Question: Can you compare the yacc file I provided with the c source code you generated?
Answer: Based on the provided files, here is a comparison between ll1.c (a generic parser engine) and combinatorics.y (a grammar specification for a parser generator).

High-Level Summary

- **ll1.c** is a **manual implementation of a generic LL(1) parser**. It contains the raw C logic to read *any* compliant grammar file at runtime, calculate parsing sets (First/Follow), and parse input sequences using a stack.
- **combinatorics.y** is a **grammar specification file** intended for a tool like Yacc or Bison. It defines the syntax for a specific language (likely a Combinatorics DSL) and relies on the tool to generate the actual C parsing logic (typically LALR(1)).

Detailed Comparison

1. Purpose and Approach

- **ll1.c (The Engine):**
  - **Goal:** To function as a standalone parser generator and runner.
  - **Approach:** It explicitly implements the algorithms for building parsing tables. It includes functions like build_first, build_follow, and build_table to process grammar rules dynamically loaded from text files (seminar_grammar.txt or dsl_grammar.txt).
  - **Flexibility:** High. You can change the grammar without recompiling the C program, provided the new grammar is LL(1).
- **combinatorics.y (The Specification):**
  - **Goal:** To define the syntax rules for a specific "Combinatorics" language.
  - **Approach:** It uses a declarative syntax to define tokens (e.g., `%token INPUT OUTPUT`) and grammar rules (e.g., `program: statement_list`)[1]. It relies on an external tool (Yacc/Bison) to convert these rules into a C file (usually y.tab.c) that contains the parsing logic.
  - **Flexibility:** Low (Runtime). Changing the grammar requires modifying the .y file and recompiling the entire project.

2. Parsing Algorithm

- **ll1.c:** Uses **Top-Down Parsing (LL(1))**.
  - It starts with the start_symbol and attempts to expand non-terminals to match the input.
  - It detects conflicts manually: [CONFLICT] At (%s, %s): Rules %d and %d.
  - It uses a "Work Stack" and "Input Stack" to simulate the leftmost derivation.
- **combinatorics.y:** Implicitly uses **Bottom-Up Parsing (LALR(1))**.
  - Yacc/Bison generates parsers that shift input onto a stack and "reduce" it according to the rules defined (e.g., reducing `ID ASSIGN expression` to `assign_stmt`)[2].

- Precedence and associativity are handled declaratively (e.g., %left PLUS MINUS, %right FACTORIAL) rather than by manual lookahead logic[3].

3. Code Structure

Feature	ll1.c (Manual C)	combinatorics.y (Yacc/Bison)
Data Structures	Explicitly defines Grammar, Set, TableEntry, and ParseNode structs to manage data.	Abstracted away. The user only defines the rules; Yacc generates the state machines and tables internally.
Lexical Analysis	Simple space-delimited tokenizer using fscanf inside parse_sequence.	Relies on an external lexer. It declares extern int yylex(); and expects tokens like ID or CONST to be supplied by it[4].
Grammar Rules	stored in Production structs and loaded from external text files at runtime.	Written directly in the file using BNF-like syntax (e.g., if_stmt: IF condition THEN ...)[5].
Main Function	Contains the driver logic to load files, build tables, and run the parser.	Contains a simple driver that calls yyparse(), the function generated by Yacc[6666].

4. Error Handling

- **ll1.c:**
  - Manual detection. If the top of the stack doesn't match the input and no transition exists in the table, it prints Syntax Error at token: ... Expected: ... and lists valid tokens derived from the parse table.
- **combinatorics.y:**

- Standardized delegation. It defines a yyerror(const char *s) function which is called automatically by the generated parser when a syntax error occurs. It prints the line number (yylineno) and the error message[7].

Summary Table

Feature	ll1.c	combinatorics.y
Type	C Source Code (Implementation)	Yacc Grammar File (Definition)
Parsing Method	LL(1) (Top-Down)	LALR(1) (Bottom-Up)
Grammar Input	Runtime (Reads .txt files)	Compile-time (Hardcoded in .y)
Lexer	Integrated (Basic string split)	External (calls yylex)
Workflow	Compile C $\to$ Run Exe $\to$ Load Grammar	Yacc $\to$ Generate C $\to$ Compile C $\to$ Run Exe

Next Step: Would you like me to explain how to integrate a lexer (like Lex/Flex) with combinatorics.y to create a fully working compiler for this combinatorics language?