CM50260 Foundations of Computation Coursework: Putting Theory Into Practice

...

November 20, 2023

Contents

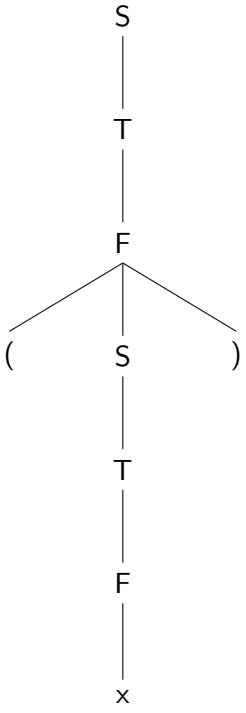
1 Question 7(a)(i)

Given the grammar G:

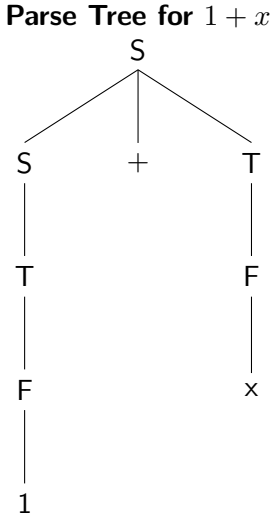
1. $S \to S + T \mid T$
2. $T \to T \times F \mid F$
3. $F \to (S) \mid 1 \mid 0 \mid x$

Below are the parse trees for the given expressions:

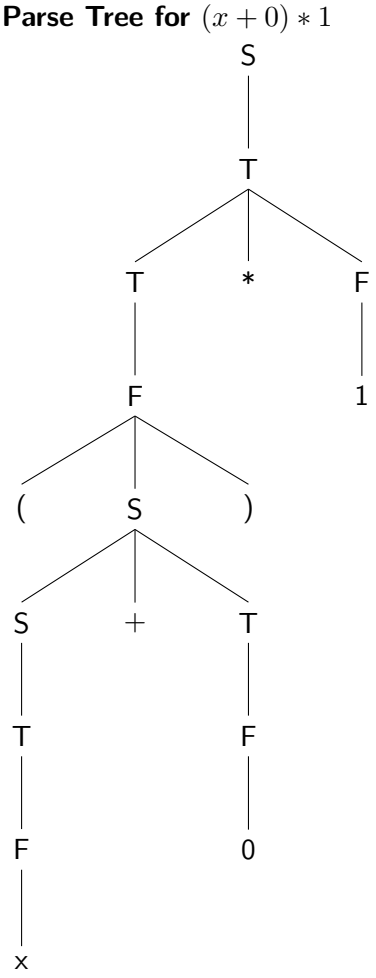
Parse Tree for (x)



Parse Tree for $1 + x$



Parse Tree for $(x + 0) * 1$



2 Question 7(a)(ii)

Given the grammar G:

1. $S \to S + T \mid T$
2. $T \to T \times F \mid F$
3. $F \to (S) \mid 1 \mid 0 \mid x$

We proceed with the conversion to Chomsky Normal Form (CNF) as follows:

Step 1: Add New Start Variable

Introduce a new start variable S_0 and a rule that produces the original start symbol.

New rule:

- $S_0 \to S$

Step 2: Eliminate ε-rules

This grammar does not contain any ε-rules (rules of the form $A \to \varepsilon$), so we can skip this step.

Step 3: Eliminate Unit Rules

Unit rules are those of the form $A \to B$. We need to replace each unit rule with the rules produced by the variable on the right-hand side of the rule.

We now eliminate unit rules from the grammar:

- For $S \to T$, replace with the productions of T.
- For $T \to F$, replace with the productions of F.

After Step 1 to 3, the grammar becomes:

1. $S_0 \to S$
2. $S \to S + T \mid T \times F \mid (S) \mid 1 \mid 0 \mid x$
3. $T \to T \times F \mid (S) \mid 1 \mid 0 \mid x$
4. $F \to (S) \mid 1 \mid 0 \mid x$

Step 4: Eliminate Rules with $k \geq 3$ Symbols

We identify rules that need to be modified:

- $S \to S + T$ can be rewritten by introducing a new variable, $U \to +T$.
- $S \to T \times F$ and $T \to T \times F$ are similar and can be addressed introducing a new variable, $V \to \times F$.
- $F \to (S)$, $T \to (S)$ and $S \to (S)$ are similar and can be addressed by introducing a new variable, $W \to S)$.

The grammar becomes:

1. $S_0 \to S$
2. $S \to SU \mid TV \mid (W \mid 1 \mid 0 \mid x$
3. $T \to TV \mid (W \mid 1 \mid 0 \mid x$
4. $F \to (W \mid 1 \mid 0 \mid x$
5. $U \to +T$
6. $V \to \times F$
7. $W \to S)$

Step 5: Eliminate Terminals in Complex Rules

In this final step, we replace terminals in complex rules (rules with more than one symbol on the right-hand side) with appropriate non-terminal symbols.

We introduce new variables only for terminals in rules with more than one symbol. Rules where a terminal appears alone are left unchanged, as they are already in Chomsky Normal Form.

- $A_+ \to +$
- $A_\times \to \times$

- $A_(\to ($
- $A_) \to)$

Now, replace terminals in complex rules with these variables:

1. $S_0 \to S$
2. $S \to SU \mid TV \mid A_(W \mid 1 \mid 0 \mid x$
3. $T \to TV \mid A_(W \mid 1 \mid 0 \mid x$
4. $F \to A_(W \mid 1 \mid 0 \mid x$
5. $U \to A_+T$
6. $V \to A_\times F$
7. $W \to SA_)$
8. $A_+ \to +$
9. $A_\times \to \times$
10. $A_(\to ($
11. $A_) \to)$

With these changes, the grammar is now in Chomsky Normal Form.

3 Question 7(b)(i)

When implementing a parser for context-free grammars in Chomsky Normal Form, particularly for the task of listing all possible derivations of length $2|w| - 1$ to determine if a word w is in the language $L(G)$ of the grammar G, the representation of derivations $S \Rightarrow^* w$ (with S as the start symbol) can be approached in many ways, balancing different requirements.

Here I present 2 data structures that are illustrative but not exhaustive, List of Tuples and Tree Structure. Both have unique characteristics suitable for different aspects of parsing.

3.1 List of Tuples

How it Works

- This method uses a list where each element is a tuple representing a production rule applied in the derivation.
- A tuple is of the form $(A, [B, C])$ or $(A, [a])$, where A is a non-terminal, and B, C or a are the symbols produced by the rule.
- Each tuple directly corresponds to a single production step, making it ideal for tracking one-step derivations.
- The list sequentially records the production rules applied from the start symbol S to derive the word w.

Pros

- Easy to implement and understand.
- Only stores necessary information about the applied production rules.

Cons

- Does not inherently represent the hierarchical nature of the parse tree.
- Requires additional processing to reconstruct the full derivation path or the parse tree.

3.2 Tree Structure

How it Works

- Each node in the tree represents a symbol (terminal or non-terminal).
- Each node expansion in the tree represents a one-step derivation, aligning with the CNF grammar rule structure.
- The tree is constructed by applying production rules, where non-terminal nodes expand according to these rules, reflecting the derivation steps.

Pros

- Clearly shows the hierarchical structure of derivations.
- Aligns well with tree-based parsing algorithms.

Cons

- Storing the entire tree structure can be more memory-intensive.
- Requires more sophisticated data structures and algorithms.

4 Question 7(b)(ii)

Converting a derivation $S \Rightarrow^* w$ into a parse tree for w involves different approaches depending on data structure used to represent the derivation. Below I describe algorithms for each data structure.

4.1 List of Tuples

Algorithm Overview

1. Start with an empty tree or a root node labeled with the start symbol S.
2. Go through each tuple in the derivation list, which represents the production rules applied in the order they were used.
3. For each tuple $(A, [B, C])$ or $(A, [a])$, find the leftmost unexpanded instance of A in your current parse tree and replace it with a subtree. The subtree has A as the parent, and B and C (or a) as children, depending on the rule.
4. Continue this process until all tuples are processed. The final tree is the parse tree for w.

Pros

- Straightforward linear processing of the derivation.
- Allows flexibility in tree construction and easy backtracking if needed.

Cons

- Requires additional logic to find the correct position in the parse tree for each expansion.
- Might be less efficient in terms of tree construction compared to directly using a tree structure.

4.2 Tree Structure

Algorithm Overview

1. Building the derivation tree is really where most of the work takes place. As you process the input word w, build the derivation tree by applying production rules from the CNF grammar. Each non-terminal node expands according to these rules.
2. The derivation tree is essentially the parse tree.

Pros

- The parse tree is built inherently during the derivation process, requiring less post-processing.

Cons

- Can be memory-intensive for complex grammars or long derivations.

5 Question 7(c)(i)

The Cocke-Younger-Kasami (CYK) algorithm is a clever way to parse valid strings using context-free grammars. It's a bit like solving a puzzle from the ground up, instead of going through every single permutation. The algorithm breaks down a string into smaller parts , substrings, and discerns how these can be assembled in accordance with the grammar rules.

To achieve this, the CYK algorithm employs a table-filling approach. It starts with the smallest substrings and gradually progresses to larger ones. For each substring, the algorithm evaluates which grammar rules could have generated it. This methodical progression from simple to complex elements is key to its efficiency.

A notable advantage of the CYK algorithm over simpler parsing methods is its optimized space and time complexity. While the complexity of simpler methods can grow rapidly, becoming impractical for longer strings, the CYK algorithm employs dynamic programming. This strategy allows it to store and reuse intermediate results, thereby curtailing the exponential growth in complexity. This efficiency is particularly valuable for parsing longer strings. Additionally, its adeptness at handling strings with multiple valid interpretations enhances its utility in both natural language processing and in various user-defined grammars."

6 Question 7(c)(ii)

Given the grammar over the alphabet {m, x, p} defined by the following rules, where S is the start symbol:

$$S \to XT$$
$$X \to x \mid XT \mid NX$$
$$T \to XP$$
$$N \to MX$$
$$M \to m$$
$$P \to p$$

We use the CYK algorithm to generate a derivation for the word $xmxp$. Note that the grammar is already in Chomsky normal form and does not require further processing. We apply the CYK algorithm to generate the following table:

1	x $X \to x$	m $M \to m$	p $P \to p$	
2	xm ~~$XM \to xm$~~	mx $N \to MX \to mx$	xx ~~$XX \to xx$~~	xp $T \to XP \to xp$
3	xmx ~~$XN \to x \cdot mx$~~ ~~$xm \cdot x$~~	mxx ~~$m \cdot xx$~~ $X \to NX \to mx \cdot x$	xxp $S \to XT \to x \cdot xp$ $X \to XT \to x \cdot xp$ ~~$xx \cdot p$~~	
4	$xmxx$ $XX \to x \cdot mxx$ ~~$xm \cdot xx$~~ ~~$xmx \cdot x$~~	$mxxp$ ~~$MS \to m \cdot xxp$~~ $N \to MX \to m \cdot xxp$ ~~$NT \to mx \cdot xp$~~ $T \to XP \to mxx \cdot p$		
5	$xmxxp$ ~~$XN \to x \cdot mxxp$~~ $S \to XT \to x \cdot mxxp$ ~~$xm \cdot xxp$~~ ~~$xmx \cdot xp$~~ ~~$XXP \to xmxx \cdot p$~~			

Starting from the bottom-left cell of the table generated by applying the CYK algorithm, we can observe that the given word $xmxxp$ can be generated from this grammar. The derivation is as follows:

1. S (from x-$mxxp$) $\to XT$
2. XT (from x) $\to xT$
3. xT (from mxx-p) $\to xXP$
4. xXP (from mx-x) $\to xNXP$
5. $xNXP$ (from mx) $\to xMXXP$
6. $xMXXP$ (from m) $\to xmXXP$
7. $xmXXP$ (from x) $\to xmxXP$
8. $xmxXP$ (from x) $\to xmxxP$
9. $xmxxP$ (from p) $\to xmxxp$