Untitled

by Jaglike Makkar

General metrics

4,666	792	71	3 min 10 sec	6 min 5 sec
characters	words	sentences	reading time	speaking time

Score

86

This text scores better than 86% of all texts checked by Grammarly

Writing Issues

30	✓	30
Issues left	Critical	Advanced

Plagiarism

6 %

5 sources

6% of your text matches 5 sources on the web or in archives of academic publications

Writing Issues

20	**Clarity**
17	Passive voice misuse
1	Word choice
2	Wordy sentences
10	**Engagement**
10	Word choice

Unique Words

Measures vocabulary diversity by calculating the percentage of words used only once in your document

25%

unique words

Rare Words

Measures depth of vocabulary by identifying words that are not among the 5,000 most common English words.

40%

rare words

Word Length

Measures average word length

4.6

characters per word

Sentence Length

Measures average sentence length

11.2

words per sentence

Untitled

Syntax Analysis

Keyword: Syntax Analysis

Meta Description: In this article, we will study syntax analysis in compiler design. We will see why it is necessary and how a compiler performs syntax analysis. We will learn several aspects related to syntax analysis, such as derivations, parse trees, etc.

Introduction

An input string is passed[1] through several phases in a compiler. The first phase [31] is the lexical[2] analysis, where the input is scanned and is divided[3] into tokens. Syntax analysis is the second phase of a compiler. The output of syntax analysis is used[4] as input to the semantic analyzer.

In syntax analysis, the compiler checks the syntactical structure of the input string, i.e., whether the given string[5] follows the grammar or not. It uses a data structure called a parse tree[6] or syntax tree to make this comparison. The parse tree is formed[7] by matching the input string with the pre-defined grammar. If the parsing is successful, the grammar can form[8] the given string[9]. Else an error is reported.

Importance of Syntax Analysis

1. It is used[10] to check if the code is grammatically correct or not.
2. It helps us to detect all types of syntax errors.
3. It gives an exact[11] description of the error.
4. It rejects invalid code before actual compiling.

Parsing Techniques

The parsing techniques can be divided[12] into two types:

32 1. **Top-down parsing:** The parse tree is constructed[13] from the root to the leaves in top-down parsing. Some most common top-down parsers are Recursive Descent Parser and LL parsers.
33 2. **Bottom-up parsing: The parse tree is constructed from the leaves to the tree's root in bottom-up parsing. Some examples of bottom-up parsers are the LR parser, SLR parser, CLR parser, etc.**

Derivation

The derivation is the process of using the production rules (grammar) to derive the input string. There are two decisions that the parser must make to form the input string:

1. Deciding which non-terminal is to be replaced[14]. There are two options to do this:
2. a) Left-most Derivation: When the non-terminals are replaced[15] from left to right, it is called left-most derivation.
3. b) Right-most Derivation: When the non-terminals are replaced[16] from right to left, it is called right-most derivation.
4. Deciding the production rule using which the non-terminal will be replaced[17].

Parse Tree

Parse trees are a graphical representation of the derivation. It is used[18] to see how the given string is derived[19] from the start symbol. The root is the start symbol of a parse tree, and the characters of the input string become the leaves.

Example

Consider the following set of production rules where ‘E’ is a non-terminal and ‘id’ is a terminal:

E -> E + E

E -> E * E

E -> id

We will construct a parse tree using the left-most derivation of “id + id * id.”

Steps

Parse Tree

Step-1: Replace E with E * E.

Result: E * E

Step-2: Replace leftmost E with E + E

Result: E + E * E

Step-3,4,5: Replace all E’s with id.

Result : id + id * id

Thus, we can generate a given string by following the production rules and using parse trees for visualization.

Ambiguity: Grammar is ambiguous if there is more than one parse tree for any string.
For example, for the above string and grammar, we can construct two parse trees:

Ambiguous grammar is not considered[20] suitable for a compiler design. No method can detect ambiguity or remove ambiguity. If the grammar is ambiguous[21], one must remove it by either rewriting the whole grammar or following associativity and precedence constraints.

Limitations of Syntax Analysis:

1. It cannot determine if the token is a valid token or not.
2. It cannot determine whether a token[22] is used[23] before or not.
3. It cannot determine whether the operation performed on tokens[24] is valid[25] or not[26].
4. It cannot tell whether the token[27] was initialized[28] or not.

FAQs

1. **What is Syntax Analysis?**
2. **Syntax analysis is the second phase of a compiler. In syntax analysis, the compiler checks the syntactical structure of the input string, i.e., whether the given string[29] follows the grammar or not.** [34]

3. **What are Parse Trees?**
4. **Parse trees or syntax trees are the data structures used by Syntax Analyzer to check if the input string can be formed using the given production rules or not. The start symbol forms the root of the parse tree and the string characters from the leaves.**
5. **What is ambiguity in syntax analysis?**
6. 35 **Grammar is ambiguous if there is more than one parse tree for any string. Such grammar is not considered[30] suitable for a compiler design.**

Key Takeaways

In this article, we learned about syntax analysis in compiler design. We discussed the importance and limitations of syntax analysis. We also how a compiler does syntax analysis using derivations and parse trees.

1.	*is passed*	Passive voice misuse	Clarity
2.	~~lexical~~ → linguistic	Word choice	Clarity
3.	*is divided*	Passive voice misuse	Clarity
4.	*is used*	Passive voice misuse	Clarity
5.	~~string~~ → line, series	Word choice	Engagement
6.	~~tree~~	Wordy sentences	Clarity
7.	*is formed*	Passive voice misuse	Clarity
8.	~~form~~ → create, start, include	Word choice	Engagement
9.	~~string~~ → line	Word choice	Engagement
10.	*is used*	Passive voice misuse	Clarity
11.	~~an exact~~ → a detailed, an accurate	Word choice	Engagement
12.	*be divided*	Passive voice misuse	Clarity
13.	*is constructed*	Passive voice misuse	Clarity
14.	*be replaced*	Passive voice misuse	Clarity
15.	*are replaced*	Passive voice misuse	Clarity
16.	*are replaced*	Passive voice misuse	Clarity
17.	*be replaced*	Passive voice misuse	Clarity
18.	*is used*	Passive voice misuse	Clarity
19.	*is derived*	Passive voice misuse	Clarity
20.	*is not considered*	Passive voice misuse	Clarity
21.	~~ambiguous~~ → unclear	Word choice	Engagement

22.	~~token~~ → permit, ticket, pass	Word choice	Engagement
23.	*is used*	Passive voice misuse	Clarity
24.	~~tokens~~ → tickets	Word choice	Engagement
25.	~~valid~~ → good, correct	Word choice	Engagement
26.	~~or not~~	Wordy sentences	Clarity
27.	~~token~~ → ticket	Word choice	Engagement
28.	*was initialized*	Passive voice misuse	Clarity
29.	~~string~~ → line, series	Word choice	Engagement
30.	*is not considered*	Passive voice misuse	Clarity
31.	*Syntax analysis is the second phase of a compiler. The*	Compiler Construction → Follow set of a given grammar ... https://cuitutorial.com/courses/compiler-construction/lessons/follow-set-of-a-given-grammar-using-array/	Originality
32.	*Top-down parsing: The parse tree is constructed from*	Compare Top-down parsing and Bottom-up parsing? - Blogger https://rsmmukesh.blogspot.com/2012/03/compare-top-down-parsing-and-bottom-up.html	Originality
33.	*Bottom-up parsing: The parse tree is constructed from the*	第九回: 上向き構文解析の原理 - 青山学院大学 https://www.sw.it.aoyama.ac.jp/2016/Compiler/lecture9.html	Originality
34.	*Syntax analysis is the second phase of a compiler.*	What is Syntax Analysis in Compiler? Definition, Types ... https://binaryterms.com/syntax-analysis.html	Originality
35.	*Grammar is ambiguous if there is more than one parse tree for*	CS143 Lecture Notes - Lecture 1 4/3/18: - Languages can be ...	Originality

https://www.studocu.com/en-us/document/stanford-university/compilers/cs143-lecture-notes/5407043