Noeta - Design - Formal Grammar (BNF)

1. Introduction

This document provides the formal Backus-Naur Form (BNF) grammar for the Noeta language. Noeta is a domain-specific language (DSL) designed for data wrangling and exploratory analysis. Its syntax aims to be declarative, intuitive, and concise.

1.1. How to Read This BNF

- `::=` means "is defined as".
- `|` means "or" (alternative).
- `<non_terminal>`: Non-terminal symbols are enclosed in angle brackets. These are constructs that can be broken down further.
- `"terminal"` or `TERMINAL_TOKEN`: Terminal symbols are enclosed in double quotes (for keywords) or represented as named tokens (e.g., `IDENTIFIER`, `STRING_LITERAL`). These are the basic building blocks (lexemes) of the language.
- `[optional_part]`: Square brackets indicate that the enclosed part is optional.
- Items are concatenated to form sequences.
- Comments in the BNF are preceded by `//`.

2. Overall Program Structure

A Noeta program consists of a sequence of statements.

```
// A program is a list of one or more statements.
<program> ::= <statement_list>

// A statement list is either a single statement or a statement followed by more statements.
// This right-recursive definition is common and avoids left-recursion issues for some parser types.
<statement_list> ::= <statement>
                   | <statement> <statement_list>
```

Example:

A Noeta script can have one or multiple lines, each being a statement.

```
load "data/my_data.csv" as initial_data
info initial_data
```

3. Statements

A statement defines a single operation or action in Noeta.

```
<statement> ::= <load_stmt>         // Loads data from a file.
```

```
            | <select_stmt>       // Selects specific columns.
            | <filter_stmt>       // Filters rows based on a condition.
            | <sort_stmt>         // Sorts data by one or more columns.
            | <join_stmt>         // Joins two datasets.
            | <groupby_stmt>      // Groups data and applies aggregate functions.
            | <sample_stmt>       // Takes a sample from a dataset.
            | <dropna_stmt>       // Removes rows with missing values.
            | <fillna_stmt>       // Fills missing values.
            | <mutate_stmt>       // Creates new columns or modifies existing ones based on
expressions.
            | <apply_stmt>        // Applies a function to specified columns.
            | <describe_stmt>     // Generates descriptive statistics.
            | <summary_stmt>      // Provides a summary of the dataset.
            | <outliers_stmt>     // Detects outliers.
            | <quantile_stmt>     // Calculates quantiles for a column.
            | <normalize_stmt>    // Normalizes specified columns.
            | <binning_stmt>      // Bins a column into discrete intervals.
            | <rolling_stmt>      // Performs rolling window calculations.
            | <hypothesis_stmt>   // Performs a hypothesis test between two datasets or columns.
            | <boxplot_stmt>      // Generates a box plot.
            | <heatmap_stmt>      // Generates a heatmap.
            | <pairplot_stmt>     // Generates a pair plot.
            | <timeseries_stmt>   // Generates a time series plot.
            | <pie_stmt>          // Generates a pie chart.
            | <save_stmt>         // Saves a dataset to a file.
            | <export_plot_stmt>  // Exports a generated plot to an image file.
            | <info_stmt>         // Displays information about a dataset.
```

3.1. Data Manipulation Statements

3.1.1. `load` Statement

Loads data from an external file (e.g., CSV) into an alias.

```
// Loads a dataset from a file path (STRING_LITERAL) and assigns it an IDENTIFIER (alias).
// Example: load "data/employees.csv" as employees_data
<load_stmt> ::= "load" STRING_LITERAL "as" IDENTIFIER
```

Example:

```
load "input/sales_data.csv" as sales
```

3.1.2. `select` Statement

Selects a subset of columns from a dataset, creating a new dataset alias.

```
// Selects specified columns from an existing IDENTIFIER (dataset alias)
// and assigns the result to a new IDENTIFIER (new_alias).
// Example: select raw_data {columnA, columnB} as refined_data
```

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

Example:

```
select sales {product_id, quantity, sale_amount} as relevant_sales_info
```

3.1.3. filter Statement

Filters rows of a dataset based on a specified condition.

```
// Filters an IDENTIFIER (dataset alias) based on a <condition>.
// The result is stored in a new IDENTIFIER (new_alias).
// The condition is enclosed in square brackets, a common convention for query-like
sub-expressions.
// Example: filter orders [amount > 100] as large_orders
<filter_stmt> ::= "filter" IDENTIFIER "[" <condition> "]" "as" IDENTIFIER
```

Example:

```
filter relevant_sales_info [sale_amount > 500] as high_value_sales
```

3.1.4. sort Statement

Sorts a dataset by one or more columns, in ascending or descending order.

```
// Sorts an IDENTIFIER (dataset alias) based on a <column_list_sort>.
// The result is stored in a new IDENTIFIER (new_alias).
// 'by:' is used as a keyword to clearly demarcate the sorting columns.
// Example: sort products by: price desc as sorted_products
<sort_stmt> ::= "sort" IDENTIFIER "by" ":" <column_list_sort> "as" IDENTIFIER
```

Example:

```
sort high_value_sales by: sale_amount desc as top_sales
```

3.1.5. join Statement

Joins two datasets based on a common column.

```
// Joins a primary IDENTIFIER (alias1) with a secondary IDENTIFIER (alias2)
// on a specified IDENTIFIER (join_column).
// 'with:' and 'on:' keywords improve readability.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: join customers with: orders on: customer_id as customer_orders
<join_stmt> ::= "join" IDENTIFIER "with" ":" IDENTIFIER "on" ":" IDENTIFIER "as" IDENTIFIER
```

Example:

```
load "users.csv" as users
load "activity.csv" as activity
join users with: activity on: user_id as user_activity
```

3.1.6. `groupby` Statement

Groups data by specified columns and applies aggregation functions.

```
// Groups an IDENTIFIER (dataset alias) by a <column_list_curly> (grouping columns)
// and applies <agg_func_list> (aggregation functions).
// 'by:' and 'agg:' keywords separate the grouping and aggregation clauses.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: groupby sales_data by: {region} agg: {sum:revenue, avg:quantity} as
regional_summary
<groupby_stmt> ::= "groupby" IDENTIFIER "by" ":" <column_list_curly> "agg" ":" <agg_func_list>
"as" IDENTIFIER
```

Example:

```
groupby user_activity by: {user_id} agg: {count:page_views, sum:time_spent_minutes} as
user_engagement
```

3.1.7. `sample` Statement

Extracts a random or fixed-size sample from a dataset.

```
// Takes a sample from an IDENTIFIER (dataset alias).
// 'n:' specifies the NUMERIC_LITERAL (number of samples).
// The 'random' keyword is optional for random sampling.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: sample all_users n: 100 random as user_sample
// Example: sample all_users n: 50 as first_50_users (if 'random' implies specific behavior like
head)
<sample_stmt> ::= "sample" IDENTIFIER "n" ":" NUMERIC_LITERAL ["random"] "as"
IDENTIFIER
```

Example:

```
sample user_engagement n: 10 random as sample_engagement
```

3.1.8. `dropna` Statement

Removes rows with missing (NA/null) values.

```
// Removes rows with missing values from an IDENTIFIER (dataset alias).
```

```
// Optionally, 'columns:' can specify a <column_list_curly> to consider for NA values.
// If 'columns:' is omitted, all columns are considered.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: dropna raw_data as cleaned_data
// Example: dropna user_profiles columns: {email, phone} as validated_profiles
<dropna_stmt> ::= "dropna" IDENTIFIER ["columns" ":" <column_list_curly>] "as" IDENTIFIER
```

Example:

```
dropna sample_engagement columns: {time_spent_minutes} as complete_engagement_sample
```

3.1.9. fillna Statement

Fills missing (NA/null) values with a specified value.

```
// Fills missing values in an IDENTIFIER (dataset alias) with a <literal> (fill_value).
// 'value:' introduces the fill value.
// Optionally, 'columns:' can specify a <column_list_curly> where filling should occur.
// If 'columns:' is omitted, filling applies to all columns with NA.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: fillna sensor_readings value: 0 as readings_no_na
// Example: fillna survey_data value: "Not Provided" columns: {feedback} as survey_filled
<fillna_stmt> ::= "fillna" IDENTIFIER "value" ":" <literal> ["columns" ":" <column_list_curly>] "as"
IDENTIFIER
```

Example:

```
fillna user_activity value: 0 columns: {time_spent_minutes} as activity_filled_time
```

3.1.10. mutate Statement

Adds new columns or modifies existing ones using expressions.

```
// Creates or modifies columns in an IDENTIFIER (dataset alias) based on a <mutation_list>.
// Each mutation defines a new column and its expression.
// The result is stored in a new IDENTIFIER (new_alias).
// Example: mutate sales {total_price: "quantity * unit_price", discount_amount: "total_price * 0.1"}
as final_sales
<mutate_stmt> ::= "mutate" IDENTIFIER <mutation_list> "as" IDENTIFIER
```

Example:

```
mutate user_engagement {engagement_score: "page_views * 0.5 + time_spent_minutes * 0.5"}
as scored_engagement
```

3.1.11. `apply` Statement

Applies a user-defined or built-in function to specified columns. The function is provided as a string.

```
// Applies a 'function' (provided as a STRING_LITERAL expression) to specified 'columns'
// (a <column_list_curly>) of an IDENTIFIER (dataset alias).
// The result is stored in a new IDENTIFIER (new_alias).
// This is powerful for custom transformations.
// Example: apply measurements columns: {value1, value2} function: "x * x" as
squared_measurements
// (where 'x' is a placeholder for the column value)
<apply_stmt> ::= "apply" IDENTIFIER "columns" ":" <column_list_curly> "function" ":"
STRING_LITERAL "as" IDENTIFIER
```

Example:

```
apply scored_engagement columns: {engagement_score} function: "log(x + 1)" as
log_engagement_score
```

3.2. Statistics & Analysis Statements

3.2.1. `describe` Statement

Generates descriptive statistics for a dataset or specified columns.

```
// Generates descriptive statistics for an IDENTIFIER (dataset alias).
// Optionally, 'columns:' can specify a <column_list_curly> to describe.
// This command typically outputs results directly and doesn't create a new alias.
// Example: describe financial_data
// Example: describe user_data columns: {age, income}
<describe_stmt> ::= "describe" IDENTIFIER ["columns" ":" <column_list_curly>]
```

Example:

```
describe scored_engagement columns: {engagement_score, page_views}
```

3.2.2. `summary` Statement

Provides a concise summary of the dataset.

```
// Provides a summary for an IDENTIFIER (dataset alias).
// Similar to describe, but might offer different or more concise information.
// Typically outputs results directly.
// Example: summary sales_transactions
<summary_stmt> ::= "summary" IDENTIFIER
```

Example:

```
summary scored_engagement
```

3.2.3. `outliers` Statement

Detects outliers in specified columns using a chosen method.

```
// Detects outliers in an IDENTIFIER (dataset alias).
// 'method:' specifies the IDENTIFIER (outlier detection method, e.g., "iqr", "zscore").
// 'columns:' specifies the <column_list_curly> to check for outliers. This is typically required.
// This command might output results or flag rows, rather than creating a new dataset alias by default.
// (The original list didn't specify 'as new_alias', so assuming it reports/highlights outliers).
// Example: outliers sensor_data method: iqr columns: {temperature}
<outliers_stmt> ::= "outliers" IDENTIFIER "method" ":" IDENTIFIER "columns" ":" <column_list_curly>
```

Example:

```
outliers scored_engagement method: iqr columns: {engagement_score}
```

3.2.4. `quantile` Statement

Calculates a specific quantile for a column.

```
// Calculates a quantile for a 'column' (IDENTIFIER) in a dataset (IDENTIFIER).
// 'q:' specifies the NUMERIC_LITERAL (quantile value, e.g., 0.25, 0.5, 0.75).
// Typically outputs the quantile value.
// Example: quantile product_prices column: price q: 0.9
<quantile_stmt> ::= "quantile" IDENTIFIER "column" ":" IDENTIFIER "q" ":" NUMERIC_LITERAL
```

Example:

```
quantile scored_engagement column: engagement_score q: 0.95
```

3.2.5. `normalize` Statement

Normalizes specified columns using a chosen method (e.g., z-score).

```
// Normalizes 'columns' (a <column_list_curly>) in an IDENTIFIER (dataset alias)
// using a specified 'method' (IDENTIFIER, e.g., "zscore", "minmax").
// The result is stored in a new IDENTIFIER (new_alias).
// Example: normalize feature_set columns: {feature1, feature2} method: zscore as normalized_features
<normalize_stmt> ::= "normalize" IDENTIFIER "columns" ":" <column_list_curly> "method" ":" IDENTIFIER "as" IDENTIFIER
```

Example:

```
normalize scored_engagement columns: {page_views, time_spent_minutes} method: zscore as normalized_engagement_metrics
```

3.2.6. `binning` Statement

Divides a continuous column into a specified number of bins.

```
// Bins a 'column' (IDENTIFIER) in an IDENTIFIER (dataset alias)
// into a NUMERIC_LITERAL (number of bins).
// The result, typically the original dataset with an added binned column, is stored in a new IDENTIFIER (new_alias).
// Example: binning customer_data column: age bins: 5 as binned_customer_age
<binning_stmt> ::= "binning" IDENTIFIER "column" ":" IDENTIFIER "bins" ":" NUMERIC_LITERAL "as" IDENTIFIER
```

Example:

```
binning scored_engagement column: engagement_score bins: 4 as engagement_bins
```

3.2.7. `rolling` Statement

Performs rolling window calculations (e.g., rolling mean) on a column.

```
// Performs a rolling window calculation on a 'column' (IDENTIFIER) in an IDENTIFIER (dataset alias).
// 'window:' specifies the NUMERIC_LITERAL (window size).
// 'function:' specifies the IDENTIFIER (aggregation function, e.g., "mean", "sum").
// The result is stored in a new IDENTIFIER (new_alias).
// Example: rolling stock_prices column: price window: 7 function: mean as 7day_avg_price
<rolling_stmt> ::= "rolling" IDENTIFIER "column" ":" IDENTIFIER "window" ":" NUMERIC_LITERAL "function" ":" IDENTIFIER "as" IDENTIFIER
```

Example:

```
load "daily_visits.csv" as daily_visits
rolling daily_visits column: visits window: 7 function: mean as weekly_avg_visits
```

3.2.8. `hypothesis` Statement

Performs a statistical hypothesis test.

```
// Performs a hypothesis test between an IDENTIFIER (alias1) and optionally another IDENTIFIER (alias2, indicated by 'vs:').
```

// (The original syntax `alias1 vs: alias2 columns: {col1} test: ttest` implies two datasets or two sets of columns).
// Let's refine based on `alias1 vs: alias2 columns: {col1} test: ttest`.
// This suggests comparing columns from two different datasets, or two columns within the same dataset.
// For simplicity, let's assume it's comparing specified columns from alias1 against columns from alias2, or two groups within alias1.
// 'columns:' specifies the <column_list_curly> involved in the test.
// 'test:' specifies the IDENTIFIER (type of test, e.g., "ttest", "chi2").
// This command typically outputs test results.
// Example: hypothesis groupA_data vs: groupB_data columns: {score} test: ttest
// Example: hypothesis treatment_data columns: {before_metric, after_metric} test: paired_ttest (if 'vs:' is optional or context implies within-dataset)
// The provided syntax is `hypothesis alias1 vs: alias2 columns: {col1} test: ttest`.
<hypothesis_stmt> ::= "hypothesis" IDENTIFIER "vs" ":" IDENTIFIER "columns" ":" <column_list_curly> "test" ":" IDENTIFIER

Example:

```
load "group_a_scores.csv" as group_a
load "group_b_scores.csv" as group_b
hypothesis group_a vs: group_b columns: {score} test: ttest_ind
```

3.3. Advanced Visualization Statements

These statements generate plots. They typically display the plot or prepare it for export, and may not create new dataset aliases.

3.3.1. `boxplot` Statement

// Generates a box plot for the specified 'columns' (a <column_list_curly>)
// from an IDENTIFIER (dataset alias).
// Example: boxplot user_metrics columns: {response_time, error_rate}
<boxplot_stmt> ::= "boxplot" IDENTIFIER "columns" ":" <column_list_curly>

Example:

```
boxplot scored_engagement columns: {engagement_score, page_views}
```

3.3.2. `heatmap` Statement

// Generates a heatmap, often from a correlation matrix or specified 'columns' (a <column_list_curly>)
// of an IDENTIFIER (dataset alias).
// Example: heatmap correlation_matrix columns: {colA, colB, colC}
// (Assuming columns are relevant for heatmap, e.g., for correlation or pivot table)
<heatmap_stmt> ::= "heatmap" IDENTIFIER "columns" ":" <column_list_curly>

Example:

```
// Assuming 'normalized_engagement_metrics' has numeric columns suitable for a correlation heatmap
// Often, a heatmap would be generated on a pre-calculated correlation matrix.
// For simplicity, if it's direct columns, they should be numeric.
normalize scored_engagement columns: {page_views, time_spent_minutes} method: zscore as metrics_for_heatmap
heatmap metrics_for_heatmap columns: {page_views, time_spent_minutes} // This might imply correlation between these
```

3.3.3. pairplot Statement

```
// Generates a pair plot (matrix of scatterplots) for the specified 'columns' (a <column_list_curly>)
// of an IDENTIFIER (dataset alias). Useful for visualizing relationships between multiple variables.
// Example: pairplot feature_data columns: {feature1, feature2, feature3}
<pairplot_stmt> ::= "pairplot" IDENTIFIER "columns" ":" <column_list_curly>
```

Example:

```
pairplot metrics_for_heatmap columns: {page_views, time_spent_minutes}
```

3.3.4. timeseries Statement

```
// Generates a time series plot.
// 'x:' specifies the IDENTIFIER (date/time column).
// 'y:' specifies the IDENTIFIER (value column).
// From an IDENTIFIER (dataset alias).
// Example: timeseries stock_data x: date y: closing_price
<timeseries_stmt> ::= "timeseries" IDENTIFIER "x" ":" IDENTIFIER "y" ":" IDENTIFIER
```

Example:

```
load "website_traffic.csv" as traffic_data // Assuming traffic_data has 'date_col' and 'visits_col'
timeseries traffic_data x: date_col y: visits_col
```

3.3.5. pie Statement

```
// Generates a pie chart.
// 'values:' specifies the IDENTIFIER (column containing numerical values for pie slices).
// 'labels:' specifies the IDENTIFIER (column containing labels for pie slices).
// From an IDENTIFIER (dataset alias).
// Example: pie category_summary values: count labels: category_name
<pie_stmt> ::= "pie" IDENTIFIER "values" ":" IDENTIFIER "labels" ":" IDENTIFIER
```

Example:

```
groupby user_activity by: {country} agg: {count:user_id} as users_by_country
pie users_by_country values: count_user_id labels: country
```

3.4. File Operations & Export Statements

3.4.1. save Statement

Saves a dataset to a file.

```
// Saves an IDENTIFIER (dataset alias) to a file specified by 'to:' STRING_LITERAL (file path).
// Optionally, 'format:' can specify the IDENTIFIER (file format, e.g., "csv", "parquet").
// If format is omitted, a default (e.g., CSV) might be assumed.
// Example: save processed_data to: "output/final_data.csv" format: csv
<save_stmt> ::= "save" IDENTIFIER "to" ":" STRING_LITERAL ["format" ":" IDENTIFIER]
```

Example:

```
save scored_engagement to: "output/scored_user_engagement.csv" format: csv
```

3.4.2. export_plot Statement

Exports the last generated plot to an image file.

```
// Exports the current or last generated plot.
// 'filename:' specifies the STRING_LITERAL (output file path).
// Optionally, 'width:' and 'height:' (NUMERIC_LITERALs) can specify plot dimensions.
// Example: export_plot filename: "charts/sales_trend.png" width: 800 height: 600
<export_plot_stmt> ::= "export_plot" "filename" ":" STRING_LITERAL ["width" ":"
NUMERIC_LITERAL] ["height" ":" NUMERIC_LITERAL]
```

Example:

```
boxplot scored_engagement columns: {engagement_score}
export_plot filename: "output/engagement_boxplot.png" width: 600 height: 400
```

3.5. Metadata Statements

3.5.1. info Statement

Displays information about a dataset (schema, types, memory usage, etc.).

```
// Displays information about an IDENTIFIER (dataset alias).
// This command typically outputs results directly.
// Example: info loaded_dataset
<info_stmt> ::= "info" IDENTIFIER
```

Example:

```
info scored_engagement
```

4. Helper Non-Terminal Rules

These rules define common structures used in multiple statements.

4.1. <condition>

Defines a condition for filtering, typically a simple comparison.

For simplicity and to avoid ambiguity with complex expressions in the initial version, conditions are kept straightforward. More complex boolean logic (AND/OR, parentheses) can be added later.

```
// A condition compares an IDENTIFIER (column) with a <literal> or another IDENTIFIER (column).
// Example: amount > 1000
// Example: columnA == columnB
<condition> ::= IDENTIFIER <operator> <literal>
              | IDENTIFIER <operator> IDENTIFIER
```

Examples used in filter:

- [price < 50]
- [status == "active"]
- [actual_value > predicted_value]

4.2. <operator>

Relational operators used in conditions.

```
// Standard comparison operators.
<operator> ::= "==" | "!=" | "<" | ">" | "<=" | ">="
```

4.3. <column_list_sort>

A list of columns for sorting, where each column can optionally have a sort direction.

```
// A list of one or more columns for sorting. Each column is an IDENTIFIER.
// 'desc' is an optional keyword for descending sort; ascending is default.
// This is a right-recursive definition.
// Example: columnA desc, columnB
<column_list_sort> ::= <sort_column_spec>
                     | <sort_column_spec> "," <column_list_sort>

<sort_column_spec> ::= IDENTIFIER ["desc"]
```

Examples used in `sort`:

- `name`
- `age desc`
- `category, price desc`

4.4. `<column_list_curly>`

A list of column names, enclosed in curly braces.

```
// A list of column names enclosed in curly braces, e.g., {col1, col2, col3}.
// Used where multiple columns are selected or specified.
<column_list_curly> ::= "{" <column_names> "}"

// <column_names> is one or more IDENTIFIERs, comma-separated.
// This is a right-recursive definition.
<column_names> ::= IDENTIFIER
                 | IDENTIFIER "," <column_names>
```

Examples used in `select`, `groupby`, `dropna`, etc.:

- `{id, name, email}`
- `{product_category}`

4.5. `<agg_func_list>`

A list of aggregation functions to apply, typically in `groupby`.

```
// A list of aggregation functions enclosed in curly braces.
// Example: {sum:total_sales, avg:price}
<agg_func_list> ::= "{" <agg_functions> "}"

// <agg_functions> is one or more <agg_function> definitions, comma-separated.
// This is a right-recursive definition.
<agg_functions> ::= <agg_function>
                  | <agg_function> "," <agg_functions>

// An <agg_function> specifies the aggregation type (e.g., "sum", "avg", "count")
// and the column to aggregate. The result column name is often implied or can be
// aliased by the system (e.g. sum_of_col3 or col3_sum).
// The syntax `sum:col3` means "apply sum to col3".
// The output column name might be `sum_col3` or similar by convention.
<agg_function> ::= IDENTIFIER ":" IDENTIFIER  // e.g., sum:revenue
(function:column_to_aggregate)
```

Examples used in `groupby`:

- `{count:user_id}`
- `{sum:amount, avg:quantity}`

4.6. <mutation_list>

A list of mutations for creating or transforming columns.

```
// A list of column mutations enclosed in curly braces.
// Example: {new_col: "colA + colB", another_col: "colC * 2"}
<mutation_list> ::= "{" <mutations> "}"

// <mutations> is one or more <mutation> definitions, comma-separated.
// This is a right-recursive definition.
<mutations> ::= <mutation>
              | <mutation> "," <mutations>

// A <mutation> defines a new column (IDENTIFIER) and its value derived from
// an expression (STRING_LITERAL). The expression string will be parsed and evaluated by the backend.
// Example: derived_value: "source_column / 100"
<mutation> ::= IDENTIFIER ":" STRING_LITERAL
```

Examples used in mutate**:**

- {price_per_unit: "total_cost / quantity"}
- {full_name: "first_name + ' ' + last_name", age_next_year: "age + 1"}

4.7. <literal>

Represents literal values like strings or numbers.

```
// A literal value can be a string or a number.
// Boolean literals (true/false) could be added if needed by conditions or functions.
<literal> ::= STRING_LITERAL
            | NUMERIC_LITERAL
            // | BOOLEAN_LITERAL // Future consideration
```

Examples:

- "completed"
- 100
- 3.14159

5. Terminal Symbols (Lexemes)

These are the basic "words" recognized by the lexical analyzer (scanner) and used by the parser. Their specific patterns (like regular expressions) are defined in the lexical analysis phase.

- IDENTIFIER: A name for datasets (aliases) or columns.
    - Typically [a-zA-Z_][a-zA-Z0-9_]*
    - Examples: my_data, column1, total_sales, _temp

- `STRING_LITERAL`: Text enclosed in double quotes.
    - Typically `\"[^"]*\"` (allowing for escaped quotes within if supported)
    - Examples: `"data/file.csv"`, `"active_users"`, `"x > 10"` (in mutate/apply)
- `NUMERIC_LITERAL`: Integer or floating-point numbers.
    - Typically `[0-9]+(\.[0-9]+)?`
    - Examples: `100`, `0.75`, `12.345`
- **Keywords**: Reserved words with special meaning. These are listed explicitly in quotes in the BNF rules (e.g., `"load"`, `"filter"`, `"as"`).
    - Examples: `load`, `as`, `filter`, `by`, `agg`, `columns`, `value`, `n`, `random`, `with`, `on`, `method`, `q`, `bins`, `window`, `function`, `vs`, `test`, `x`, `y`, `values`, `labels`, `to`, `format`, `filename`, `width`, `height`, `desc`.
- **Punctuation**: Characters like `:`, `[`, `]`, `{`, `}`, `,`. These are also listed explicitly in quotes.

This BNF grammar provides a solid foundation for the Noeta language parser. It is designed to be extensible for future features while maintaining clarity for current directives.