Noeta - Design - Abstract Syntax Tree (AST)

1. Introduction

The Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in the Noeta language. After the Noeta parser successfully validates the syntax of a script against the formal grammar (BNF), it constructs an AST. This tree hierarchically organizes the program's constructs, making it easier for subsequent compiler phases, such as semantic analysis and code generation, to understand and process the code.

Each node in the AST represents a construct in the Noeta language, like a statement, an expression, an identifier, or a literal value. The structure of the AST mirrors the structure of the Noeta program, but abstracts away some of the finer syntactic details (like punctuation) that are important for parsing but not for understanding the program's meaning.

2. General Node Structure

While not strictly defining a class hierarchy here, all AST nodes can be thought of as having a "type" (e.g., `LoadNode`, `FilterNode`) and a set of attributes that store the relevant information for that construct. For instance, many statement nodes will have a `source_alias` (the dataset they operate on) and a `new_alias` (the dataset they produce).

Line numbers and column information from the source code can also be associated with AST nodes to provide better error reporting. For simplicity in this document, these are not explicitly listed for every node but are a common practice.

3. Core Node Types

These are fundamental nodes that might be part of many other, more complex nodes.

3.1. `ProgramNode`

- **Description**: The root node of the AST, representing an entire Noeta program or script.
- **Attributes**:
    - `statements` (List of StatementNode): An ordered list of the statements that make up the program.

3.2. `IdentifierNode`

- **Description**: Represents an identifier, such as a dataset alias or a column name.
- **Attributes**:

- `name` (String): The string value of the identifier (e.g., "my_data", "column_A").

3.3. `LiteralNode`

- **Description**: Represents a literal value in the code.
- **Attributes**:
  - `value`: The actual value of the literal (can be String, Number).
  - `type` (String): The type of the literal (e.g., "STRING", "NUMERIC").

4. Statement Node Types

These nodes represent the various statements defined in the Noeta BNF grammar.

4.1. Data Manipulation Statement Nodes

4.1.1. `LoadNode`

- **Description**: Represents a `load` statement, used to load data from an external file.
- **Attributes**:
  - `file_path` (String): The path to the data file (e.g., "data/sales.csv").
  - `alias` (String): The alias assigned to the loaded dataset (e.g., "sales_data").

4.1.2. `SelectNode`

- **Description**: Represents a `select` statement, used to choose specific columns from a dataset.
- **Attributes**:
  - `source_alias` (String): The alias of the dataset to select from.
  - `columns` (List of String): The names of the columns to select.
  - `new_alias` (String): The alias assigned to the resulting dataset.

4.1.3. `FilterNode`

- **Description**: Represents a `filter` statement, used to filter rows based on a condition.
- **Attributes**:
  - `source_alias` (String): The alias of the dataset to filter.
  - `condition` (ConditionNode): The condition used for filtering.
  - `new_alias` (String): The alias assigned to the filtered dataset.

4.1.4. `SortNode`

- **Description**: Represents a `sort` statement, used to sort a dataset.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to sort.
    - `sort_specs` (List of SortSpecNode): A list of specifications for sorting (column name and direction).
    - `new_alias` (String): The alias assigned to the sorted dataset.

4.1.5. `JoinNode`

- **Description**: Represents a `join` statement, used to combine two datasets.
- **Attributes**:
    - `alias1` (String): The alias of the first (left) dataset.
    - `alias2` (String): The alias of the second (right) dataset.
    - `join_column` (String): The name of the column to join on.
    - `new_alias` (String): The alias assigned to the joined dataset.

4.1.6. `GroupByNode`

- **Description**: Represents a `groupby` statement, used for grouping and aggregation.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to group.
    - `group_by_columns` (List of String): A list of column names to group by.
    - `aggregations` (List of AggregationNode): A list of aggregation functions to apply.
    - `new_alias` (String): The alias assigned to the grouped and aggregated dataset.

4.1.7. `SampleNode`

- **Description**: Represents a `sample` statement, used to take a sample from a dataset.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to sample from.
    - `sample_size` (Numeric): The number of samples to take.
    - `is_random` (Boolean): True if random sampling is specified, false otherwise.
    - `new_alias` (String): The alias assigned to the sampled dataset.

4.1.8. `DropNANode`

- **Description**: Represents a `dropna` statement, used to remove rows with missing values.
- **Attributes**:

    - `source_alias` (String): The alias of the dataset to process.
    - `columns` (Optional List of String): Specific columns to consider for dropping NA. If None, all columns are considered.
    - `new_alias` (String): The alias assigned to the dataset after NA removal.

4.1.9. `FillNANode`

- **Description**: Represents a `fillna` statement, used to fill missing values.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to process.
    - `fill_value` (LiteralNode): The literal value to use for filling NA.
    - `columns` (Optional List of String): Specific columns to fill. If None, applies to all applicable columns.
    - `new_alias` (String): The alias assigned to the dataset after filling NA.

4.1.10. `MutateNode`

- **Description**: Represents a `mutate` statement, used to create or modify columns using expressions.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to mutate.
    - `mutations` (List of MutationSpecNode): A list of mutation specifications (new column name and its expression).
    - `new_alias` (String): The alias assigned to the mutated dataset.

4.1.11. `ApplyNode`

- **Description**: Represents an `apply` statement, used to apply a function to columns.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `columns` (List of String): The columns to apply the function to.
    - `function_expression` (String): The function (as a string expression, e.g., "x**2").
    - `new_alias` (String): The alias for the dataset with applied function results.

4.2. Statistics & Analysis Statement Nodes

4.2.1. `DescribeNode`

- **Description**: Represents a `describe` statement, for generating descriptive statistics.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to describe.

    - `columns` (Optional List of String): Specific columns to describe. If None, describes all relevant columns.

4.2.2. `SummaryNode`

- **Description**: Represents a `summary` statement, for providing a dataset summary.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to summarize.

4.2.3. `OutliersNode`

- **Description**: Represents an `outliers` statement, for detecting outliers.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `method` (String): The outlier detection method (e.g., "iqr").
    - `columns` (List of String): The columns to check for outliers.

4.2.4. `QuantileNode`

- **Description**: Represents a `quantile` statement, for calculating quantiles.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `column` (String): The column for which to calculate the quantile.
    - `quantile_value` (Numeric): The quantile to calculate (e.g., 0.75).

4.2.5. `NormalizeNode`

- **Description**: Represents a `normalize` statement, for normalizing data.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `columns` (List of String): The columns to normalize.
    - `method` (String): The normalization method (e.g., "zscore").
    - `new_alias` (String): The alias for the dataset with normalized columns.

4.2.6. `BinningNode`

- **Description**: Represents a `binning` statement, for discretizing a column.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `column` (String): The column to bin.
    - `num_bins` (Numeric): The number of bins to create.
    - `new_alias` (String): The alias for the dataset with the binned column.

4.2.7. `RollingNode`

- **Description**: Represents a `rolling` statement, for rolling window calculations.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `column` (String): The column to perform rolling calculation on.
    - `window_size` (Numeric): The size of the rolling window.
    - `function_name` (String): The aggregation function for the window (e.g., "mean", "sum").
    - `new_alias` (String): The alias for the dataset with the rolling calculation result.

4.2.8. `HypothesisNode`

- **Description**: Represents a `hypothesis` statement, for statistical hypothesis testing.
- **Attributes**:
    - `alias1` (String): The alias of the first dataset/group.
    - `alias2` (String): The alias of the second dataset/group.
    - `columns` (List of String): The columns involved in the test.
    - `test_type` (String): The type of hypothesis test (e.g., "ttest").

4.3. Advanced Visualization Statement Nodes

These nodes typically trigger a plotting action rather than creating a new dataset alias.

4.3.1. `BoxPlotNode`

- **Description**: Represents a `boxplot` statement.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to plot.
    - `columns` (List of String): The columns to include in the box plot.

4.3.2. `HeatmapNode`

- **Description**: Represents a `heatmap` statement.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to plot.
    - `columns` (List of String): The columns to include in the heatmap (often numeric for correlation or pivot-like data).

4.3.3. `PairPlotNode`

- **Description**: Represents a `pairplot` statement.
- **Attributes**:

    - `source_alias` (String): The alias of the dataset to plot.
    - `columns` (List of String): The columns to include in the pair plot.

4.3.4. `TimeSeriesPlotNode`

- **Description**: Represents a `timeseries` statement.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `x_column` (String): The column for the x-axis (typically time).
    - `y_column` (String): The column for the y-axis (values).

4.3.5. `PieChartNode`

- **Description**: Represents a `pie` statement.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset.
    - `values_column` (String): The column containing numerical values for slices.
    - `labels_column` (String): The column containing labels for slices.

4.4. File Operations & Export Statement Nodes

4.4.1. `SaveNode`

- **Description**: Represents a `save` statement, for saving a dataset to a file.
- **Attributes**:
    - `source_alias` (String): The alias of the dataset to save.
    - `file_path` (String): The destination file path.
    - `format_type` (Optional String): The file format (e.g., "csv"). Defaults if not provided.

4.4.2. `ExportPlotNode`

- **Description**: Represents an `export_plot` statement, for saving a generated plot.
- **Attributes**:
    - `file_name` (String): The name of the file to save the plot to.
    - `width` (Optional Numeric): The width of the exported image.
    - `height` (Optional Numeric): The height of the exported image.

4.5. Metadata Statement Nodes

4.5.1. `InfoNode`

- **Description**: Represents an `info` statement, for displaying dataset information.
- **Attributes**:

- `source_alias` (String): The alias of the dataset to get information about.

5. Helper and Expression Node Types

These nodes represent smaller syntactic structures used within statements.

5.1. `ConditionNode`

- **Description**: Represents a condition used in `filter` statements.
- **Attributes**:
  - `left_operand` (IdentifierNode or LiteralNode): The left side of the comparison.
  - `operator` (String): The comparison operator (e.g., "==", ">", "<=").
  - `right_operand` (IdentifierNode or LiteralNode): The right side of the comparison.
  - *(Note: For simplicity, operands are shown as direct values/names. They could be more structured nodes like `ColumnIdentifierNode` or `LiteralValueNode` if deeper expression parsing is implemented.)*

5.2. `SortSpecNode`

- **Description**: Represents a single column sort specification used in `sort` statements.
- **Attributes**:
  - `column_name` (String): The name of the column to sort by.
  - `direction` (String): The sort direction ("ASC" for ascending, "DESC" for descending).

5.3. `AggregationNode`

- **Description**: Represents an aggregation operation used in `groupby` statements.
- **Attributes**:
  - `function_name` (String): The name of the aggregation function (e.g., "sum", "avg", "count").
  - `column_name` (String): The name of the column to aggregate.
  - *(The output column name for this aggregation is often implicitly `function_name_column_name` or handled by the code generator).*

5.4. `MutationSpecNode`

- **Description**: Represents a single column mutation in a `mutate` statement.
- **Attributes**:
  - `new_column_name` (String): The name of the new or modified column.

- `expression_string` (String): The string representation of the expression to compute the column's value.

6. Example AST Diagrams

Let's illustrate with a couple of simple Noeta code snippets.

Example 1: `load` and `info`

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

AST Diagram (simplified text representation):

```
ProgramNode
  statements:
    - LoadNode
        file_path: "data/input.csv"
        alias: "my_data"
    - InfoNode
        source_alias: "my_data"
```

Example 2: `filter` and `save`

```
load "products.csv" as prods
filter prods [price > 100] as expensive_prods
save expensive_prods to: "output/filtered.csv"
```

AST Diagram (simplified text representation):

```
ProgramNode
  statements:
    - LoadNode
        file_path: "products.csv"
        alias: "prods"
    - FilterNode
        source_alias: "prods"
        condition: ConditionNode
                    left_operand: IdentifierNode {name: "price"}
                    operator: ">"
                    right_operand: LiteralNode {value: 100, type: "NUMERIC"}
        new_alias: "expensive_prods"
    - SaveNode
        source_alias: "expensive_prods"
        file_path: "output/filtered.csv"
        format_type: null (or default like "csv")
```

Example 3: groupby

```
load "sales.csv" as sales_records
groupby sales_records by: {category} agg: {sum:amount, count:id} as category_summary
```

AST Diagram (simplified text representation):

```
ProgramNode
  statements:
    - LoadNode
        file_path: "sales.csv"
        alias: "sales_records"
    - GroupByNode
        source_alias: "sales_records"
        group_by_columns: ["category"]
        aggregations:
          - AggregationNode
              function_name: "sum"
              column_name: "amount"
          - AggregationNode
              function_name: "count"
              column_name: "id"
        new_alias: "category_summary"
```

This AST design provides a structured way to represent Noeta programs, facilitating further processing by the compiler. The node types and their attributes are derived directly from the Noeta language's BNF grammar.