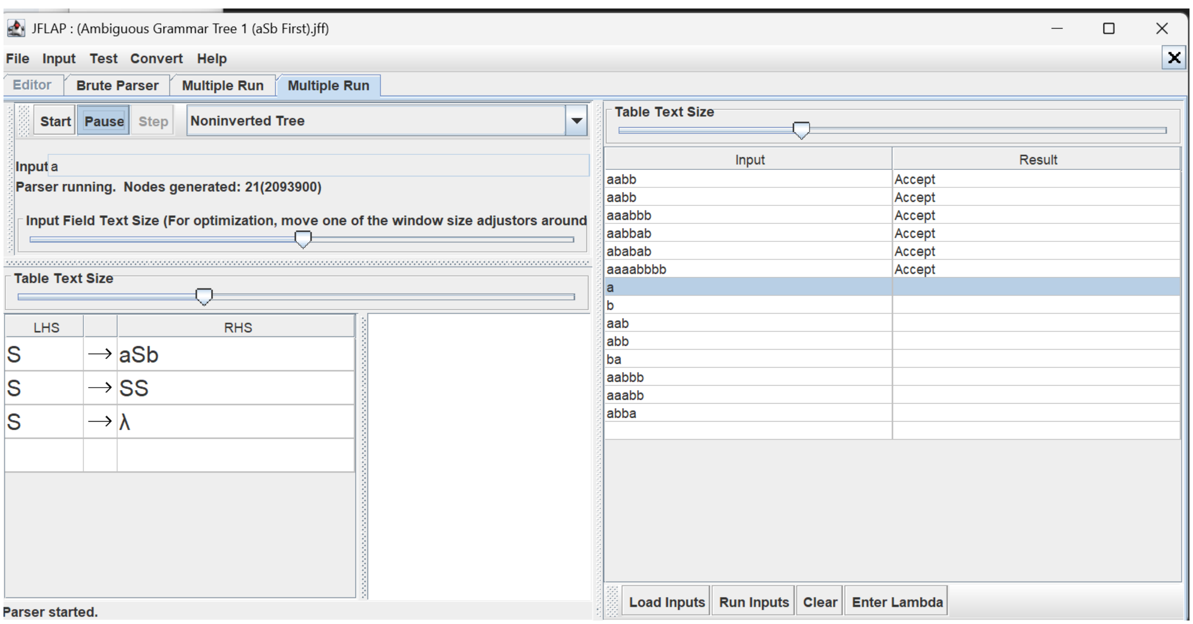

Figure 5: Ambiguous grammar test results showing accepted and rejected strings

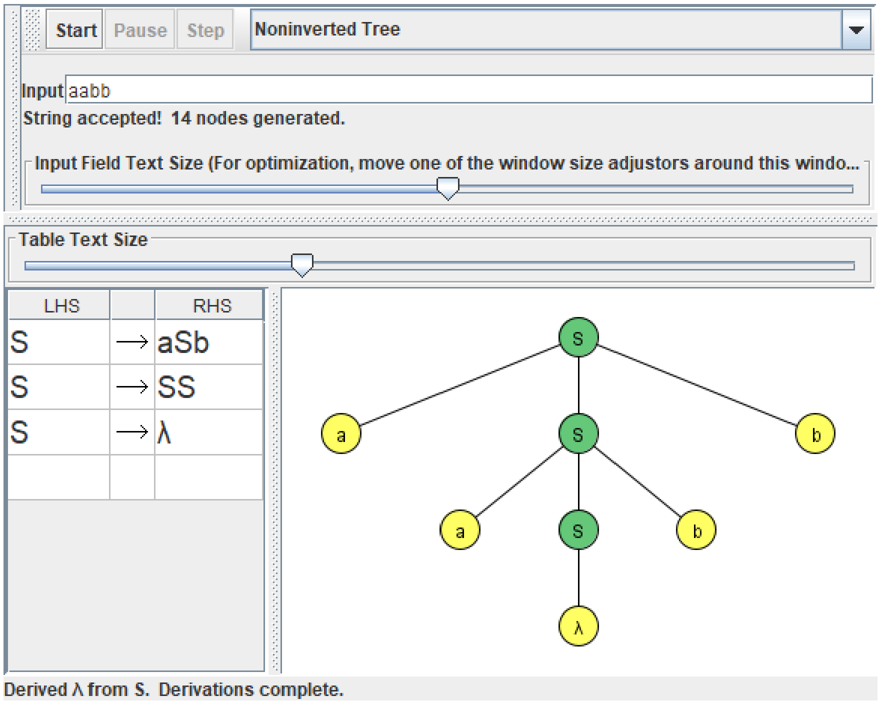

Figure 6: Parse Tree 1 for $aabb$ using nested structure ($S \to aSb$ first): $a(a()b)b$

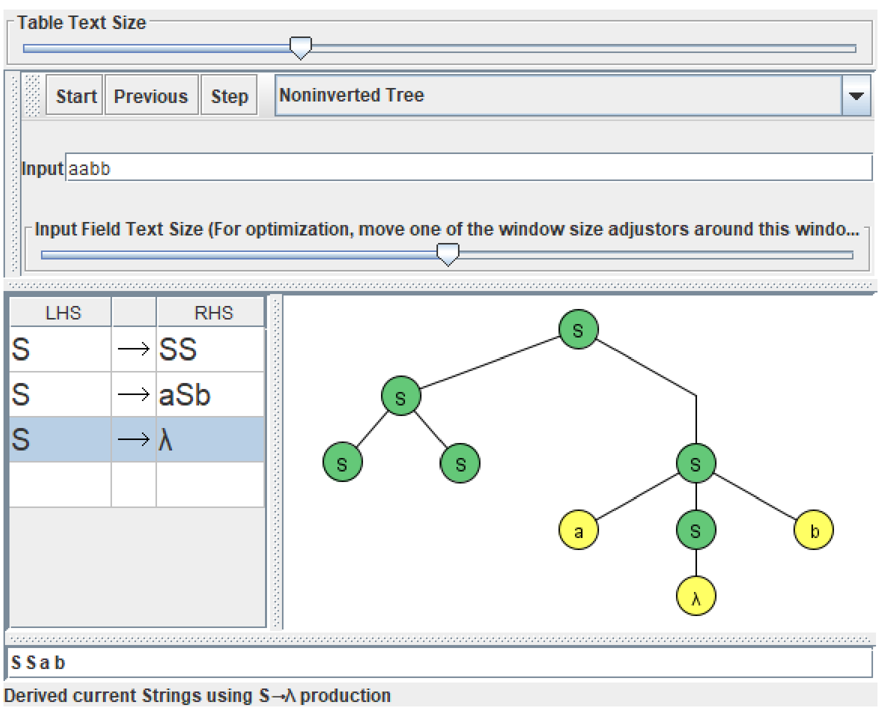

Figure 7: Parse Tree 2 for $aabb$ using concatenation structure ($S \to SS$ first): $(ab)(ab)$

Part II: Proving the Language is NOT Inherently Ambiguous

Definition: A language is inherently ambiguous if every grammar that generates it is ambiguous. Conversely, a language is NOT inherently ambiguous if there exists at least one unambiguous grammar that generates it.

Proof Strategy: We construct an unambiguous grammar that generates the same language as G_A.

Language Analysis:

First, we determine what language G_A generates. The grammar has three rules:

- $S \to aSb$: Adds matching a and b on the outside
- $S \to SS$: Concatenates two strings from the language
- $S \to \lambda$: Generates the empty string

By analyzing the grammar, we can show that $L(G_A) = \{w \in \{a, b\}^* : n_a(w) = n_b(w)$ and every prefix ha $n_b\}$. This is the language of **balanced parentheses** (if we think of a as "(" and b as ")").

Unambiguous Grammar:

$$G_U : S \to aSbS \mid \lambda$$

Why G_U is Unambiguous:

The grammar G_U has only two rules, and the structure is carefully designed:

1. $S \to aSbS$: This rule adds an a, then recursively generates a balanced substring, then adds a b, then recursively generates another balanced substring
2. $S \to \lambda$: Base case

For any string in the language, there is exactly one way to parse it:

- The first a in the string must be matched with its corresponding b (the first b that closes the balanced prefix)
- This uniquely determines where to split the string
- The grammar enforces a **canonical parsing** where each a is immediately followed by its matching balanced substring before the closing b

Proof that $L(G_U) = L(G_A)$:

We need to show that both grammars generate the same language.

($L(G_U) \subseteq L(G_A)$): Every string generated by G_U can be generated by G_A:

- λ is generated by both
- If G_U generates $aSbS$, we can use G_A to generate aSb (from the first part) and S (from the second part), then concatenate using SS

($L(G_A) \subseteq L(G_U)$): Every string generated by G_A can be generated by G_U:

- This requires showing that the concatenation rule SS in G_A can be simulated by the $aSbS$ rule in G_U
- The key insight is that any balanced string can be decomposed into a form where the first a is matched with a specific b, and the remaining parts are balanced

Verification with JFLAP:
The screenshots show that G_U accepts the same set of test strings as G_A:

- Both accept: λ, ab, $aabb$, $abab$, $aaabbb$, etc.
- Both reject: a, b, ba, abb, aab, etc.

Conclusion: Since we have constructed an unambiguous grammar G_U that generates the same language as G_A, the language $L(G_A)$ is **NOT inherently ambiguous**.

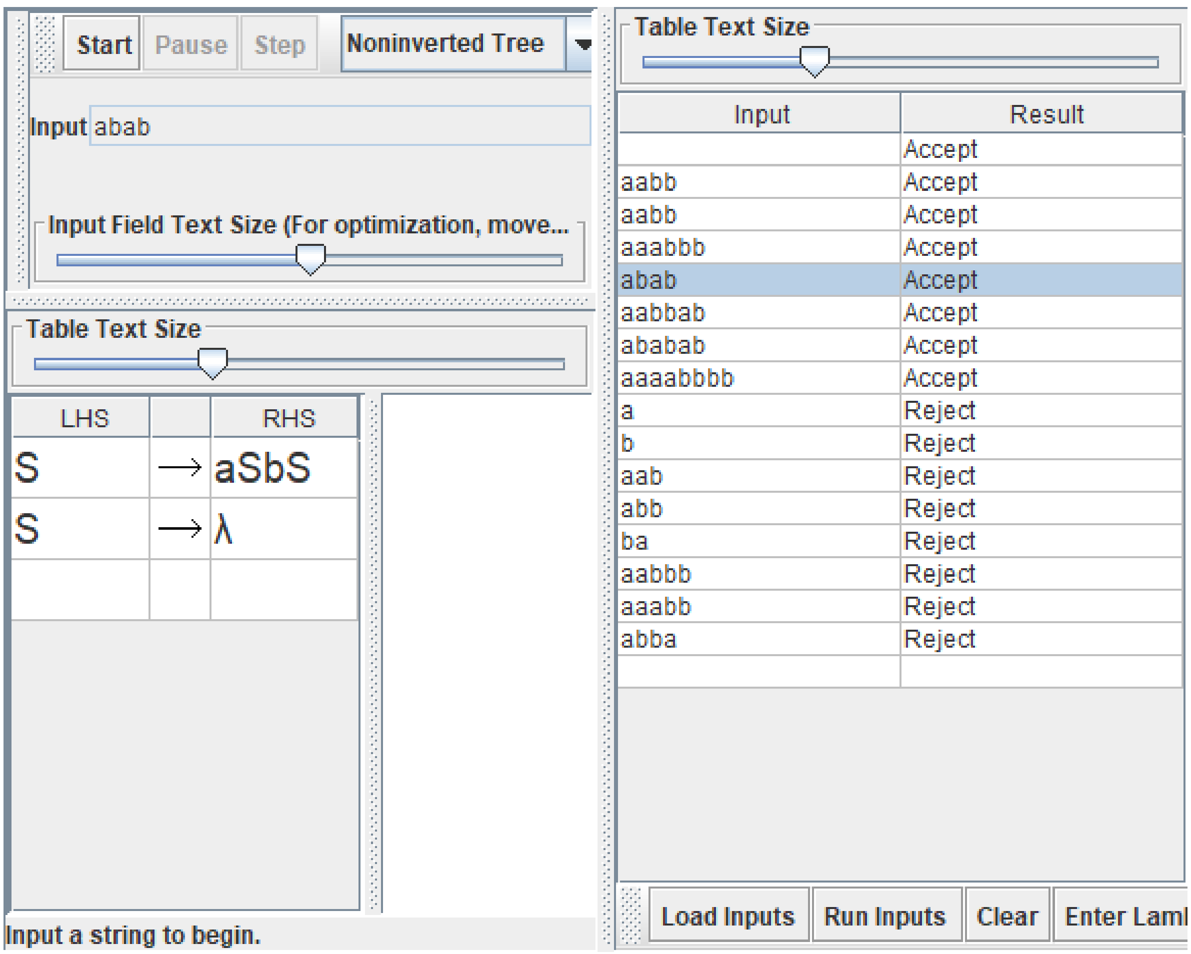

Figure 8: Unambiguous grammar test results showing identical accept/reject behavior to the ambiguous grammar

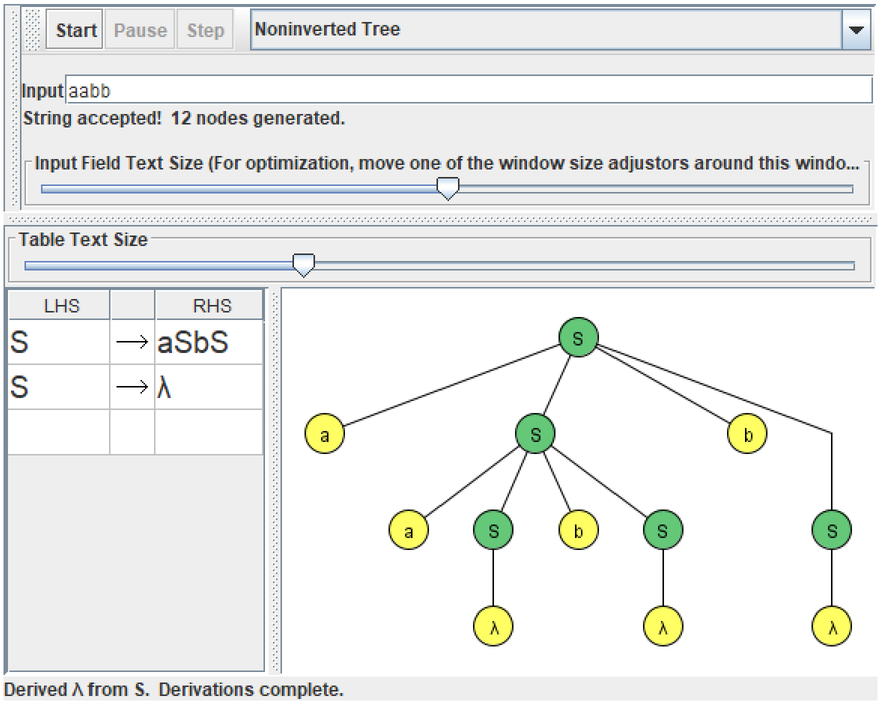

Figure 9: Parse tree for $aabb$ using the unambiguous grammar $S \to aSbS \mid \lambda$, showing unique parsing

Summary:

- **Part I:** The grammar $S \to aSb \mid SS \mid \lambda$ is ambiguous (proven by exhibiting two parse trees for $aabb$)
- **Part II:** The language generated by this grammar is not inherently ambiguous (proven by constructing the unambiguous grammar $S \to aSbS \mid \lambda$)

Problem 5: Unambiguous Single-Production Grammars

Theorem Statement:

Theorem: Let $G = (V, T, S, P)$ be a context-free grammar such that each variable $A \in V$ appears on the left-hand side of at most one production in P. Then G is unambiguous.

Proof:

We prove this directly by showing that every string $w \in L(G)$ has a unique derivation tree.

We proceed by strong induction on the number of derivation steps required to generate w.

Base Case ($n = 1$):

If w is derived in one step, the derivation is $S \to w$. Since S has at most one production in P, this derivation (and its resulting tree) is uniquely determined.

Inductive Hypothesis:

Assume that for all strings derivable in k or fewer steps ($k \geq 1$), the derivation tree is unique.

Inductive Step (Conclusion):

Consider a string w derived in $k+1$ steps. The derivation must begin with the first production:

$$S \Rightarrow \alpha$$

where $\alpha = \alpha_1\alpha_2 \ldots \alpha_m$. Since the variable S appears on the left side of at most one production, this initial step $S \to \alpha$ is uniquely determined.

The final string w is partitioned as $w = w_1 w_2 \ldots w_m$. For any $\alpha_i \in V$ (a variable), w_i is derived from α_i in fewer than $k+1$ steps. Since each variable α_i has at most one production, the derivation tree for w_i from α_i is unique by the inductive hypothesis.

Since:

1. The first step ($S \to \alpha$) is unique.
2. Every subsequent subtree derived from α_i is unique.

The entire derivation tree for w is uniquely determined.

By the principle of mathematical induction, every string in $L(G)$ has a unique derivation tree.

Therefore, the grammar **G** is unambiguous. □

Intuitive Explanation:

The key insight of this proof is that the constraint "no variable appears on the left side of more than one production" eliminates all sources of ambiguity:

1. **No choice at any step:** When deriving from a variable A, there is at most one production $A \to \beta$ to apply
2. **Deterministic derivation:** The derivation process becomes deterministic—at each step, there is only one possible production to apply
3. **Unique parse tree:** Since every derivation step is forced, there can be only one parse tree for any string

This is a very restrictive condition on grammars (most useful grammars have multiple productions for the same variable), but it guarantees unambiguity.

Example:

Consider the grammar:

$$S \to AB$$
$$A \to a$$
$$B \to b$$

Each variable (S, A, B) appears on the left side of exactly one production. For the string ab:

- Must start with $S \to AB$ (only production for S)
- Must derive $A \to a$ (only production for A)

- Must derive $B \to b$ (only production for B)
- Result: unique derivation tree

This grammar satisfies the theorem's hypothesis and is indeed unambiguous.