National University of Computer and Emerging Sciences
FAST School of Computing

CS–4031 Compiler Construction — Spring 2025

Assignment 04

From JSON to Relational CSV with Flex & Yacc

Released: 29 April 2025 • Due: **11 May 2025**

Why this?

Nested JSON is common, but SQL databases need flat tables. Your task is to build a tool that reads any valid JSON file and creates CSV tables in a relational manner. Use `Flex` for scanning, `Yacc/Bison` for parsing, and C for the rest (AST, schema, CSV).

Learning goals

- Write a JSON lexer and parser with Flex and Yacc.
- Build and use an Abstract Syntax Tree (AST).
- Map JSON objects to relational tables.
- Stream CSV output without large memory buffers.
- Write clean, memory-safe C code.

Quick specification

Run your tool as:

```
./json2relcsv <input.json> [--print-ast] [--out-dir DIR]
```

Requirements:

1. Handle any valid JSON up to 30 MiB.
2. Build an AST that lasts until the program ends.
3. Stream CSV rows using conversion rules.
4. Assign integer primary keys (`id`) and foreign keys.
5. Print AST to `stdout` if `--print-ast` is used.

6. Write one `.csv` file per table in `DIR` (default: current folder).
7. Report first error's line and column, exit non-zero on bad JSON.

Conversion rules

R1. **Object → table row**: Objects with same keys go in one table.

R2. **Array of objects → child table**: One row per element, with a foreign key to parent.

R3. **Array of scalars → junction table**: Columns `parent_id`, `index`, `value`.

R4. **Scalars → columns**: JSON `null` becomes empty.

R5. Every row gets an `id`. Foreign keys are `<parent>_id`.

R6. File name = table name + `.csv`; include header row.

Worked examples (simple → complex)

Each block shows the minimum CSV output.

Example 1 – Flat object

Input

```
{ "id": 1, "name": "Ali", "age": 19 }
```

people.csv

```
id,name,age
1,Ali,19
```

Example 2 – Array of scalars

Input

```
{
  "movie": "Inception",
  "genres": ["Action", "Sci-Fi", "Thriller"]
}
```

movies.csv

```
id,movie
1,Inception
```

genres.csv

```
movie_id,index,value
1,0,Action
1,1,Sci-Fi
1,2,Thriller
```

Example 3 – Array of objects

Input

```
{
  "orderId": 7,
  "items": [
    {"sku": "X1", "qty": 2},
    {"sku": "Y9", "qty": 1}
  ]
}
```

orders.csv

```
id,orderId
1,7
```

items.csv

```
order_id,seq,sku,qty
1,0,X1,2
1,1,Y9,1
```

Example 4 – Nested objects + reused shape

Input

```
{
  "postId": 101,
  "author": {"uid": "u1", "name": "Sara"},
  "comments": [
    {"uid": "u2", "text": "Nice!"},
    {"uid": "u3", "text": "+1"}
  ]
}
```

posts.csv

```
id,postId,author_id
1,101,1
```

users.csv

```
id,uid,name
1,u1,Sara
2,u2,
3,u3,
```

comments.csv

```
post_id,seq,user_id,text
1,0,2,"Nice!"
1,1,3,"+1"
```

Technical Details

How to build the JSON-to-CSV tool, step by step, from reading the JSON input to writing CSV files. Each step processes the data further to create relational tables.

Step 1: Tokenization

Use `Flex` in `scanner.l` to split the JSON file into tokens (like words or symbols). Tokens are sent to the parser to check the JSON structure.

What to tokenize:

- **Punctuation**: `{`, `}`, `[`, `]`, `:`, `,`.
- **Strings**: Text in quotes, e.g., `"hello"`. Support escapes like `\n` or `\u1234`.
- **Numbers**: Integers (e.g., `123`), decimals (e.g., `12.34`), or scientific (e.g., `1e-4`).
- **Keywords**: `true`, `false`, `null`.
- **Whitespace**: Ignore spaces, tabs, newlines, but track line/column for errors.

Tasks:

- Write `Flex` rules to match each token type.
- Save string or number values for the parser.
- Track line and column for error messages.
- Send tokens (e.g., `STRING`, `NUMBER`) to `Yacc`.

Step 2: Parsing

Use `Yacc/Bison` in `parser.y` to check if tokens form valid JSON and build an Abstract Syntax Tree (AST). The AST organizes the JSON data for creating tables.

What you're parsing:

- **Goal**: Verify JSON is correct (e.g., proper objects/arrays) and build an AST with the data structure.
- **Grammar (simplified)**:

  - `value`: Object, array, string, number, `true`, `false`, `null`.
  - `object`: `{` key-value pairs `}`, e.g., `{"key": 1}`.
  - `array`: `[` values `]`, e.g., `[1, 2]`.
  - `pair`: `"key": value`, e.g., `"orderId": 7`.

- The parser builds an **AST** directly, not a full parse tree, to make processing easier.

Parse Tree vs. AST:

- **Parse Tree**: Shows every grammar detail (e.g., `{`, `,`). Example for `{"key": 1}`:

```
object
 '{'
 pair
    STRING ("key")
    ':'
    NUMBER (1)
 '}'
```

- **AST**: Shows only data (objects, arrays, values). Same example:

```
OBJECT
 "key": NUMBER (1)
```

- You create the AST in Yacc, skipping the parse tree.

Tasks:

- Write `Yacc` rules for JSON grammar.
- Build AST nodes in C (e.g., `OBJECT`, `ARRAY`, `NUMBER`).
- Connect nodes to form the AST.
- Report syntax errors (e.g., missing `}`) with line/column.

Step 3: Semantic Analysis

The AST is a tree of JSON data (objects, arrays, numbers, strings) built by the parser. Use it to group data into tables based on the conversion rules.

What the AST does:

- Stores JSON structure (e.g., objects with keys, arrays with values).
- Groups objects with the same keys into one table (e.g., `{sku, qty}`).
- Links parent and child data for foreign keys (e.g., `order`$_id$).

Tasks:

- Define C structs for AST nodes (e.g., `OBJECT`, `ARRAY`, `STRING`, `NUMBER`).
- Walk the AST to find objects with the same keys and create table schemas.
- Add `id` for each row and `<parent>_id` for foreign keys.
- Prepare row data for CSV files.
- Print the AST to `stdout` if `--print-ast` is used (indented format).

Step 4: CSV Generation

Write the table data from the AST to CSV files, one per table, using the schemas from semantic analysis.

What to do:

- Create a `.csv` file for each table (e.g., `orders.csv`).
- Write a header row (e.g., `id,orderId`).
- Add rows with `id`, foreign keys, and values (quote strings, leave `null` empty).

Tasks:

- Open a file for each table in the output directory.
- Write headers and rows while walking the AST.
- Save files often to handle large inputs (up to 30 MiB).
- Check for file write errors.

Step 5: Error Handling

Catch errors at each step, report them clearly, and exit cleanly.

Types of errors:

- **Lexical**: Bad tokens, e.g., unclosed string `"hello`.
- **Syntax**: Wrong grammar, e.g., missing `,` in `{"key": 1 "key2": 2}`.
- **Other**: File or memory issues.

Tasks:

- Report lexical errors in `Flex` with line/column.
- Report syntax errors in `Yacc` with line/column.
- Check for file/memory errors in C code.
- Show errors like: `Error: Missing comma at line 2, column 5`.
- Free all memory (e.g., AST) before exiting.

Deliverables

Upload one ZIP file named `Rollno1-Rollno2-Section.zip` containing:

- `scanner.l`, `parser.y`, C files, and `Makefile (optional)`.
- `README.md` with build/run instructions and design notes.
- At least five JSON test cases and their expected CSV outputs.

Grading rubric (100 pts)

Area (pts)	**What we look for**
Lexing (15)	Correct tokenisation, Unicode escapes, line/column on errors.
Parsing (30)	Complete JSON grammar, AST integrity, no memory leaks.
Schema (15)	Table detection, PK/FK correctness, duplicate-shape reuse.
CSV output (10)	Headers, quoting, streaming for large files.
AST print (10)	Matches required indented format.
Error path (5)	First-error line:col, clean exit status.
Code quality (5)	Modularity, naming, inline docs, Valgrind clean.
Documentation (10)	Clear README and code comments.

Happy parsing & flattening!