

# Raccoons : DSL for CSV Data Processing

Team members:-

**Mane Pooja Vinod:-** CS22BTECH11035 (Project Manager)

**Deva Suvedh:-** CS22BTECH11016 (System Architect)

**Medikonda Sreekar:-** CS22BTECH11037 (System Architect)

**Bolla Nehasree:-** CS22BTECH11012 (Language Guru)

**C Sree Vyshnavi:-** CS24RESCH11010 (Language Guru)

**Surbhi:-** CS22BTECH11057 (System Integrator)

**Simhadri Nayak Ramavath:-** CS22BTECH11049 (Tester)



# LEXICAL ANALYZER

The lexical analyzer scans the input **.rc** files and categorizes sequences of characters into tokens based on predefined patterns. Here's a breakdown of its key components:

## 1. Tokens:-

There are several tokens like keywords, operators, punctuation and special types of tokens.

- **Keywords:-** **for, while, if, else, int, float, string, bool, true, false, input, output, print, continue, break.**
- **Operators:-** **+=, -=, \*=, /=, %=**, relational operators like **<=, >=, ==, !=, +, -, \*, \, %, =, <, >, &, ^, |** and logical operators like **&&, ||, ++, --, >>, <<.**
- **Punctuation:** **;, ,, :, (, ), [, ], {, }, ..., ", '.**
- **Special Types:** **EXPONENTIAL, PERCENTAGE, INTEGER, FLOAT, STRING, CSVFILE**

## 2. Regular Expressions for Token Matching:-

The `%%` block contains regular expressions that match various components of the input source code and map them to specific token types. For example:

- `"for"`:- Matches the keyword "for" and returns the token **FOR**.
- `[0-9]+(\.[0-9]+)?%` :- Matches percentage numbers and returns the **PERCENTAGE** token.
- `"\"([^\\"\\]|\\.)*\""` :- Matches string literals enclosed in double quotes and returns the **STRING** token.
- `[A-Za-z_]+\.``csv` :- Matches filenames with the `.csv` extension and returns the **CSVFILE** token.

## 3. File I/O for Output:-

- The lexer reads input from a file specified by the user and writes the tokenized output to an output file named **output.txt**.
- It uses **yyin** for reading from the input file and writes the lexed tokens to **outfile**.

## 4. Multi-line and Single-line Comment Handling:-

- **Single-line comments**: Lines starting with `//` are ignored by the lexer.
- **Multi-line comments**: Handled by matching the **start** `(/*)` and **end** `(*/)` sequences and skipping everything in between.

## 5. Exponential Numbers, Identifiers, and Dust:-

- The lexer is designed to recognize exponential numbers by the pattern `[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)`.
- Identifiers follow the pattern `[A-Za-z][A-Za-z0-9_]*`.
- The token **DUST** is returned for any unrecognized characters that don't fit other patterns.

## 6. Count() Function:-

This function counts the columns and keeps track of the current position in the input file, making sure to handle newlines and other characters correctly.

## 7. End-of-File Handling:-

The `yywrap()` function ensures that the lexer returns **1** when it reaches to the end of the input, indicating that tokenization is complete.

## 8. Main Function:-

The `main()` function reads the input file name, opens the input and output files, and continuously calls `yylex()` to process tokens until the end of the input is reached.

# PARSER ANALYZER

Parser takes input code is analyzed according to the grammar rules defined using Yacc. The goal of this phase is to check the syntactic structure of the input. Here's a breakdown of its key components:

## Operator Precedence and Associativity:

- Operators like `+`, `-`, `*`, `/`, and `%` have their precedence defined with `%left`, and `%right` for right-associative operators.
- `%nonassoc` is used to define non-associative operators like `<`, `>`, and a special precedence rule `LOWER_THAN_ELSE` to handle **if-else** ambiguity.

## Token Definitions:

- Tokens like **FOR**, **WHILE**, **IF**, **ELSE**, **READCSVFUNC**, and various operators (**ADD\_ASSIGN\_OPERATOR**, **EQ\_OPERATOR**, etc.) are defined to represent keywords, operators, and function calls in the DSL.
- **SINGLE\_QUOTED\_STRING**, **INTEGER**, and **IDENTIFIER** capture the string literals, integers, and variable names.
- **Function Tokens**: Tokens like **READCSVFUNC**, **HEADFUNC**, and **MERGEFUNC** represent specific DSL functions for CSV handling.

## Grammar Rules:

- **translation\_unit**: The start symbol, representing the entire program. It's composed of one or more declarations.
- **declaration**: Handles type declarations, assignments, function definitions, and input operations.
- **function\_call\_statement**: Describes function calls like **READCSVFUNC()**, **HEADFUNC()**, and others used for CSV manipulation (grouping, merging, etc.).
- **assignment\_statement**: Defines how to assign results of function calls to identifiers.
- **control structures**: Implements **if-else**, **while**, and **for** loops, allowing control flow in the DSL.
- **parameter\_list**: Lists parameters for function calls.
- **expression**: Captures arithmetic and logical expressions, constants, and nested constructs.

## CSV-Specific Functionality:

- Functions like **READCSVFUNC**, **HEADFUNC**, **TOCSVFUNC**, **GROUPFUNC**, **MERGEFUNC**, and **MISSVALUEFUNC** provide specialized operations:
  - Reading, writing, and viewing CSV files.
  - Grouping and merging datasets.
  - Handling missing values with specific actions (**fill\_action**).

## Error Handling and Cleanup

- **yyerror()**: Reports syntax errors with meaningful messages.
- **yywrap()**: Handles parsing termination cleanly.

# SEMANTICS

## Structure Definition

- **struct sym:**
  - Represents each entry in the symbol table.
  - Fields:
    - **sno:** Serial number of the entry.
    - **token:** Name of the variable or function.
    - **type:** Data type(s) of the token (e.g., INT, FLOAT).
    - **paratype:** Parameter types for functions.
    - **tn:** Number of data type tokens.
    - **pn:** Number of parameters.
    - **scope:** The scope level of the token.

## Symbol Management

- **Insert Operations:**
  - **insert:** Adds new variables or functions to the symbol table.
  - **insert\_dup:** Inserts a duplicate variable with a specific scope.
  - **inserttp:** Adds parameter types to functions.
- **Lookup and Updates:**
  - **lookup:** Checks if a token exists in the symbol table.
  - **insertscope:** Updates the scope of a token.
  - **returntype:** Retrieves the type of a variable in a specific scope.
  - **returntypef:** Gets the return type of a function.

## Scope Handling

- **returnscope:** Finds the highest scope for a given token in the current scope.

## Functionality for Functions

- **Return Type:**
  - **storereturn:** Stores the return type of functions.
  - **Parameter Type Checking:**
    - **checkp:** Verifies parameter types during function calls.



## Display Function

- **display:**
  - Prints the symbol table in a formatted manner.
  - Shows identifiers, types, and parameter types.
  - Supports multiple data types (INT, FLOAT, VOID, FUNCTION).

## Applications

This symbol table implementation is suitable for:

- Managing scopes and types in programming languages.
- Handling parameter type checking and function overloading.
- Debugging or semantic analysis during compilation.

# SEMANTICS CHECKS

## Allowed Functions and Parameters:

- The **allowed\_functions** set defines valid DSL functions.
- The **function\_parameters** map specifies valid parameters for each function.

## Validation Methods:

- **checkFunction**: Verifies if the function is defined in the DSL and checks for invalid parameters.
- **checkMandatoryParameters**: Ensures all required parameters for a function are provided.
- **checkColumnExists**: Validates the existence of a specified column.
- **checkDataFrameExists**: Confirms the existence of a dataframe.
- **checkTypeCompatibility**: Ensures column types match expected types for operations.

## Function-Specific Analysis:

- **analyzeRead**: Validates the **read** function, ensuring mandatory parameters like **csv\_file** are provided and optional parameters are valid (e.g., separator values like **,** or **;**).
- **analyzeAggregateFunction**: Verifies numeric columns for aggregate operations like **mean** and **sum**.
- **analyzeConcat**: Validates concatenation of multiple dataframes along a specified axis (**0** for rows or **1** for columns).
- **analyzeMerge**: Ensures the correctness of merge operations by checking the merge type (e.g., **inner**, **outer**, etc.) and verifying dataframe existence.

## Error Handling:

- Comprehensive error reporting using **std::runtime\_error** for invalid function names, parameters, missing mandatory parameters, non-existent columns/dataframes, and type mismatches.

## Main Function:

- Defines example column types (**column\_types**) for semantic validation.
- Demonstrates semantic analysis of common DSL operations:
  - File reading (**analyzeRead**)
  - Aggregation (**mean** and **sum**)
  - Concatenation (**analyzeConcat**)
  - Merging (**analyzeMerge**)

# B+ TREE IMPLEMENTATION

**B+ tree**, a self-balancing tree structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. This tree is designed to store key-value pairs where each key is an integer, and the value is a string (representing row data). The tree also implements features like **splitting nodes**, **insertion in non-full nodes**, and **linked leaf nodes** for sequential access. It can load data from a CSV file, perform **head** and **tail** operations (viewing the first and last **n** rows), and **save the tree back to CSV**.

## Structure

### Node Struct

The **Node** struct represents a single node in the B+ tree:

- **is\_leaf**: A flag indicating if the node is a leaf (1) or internal (0).
- **num\_keys**: The number of keys currently stored in the node.
- **keys[]**: An array holding the keys.
- **data[]**: An array holding the data associated with each key.
- **children[]**: Pointers to child nodes. For leaf nodes, it is used to store **NULL** pointers, while internal nodes store child node references.
- **next**: A pointer to the next leaf node in the linked list.

## BPlusTree Struct

The BPlusTree struct represents the B+ tree itself, with a pointer to the root node.

### Insertion and Node Splitting

#### 1. Initial Insertion:

- The insert function starts by checking if the root node is full.
- If full, the root is split, and a new root is created, making the tree grow in height.

#### 2. Splitting a Node:

- The **split\_child** function splits the child node, moving half the keys and data to a new node.
- It also handles child pointers (for internal nodes) and links the new leaf node to the previous one if needed.

#### 3. Insertion into Non-Full Node:

- If the node is not full, the key-value pair is inserted at the correct position, ensuring that the node remains sorted.
- For internal nodes, if the appropriate child is full, the algorithm splits it before inserting.


### OVERVIEW:


**The B+ tree implementation efficiently manages key-value pairs with operations for insertion, node splitting, and sequential access (head and tail). The tree supports CSV data loading and saving, and the code ensures proper memory management. Future improvements could focus on handling deletions, optimizing certain operations, and improving error handling.**

# INPUTS


## INPUT-1

```
samp.rc ×  
samples > samp.rc  
1 input 'weather_data.csv';  
2 // Converts CSV file into data frame  
3  
4 df(df_1) = read('weather_data.csv');  
5 print(df(df_1));
```

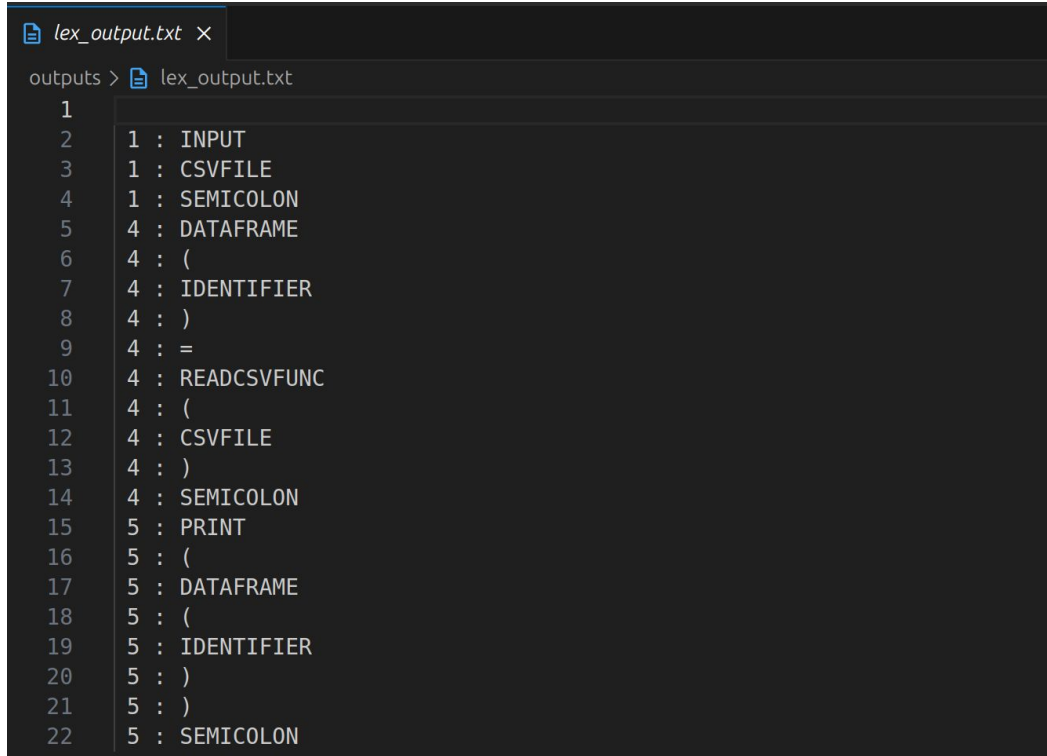
```
samples >  sample2.rc
1  //reads csv file as input
2  input weather_data.csv;
3  // Converts CSV file into data frame
4  df = read('weather_data.csv');
5  // Fills missing values with the median
6  df_filled = df.miss_value(fill,method=ffill);
7  // Group by Humidity and calculate the mean temperature
8  grouped = df.groupby('Humidity').agg({'Temperature': 'mean'});
9  // Reset index to make 'Humidity' a column again
10 grouped.reset_index();
11 //creating an output csv file
12 grouped.to_csv('Output.csv', index=False)
13 // Creating output csv of the grouped and aggregated data
14 output Output.csv
```

```
samples >  sample3.rc
1  //reads csv file as input
2  input weather_data1.csv;
3  // Converts CSV file into data frame
4  df=df_filled
5  //obtained from EXAMPLE CODE 1
6  df2 =read('weather_data1.csv')
7  // Merge the two datasets on 'Date'
8  merged_df = pd.merge(df, df2, on='Date', how='outer',suffixes=('_1', '_2')));
9  // Interpolate missing values
10 interpolated_df = merged_df.interpolate(inplace='False');
11 // Print the first few rows of the merged and interpolated data
12 interpolated_df.head();
```



```
samples >  sample4.rc
1  input 'weather_data.csv';
2  df(df_1) = read('weather_data.csv');
3
4  if(suvedh==1)
5  {
6      print("super");
7  }
8  else
9  {
10     print("not wow");
11 }
12
13 loop(i=0; i<10; i=i+1)
14 {
15     suvedh = suvedh+1;
16     suvedh = 100;
17 }
```

# OUTPUTS



```
lex_output.txt ×
outputs > lex_output.txt
1
2 1 : INPUT
3 1 : CSVFILE
4 1 : SEMICOLON
5 4 : DATAFRAME
6 4 : (
7 4 : IDENTIFIER
8 4 : )
9 4 : =
10 4 : READCSVFUNC
11 4 : (
12 4 : CSVFILE
13 4 : )
14 4 : SEMICOLON
15 5 : PRINT
16 5 : (
17 5 : DATAFRAME
18 5 : (
19 5 : IDENTIFIER
20 5 : )
21 5 : )
22 5 : SEMICOLON
```