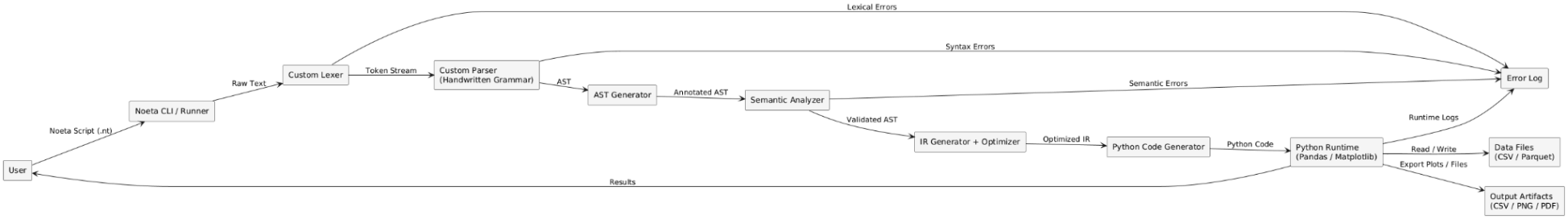
Dry Run:



---

1. Input Script

load "data.csv" as my_data
select my_data {colA, colB} as subset_data

---

2. Lexical Analysis

Line & Text	Tokens (type:lexeme)
1. `load "data.csv" as my_data`	`KEYWORD:loadSTRING_LITERAL:"data.csv"KEYWORD:asIDENTIFIER:my_data`

2.	
`select`	`KEYWORD:select`
`my_dat`	`IDENTIFIER:my_data`
`a`	`PUNCTUATION:{`
`{colA,`	`IDENTIFIER:colA`
`colB}`	`PUNCTUATION:,`
`as`	`IDENTIFIER:colB`
`subset`	`PUNCTUATION:}`
`_data`	`KEYWORD:as`
	`IDENTIFIER:subset_data`

---

3. Parsing → AST Generation

The **Custom Parser** (hand-written grammar) consumes the token streams and, following the BNF rules, builds AST nodes:

`load_stmt`

```
<load_stmt> ::= "load" STRING_LITERAL "as" IDENTIFIER
yields:

LoadNode(
 filepath="data.csv",
 alias="my_data"
)
```

1.

select_stmt

<select_stmt> ::= "select" IDENTIFIER <column_list_curly> "as" IDENTIFIER
yields:

```
SelectNode(
  source_alias="my_data",
  columns=["colA","colB"],
  alias="subset_data"
)
```

2.

So after parsing you have an AST like:

```
Program([
  LoadNode(filepath="data.csv", alias="my_data"),
  SelectNode(source_alias="my_data", columns=["colA","colB"], alias="subset_data")
])
```

---

4. Semantic Analysis

The **Semantic Analyzer** traverses the AST and uses the Symbol Table to enforce meaning and build schemas:

1. **Before any nodes**
   SymbolTable = { }

2. **Process `LoadNode`**
    - Check no existing `my_data` alias → OK
    - Infer schema by peeking into `"data.csv"` (e.g. columns `["colA","colB","colC",…]`)

Add

my_data → { schema: ["colA","colB","colC",…"], source: "load" }

    -
3. **Process `SelectNode`**
    - **Declaration-Before-Use**: verify `my_data` is in SymbolTable → OK
    - **Column Existence**: ensure `colA` and `colB` are in `my_data`'s schema → OK
    - **Alias Uniqueness**: `subset_data` not yet defined → OK

Add

subset_data → { schema: ["colA","colB"], source: "select" }

    -

After this stage, your symbol table contains exactly the two aliases, each with its derived schema.

Visualizing in the DFD

User → CLI : "load …\nselect …"
CLI --> Lexer : Raw Text
Lexer --> Parser : (LOAD, "data.csv", AS, my_data, …)
Parser --> AST : [LoadNode, SelectNode]
AST --> Sem : Annotated AST (with file paths, aliases, columns)
Sem --> IR : (would emit a validated, type-correct IR)
IR → Python Code Generator (Python code generated from IR)
Python Code Generator → Executed Code (Output shown to user: can be output on CLI or in image form that can be exported)

For these two statements, the pipeline stops cleanly with a validated AST and symbol table entries ready for the next stages (IR → code generation → runtime).
