

二 {} > ChatGPT 5.2 Thinking ▾

🔗 ⏷ ⏸ ⋮

i will give you a source code that takes a pif as input and parses it according to the grammar  
I will also attach the grammar

Perfect — send both files (the parser source code and the grammar file).

Once you attach them, I'll:

- trace how the code reads the grammar + PIF,
- explain how the parsing table / stack logic works,
- point out any bugs or edge-cases (epsilon handling, FOLLOW sets, token mapping, sync/panic mode, etc.),
- and adapt/fix the code if needed so it parses **your** grammar format and **your** PIF format.

🔗 👍 🗺 ⏪ ⋮

main.py  
Python

grammar.txt  
Document

transform the source code from python to C++

Thought for 1m 22s >

Thought for 1m 37s >

Here's the **C++17** equivalent of your Python LL(1) parser (same grammar format, FIRST/FOLLOW, LL(1) table, parsing + parse-tree with father/sibling, and PIF→tokens mapping), translated from your `main.py` and compatible with your `grammar.txt`

🔗 grammar .

[Download parser.cpp](#)

Build + run:

bash

🔗 Copy code

```
g++ -std=c++17 -O2 -Wall -Wextra parser.cpp -o parser

# Requirement 1 (sequence of tokens given on CLI)
./parser req1 req1/seminar_grammar.txt a + a

# Requirement 2 (read tokens from a PIF file)
./parser req2 grammar.txt prog1_PIF.txt
```

Sources



This is the output generated by your code:

| Idx | Symbol | Father | Sibling |
|-----|--------|--------|---------|
|-----|--------|--------|---------|

| Idx | Symbol          | Father | Sibling |
|-----|-----------------|--------|---------|
| 0   | program         | -1     | -1      |
| 1   | statement_list  | 0      | -1      |
| 2   | statement       | 1      | 3       |
| 3   | statement_list  | 1      | -1      |
| 4   | assignment_stmt | 2      | -1      |
| 5   | ID              | 4      | 6       |
| 6   | ASSIGN          | 4      | 7       |
| 7   | STRING          | 4      | -1      |
| 8   | statement       | 3      | 9       |
| 9   | statement_list  | 3      | -1      |
| 10  | load_stmt       | 8      | -1      |
| 11  | LOAD            | 10     | 12      |
| 12  | ID              | 10     | -1      |
| 13  | statement       | 9      | 14      |
| 14  | statement_list  | 9      | -1      |
| 15  | split_stmt      | 13     | -1      |
| 16  | SPLIT           | 15     | 17      |
| 17  | BY              | 15     | 18      |
| 18  | STRING          | 15     | -1      |
| 19  | statement       | 14     | 20      |
| 20  | statement_list  | 14     | -1      |
| 21  | join_stmt       | 19     | -1      |
| 22  | JOIN            | 21     | 23      |
| 23  | WITH            | 21     | 24      |
| 24  | STRING          | 21     | -1      |

|    |                 |    |    |
|----|-----------------|----|----|
| 25 | statement       | 20 | 26 |
| 26 | statement_list  | 20 | -1 |
| 27 | assignment_stmt | 25 | -1 |
| 28 | ID              | 27 | 29 |
| 29 | ASSIGN          | 27 | 30 |
| 30 | STRING          | 27 | -1 |
| 31 | statement       | 26 | 32 |
| 32 | statement_list  | 26 | -1 |
| 33 | save_stmt       | 31 | -1 |
| 34 | SAVE            | 33 | 35 |
| 35 | ID              | 33 | -1 |

This is the output from yacc:

```
ASSIGN in_file = "data.csv"
LOAD in_file
SPLIT BY ","
JOIN WITH ";"
ASSIGN out_file = "converted_data.txt"
SAVE out_file
```

compare the 2 and validate that your code is ok

Thought for 44s >

Yep — your **parse-tree output is consistent with the YACC pretty-printed program**, given your grammar.

## 1) Same statement order

YACC prints 6 statements, in this order:

1. ASSIGN ...
2. LOAD ...
3. SPLIT ...
4. JOIN ...
5. ASSIGN ...
6. SAVE ...

Your tree has exactly the same sequence of statement types:

- assignment\_stmt at node 4
- load\_stmt at node 10

- `split_stmt` at node **15**
- `join_stmt` at node **21**
- `assignment_stmt` at node **27**
- `save_stmt` at node **33**

So the **structure and order match**.

## 2) Why you see many `statement_list` nodes

Your grammar is (right-)recursive like:

- `statement_list` → `statement statement_list | epsilon`

So the tree *must* look like a chain:

```
statement_list( statement , statement_list( statement , statement_list( ...  
)))
```

That's exactly what your output shows (nodes 1 → 3 → 9 → 14 → 20 → 26 → 32 ...).

Also: my code **doesn't add explicit epsilon nodes** to the tree, so the list ends "silently" (which is normal).

## 3) Why YACC shows `in_file`, "data.csv" but your tree shows ID, STRING

Your tree is built from **token categories** coming from the PIF mapping:

- ID
- STRING
- ASSIGN

YACC's output includes the **lexeme values** (`in_file`, "data.csv", etc.) because it prints the parsed program with semantic info.

So it's expected that:

- tree: ID ASSIGN STRING
- YACC: `in_file = "data.csv"`

## 4) One important detail: your grammar's `assignment_stmt`

Your `grammar.txt` says:

- `assignment_stmt` → ID ASSIGN STRING

And your tree matches that perfectly (nodes 5,6,7 under node 4; and nodes 28,29,30 under node 27).

So unless your *real* language is actually `ASSIGN ID = STRING` (different grammar / different lexer tokens), your parser is doing the right thing.

---

- ✓ Conclusion: **your C++ parser output is correct**, and it matches what YACC prints (tree form vs pretty-printed form + lexemes). If you want, I can also show the exact productions used (Requirement 1 style) for that same PIF input so you can compare derivations line-by-line.

