# CS3423: Compilers 2

---

# Raccoons

~DSL for CSV Data processing~

RC

---

## Project Team Members

| Name | Roll Number | Roles |
|---|---|---|
| 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 |

**Instructor: Ramakrishna Upadrasta**

# Content

**1. Introduction** ..... 4
**2. Unique Service Proposition** ..... 5
2.1 Algorithm ..... 5
2.1.1 B+ Tree Implementation ..... 5
2.1.2 The two benefits of B+ Tree include ..... 5
2.1.3 Drawbacks of the Use of B+ Tree ..... 5
2.1.4 Why B+ Tree? ..... 6
2.2 Automated Type Checker ..... 6
2.3 Fork Implementation ..... 8
2.4 Multithreading for CSV File Functions ..... 8
2.5 Intermediate Implementation ..... 9
2.5.1 CSV data processing using MLIR ..... 9
2.5.2 Advantages of MLIR implementation:- ..... 10
**3. Declarations** ..... 11
3.1 VARIABLE DECLARATIONS ..... 11
3.2 FUNCTION DECLARATIONS ..... 11
3.2.1 Read CSV ..... 11
3.2.2 Head ..... 11
3.2.3 Tail ..... 11
3.2.3 Reset Index ..... 12
3.2.4 To CSV: ..... 12
3.2.5 Describe: ..... 12
3.2.6 Aggregate Functions ..... 12
3.2.6.1 mean ..... 12
3.2.6.2 mode ..... 12
3.2.6.3 median. ..... 12
3.2.6.4 sum ..... 13
3.2.6.5 min. ..... 13
3.2.6.6 max. ..... 13
3.2.7 Missing Values ..... 13
3.2.8 Exchange ..... 13
3.2.9 Grouping ..... 13
3.2.9.1 Groupby ..... 13
3.2.9.2 Concat ..... 14
3.2.9.3 Merge ..... 14
3.2.9.4 Join ..... 14
**4. Tokens** ..... 15

4.1 Comments ..... 15
4.2 Whitespace ..... 15
4.3 Keywords. ..... 16
4.4 Punctuators. ..... 17
4.5 Identifiers. ..... 18
4.6 Constants ..... 19
4.7 Operators. ..... 20
**5. Sample codes ..... 21**
EXAMPLE CODE1: Handling Missing Values and Aggregating Data ..... 21
EXAMPLE CODE 2 : Grouping and Aggregation with 'groupby' ..... 22
EXAMPLE CODE 3: Merging Datasets and Handling Missing Values ..... 23

# 1. Introduction

In recent years, data is ubiquitous in almost every domain. It is being leveraged for predictions, recommendations, decision making etc. Most of this data is in the form of Comma Separated Values (CSV) due to the simplicity of the format and the ability to view them in an intuitive tabular format. Despite their simplicity, as the datasets grow in size, managing CSV files and analysing data can become cumbersome.

Some existing solutions for this include the famous Pandas library in Python, and the R language. Pandas helps to deal with data present in different formats and provides extensive functions for various data processing tasks. However, since python is a general purpose interpreted language, it is not the best in terms of performance. In addition, Pandas is single-threaded making it slower when the size of the data increases. Another widely used alternative is R. The R language, though good in terms of performance in data wrangling tasks, doesn't have intuitive syntax making it difficult for non-programmers to understand and write R code for managing and analysing the data.

We provide a more efficient alternative to these available commonly used options by designing a user-friendly and well-performing Domain Specific Language to deal with huge datasets in the form of CSV files. Our main goal is to improve programmer productivity and allow people from different domains to deal with CSV files comfortably by providing a highly intuitive syntax while ensuring that all the necessary data handling operations are supported. The target users for our DSL include data scientists, data analysts as well as domain experts with minimal programming experience. We design a robust and highly optimised compiler for our DSL using the Multi-Level Intermediate Representations (MLIR) infrastructure.

We named our DSL as 'Raccoons' inspired by the strategic problem solving ability of these creatures. Raccoons are intelligent and highly adaptable animals. Our DSL is designed to handle large CSV datasets providing flexibility to perform various data processing and analysis tasks similar to how raccoons navigate complex environments with precision.

Our DSL's syntax is highly inspired from the simplicity and user-friendliness of Pandas library in Python. We provide most of the useful features provided by Pandas and perform optimizations at various stages to improve the performance of the data processing tasks. We use the C++ programming language for implementing our Domain Specific Language to ensure that the performance of the code is good but provide the users highly flexible and intuitive functions to manage their CSV data. In essence, we intend to bridge the gap between intuitiveness, user-friendliness and performance in data processing tasks.

Our DSL has a lot of unique features compared to other existing CSV data processing toolkits and libraries. The unique features of our DSL are enlisted in chapter 2. The language features including tokens and various functions we are going to provide for data cleaning and data analysis tasks for the domain experts are elaborately described in chapters 3 and 4.

# 2. Unique Service Proposition

This part discusses the specific characteristics of our system with an emphasis on the utilization of high-performance data structure, parallelism and forking and grouping functionality.

## 2.1 Algorithm

This subsection contains descriptions of the algorithms that underpin our system including examples of how they are used, their advantages and disadvantages.

### 2.1.1 B+ Tree Implementation

The Raccoons DataFrame is employed within our system. The system utilizes a B+ Tree for the storage and management of data. The use of the B+ Tree is made because of its balanced nature, which is good for both the read and the write operations. Using the B+ Tree, we can add, delete, search, and update the information in the large data base with high access frequency and, at the same time, organize the information in the necessary order. This implementation helps us to optimise utilizing such resources for complicated data queries as well as storage processes. That is, we have also incorporated indexing in the B+ Tree, which further increases the data search speed. This indexing mechanism is even faster in retrieving the records, and therefore, the B+ Tree is a key part of our system's performance plan.

### 2.1.2 The two benefits of B+ Tree include

- **Efficient Data Access**: Since search, insert and delete operations of the proposed B+ Tree are log base 'N' time complexity, they are efficient to handle big datasets.
- **Scalable Storage:** Due to the structure of the B+ Tree, which is a balanced tree structure, larger amounts of data could be stored without posing a severe problem to access through comparatively fewer number of I/O operations.
- **Optimized for Range Queries:** Because of the ordered structure of B+ Trees, range query is possible in a faster manner and its usage is very ideal for grouping or sorting of data together with the indexing mechanism that can also speed up the retrieval time, thus making the B+ Tree as an important facet in the performance of our system.

### 2.1.3 Drawbacks of the Use of B+ Tree

- **Implementation Complexity:** Compared with simple data structures like Lists, Stacks and Queues implementing and sustaining B+ Trees involves additional work including node construction and the practice of balancing.
- **Memory Overhead:** There is a constant necessity of the space for holding "pointers" and other extra data in the tree nodes to enhance the functionality of the search tree, and this gives rise to memory consumption as a potential disadvantage that emerges in circumstances where system resources are scarce.
- **Insertion Slowness:** A disadvantage of using B+ Trees is that insertion will be a bit slower when compared with binary search tree and this is made even worse as the complexity of the tree increases.

### 2.1.4 Why B+ Tree?

The reasons behind the choice of the B+ Tree for our system is because of its capacity to store a huge amount of data which often have numerous accesses. Because balanced structures significantly maintain the data operation speed and coherence, it is the best match for the system that needs fast and efficient operation.

### 2.2 Automated Type Checker

**AUTO-TYPE CHECKING:**

We want to check and assign the inputs to their respective data types, For this what we are going to do is checking the input to which class it is going to be, like we are going to check whether it is an exponent or not first, if not then it will check the percentage type and follows a series. One may ask why the order is like this only, it's because if we are going to reverse the order then every other rule is going to be overshadowed, to avoid this we are checking from most specific to least specific types.

**Example Implementation:**

```
“int”       { return INT_KEYWORD; }
“float”     { return FLOAT_KEYWORD; }
“string”    { return STRING_KEYWORD; }
“bool”      { return BOOL_KEYWORD; }
“true”      { return TRUE_KEYWORD; }
“false”     { return FALSE_KEYWORD; }
“input”     { return INPUT_KEYWORD; }
“output”    { return OUTPUT_KEYWORD; }
“datatype”  { return DATATYPE_KEYWORD; }
“percentage” {return PERCENTAGE_KEYWORD;}

[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+) { return EXPONENTIAL; }
[0-9]+(\.[0-9]+)?%    { return PERCENTAGE; }
[0-9]+(\.[0-9]+)    { return FLOAT; }
[+-]?[0-9]+               { return INTEGER; }
\”([^\"\\]|\\.)*\” { return STRING; }
.                   { /* DUST_BIN */ }
```

**AUTO-TYPE CASTING:**

In normal cases it will check what are different constants are going to participate in aggregate operations and based on the priority it will be type casted automatically. For example:

```
int typeCheck(int type1, int type2, int op)
{
   if (type1 == INTEGER && type2 == FLOAT)
   {
      return FLOAT; // Only deals with int and float number
   }
```

```
    return -1;
}

int applyCast(int from_type, int to_type, int value)
{
    if (from_type == INTEGER && to_type == FLOAT)
    {
        return (float)value; // Casting from One value to another when required
    }
    return value;
}
```

**ADJUSTING OVERFLOWS AND UNDERFLOWS:**

```
int main()
{
    int a, b;
    scanf("%d %d", &a, &b);

    int int_result = a + b;

    if ((a > 0 && b > 0 && int_result < 0) || (a < 0 && b < 0 && int_result > 0))
    {
        long int long_result = (long int)a + (long int)b;

        if (long_result > LONG_MAX || long_result < LONG_MIN)
        {
            long long int ll_result = (long long int)a + (long long int)b;
         }
    }

    return 0;
}
```

Basically we are going to check each and every time the best fit data type for our aggregation operations (Inspired from Best Fit in paging), Our code will every time try to compute the aggregations for example add in this case, it will check did the data got overflowed or underflowed based on it the data type is going to be fixed.

**Main Motive:**

One may ask that I'll always give a long long int for everything to reduce user coding complexity, but it will lead to use of excess space than required, so that internal fragmentation is going to be very high. So to avoid this we are going to check the Best Fit by going from lower memory to higher memory. In general it can be used for any kind of aggregate operation and this is our basic implementation for that problem.

### 2.3 Fork Implementation

The primary goal is to preserve the original B+Tree as a constant data frame. To enable manipulation of the B+Tree without altering the original, we need to implement a fork() function.

The purpose of the fork() function is to ensure that the original B+Tree remains unchanged, which is crucial for maintaining the integrity of operations like Groupby() that rely on a stable data structure. When fork() is invoked, it creates two processes, a parent process that holds the original B+Tree and a child process that contains a copy of the B+Tree. All modifications are

made within the child process, leaving the parent process, and thus the original B+Tree, untouched.

**Parent Process:** This would hold the original B+Tree. Any operation that requires a stable, unaltered data structure would reference this tree.

**Child Process**: This would be a copy of the B+Tree created by the fork() function. Any manipulations or temporary changes would be made here. If the changes need to be permanent, the child process can replace the parent process ensuring that the integrity of the B+Tree is maintained while allowing for necessary modifications. This approach guarantees that the original B+Tree remains unaffected, preventing any issues during the execution of various functions.

This ensures that the main B+Tree remains unchanged, so we avoid any potential problems or issues during operations.

### 2.4 Multithreading for CSV File Functions

In our system, we use multithreading for CSV file operations among others as a way of improving performance especially for any function that involves aggregation and missing values. These operations can actually be done in parallel in multithreaded environments thus averting time wastage for huge data sets. We divide our B+ tree into multiple subtrees and use multithreading on each subtree, it is now easier to undertake extensive data processing since the processing task is divided into many threads, hence, large and complex CSV files can be processed easily.

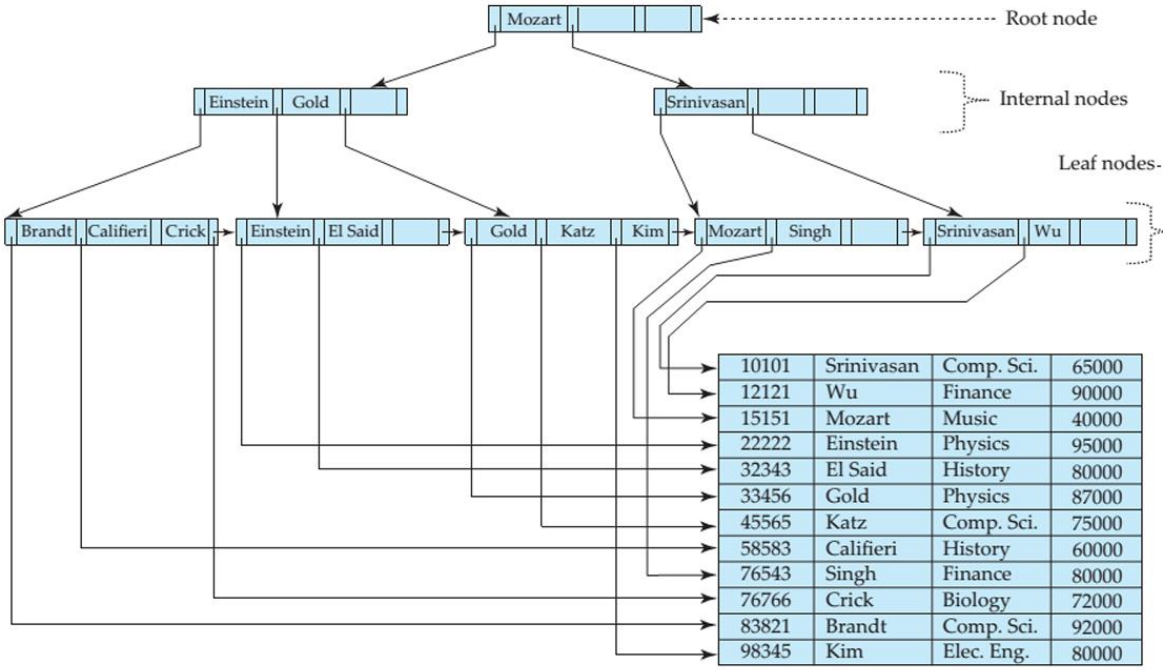


Image taken from Silberschatz A databases

### 2.5 Intermediate Implementation

**MLIR (Multi-Level Intermediate Representation**) is a flexible compiler infrastructure that can be used for tasks like data preprocessing and data analysis for the CSV processes.

#### 2.5.1 CSV data processing using MLIR

**The approach for processing CSV using MLIR:-**

- **Define a Language Design**
- Syntax and Semantics: Our DSL is designed specifically for CSV processing, focusing on operations like reading, filtering, and writing CSV files.
- We Create a custom MLIR language that represents the operations and constructs defined in your DSL.
- Implementation of the necessary operations, specifying their behavior and attributes are done within the MLIR framework.

- **Parsing CSV Data**
- Lexical Analysis, and parser implementation is independent of MLIR.
- Syntax tree to MLIR Conversion: We write frontend that translates the syntax tree into MLIR operations which are in the custom language format.

- **Transformations and Optimizations**
- High-Level Optimizations: We implement MLIR passes to optimize the CSV processing operations. Examples include operation fusion and elimination of redundant operations.

- Data Layout Optimizations: Also Optimization of the data layout for efficient processing.
- Lowering Passes: We then gradually lower the high-level operations in the custom MLIR language to more generic MLIR languages.

**Example Workflow**

DSL Code: CSV processing code in the DSL.

```
example:

read_csv("data.csv")
 df.exchange (2, 20,{B},inplace=True); // at column B
to_csv("filtered_data.csv")
```

**2.5.2 Advantages of MLIR implementation:-**

a) Multi-Level Abstraction: Our MLIR supports multiple levels of abstraction, from high-level operations down to low-level code. So this flexibility allows for more domain-specific optimizations tailored to CSV processing.

b) Custom Language: Our MLIR allows the creation of custom language, which can represent high-level CSV processing operations directly. This makes it easier to implement and optimize domain-specific Transformations.

c) Progressive Lowering: Our MLIR supports progressive lowering, where high-level operations are gradually transformed into lower-level representations. With the help of this approach, maintaining high-level information for a longer period of time and thus enabling better optimizations.

# 3. Declarations

Declaration statements are used to define variables or functions, specifying their data type and name. By default, the int type variable is initialized by 0, the float type variable by 0.0, string type variable by “” or null and the bool type variable by false.

### 3.1 VARIABLE DECLARATIONS

A variable is a named storage location that holds data. It is followed by a datatype and the statement ends by ’;’

Examples:-

```
int x;
float a = 0.14;
string = ”HELLO”;
bool = True;
```

### 3.2 FUNCTION DECLARATIONS

These Functions are mainly used for Data Cleaning and Data Manipulation.

#### 3.2.1 Read CSV

-read(): This function reads a CSV file into a data frame, the B+ tree.

Example:

```
df = read_csv(
    'data.csv',         // Path to the CSV file
    sep=';',            // Use a semicolon as the delimiter
    header=0,           // First row as the header
    index_col=0,        // Use the first column as the index
    usecols=['A', 'B']  // Only read the columns 'A' and 'B'
);
```

#### 3.2.2 Head

-head(): This function is used to view the first few rows of the data frame.

Example:

```
// Get the first 5 rows of the DataFrame
df_head = df.head();  // Default is 5 rows

// Alternatively, get the first N rows (e.g., 10 rows)
df_head_10 = df.head(10);
```

#### 3.2.3 Tail

-tail(): This function is used to view the final few rows of the data frame.

Example:

```
// Get the tail 5 rows of the DataFrame
df_tail = df.tail();  // Default is 5 rows

// Alternatively, get the tail N rows (e.g., 10 rows)
```

df_tail_10 = df.tail(10);

### 3.2.3 Reset Index

-reset_index(): This function is used to reset the index of a DataFrame. It is often used after performing operations that change the index, and you want to revert to a default integer index.
'drop =True' means the old index is not added as a new column in the DataFrame.
'inplace = True' means the operation is done in place, modifying the existing DataFrame.

Example:
df.reset_index(drop=True, inplace=True);

### 3.2.4 To CSV:

-to_csv(): This function writes a DataFrame to a CSV file. It's the opposite of read() and allows you to save your processed data.
'inplace = True' means the operation is done in place, modifying the existing DataFrame.
Example:-
df.to_csv('output.csv', index=True);

### 3.2.5 Describe:

- describe(): This function provides a quick overview of the statistical properties of the numerical columns in a data frame, such as mean, min and max.
Example:- df.describe();

### 3.2.6 Aggregate Functions

- Aggregate functions are used to perform operations on a group of data, returning a single summary value. Common aggregate functions include mean(),mode(),median(), sum(), min(), max().
In aggregate function axis is used to specify whether it is column or row but in default (axis = 0) is used for column wise and (axis = 1) is used for row wise.

**3.2.6.1 mean():-** mean is used to calculate the average(mean) along a specific axis in the dataframe.
Example:-

```
// Calculate the mean for each column (axis=0)
column_mean = df.mean(axis=0) ;
// Calculate the mean for each row (axis=1)
row_mean = df.mean(axis=1);
```

**3.2.6.2 mode():- mode is used to calculate the most frequency occurring value along a specific axis in the dataframe.**
Example:- df.mode();

**3.2.6.3 median():-** median is used to calculate the middle value along a specific axis in the dataframe.
Example:- df.median();

**3.2.6.4 sum():- sum is used to calculate the sum of values along a specific axis in a dataframe.**

Example:- df.sum();

**3.2.6.5 min():-** min is used to calculate the minimum values along a specific axis in a dataframe.

Example:- df.min();

**3.2.6.6 max():-** max is used to calculate the maximum values along a specific axis in a dataframe.

Example:- df.max()'

### 3.2.7 Missing Values

Handling missing values is crucial in data cleaning. It offers several methods:

i) Replaces missing values with a specific value.

Example:-

df.miss_value(fill,const,inplace=True/False);

ii)Using Method

a) Method = ffill, performs forward fill with previous value.

Example:-

df. miss_value(fill,method= ffill,inplace=True/False);

b) Method = bfill, performs backward fill with next value.

Example:-

df. miss_value(fill,method= bfill,inplace=True/False);

c) Method = interpolation, performs fill with mean of previous and next value.

Example:-

df.miss_value(fill,method= interpolate,inplace=True/False);

iii) Removes rows or columns with missing values.

Example:-

df.miss_value(drop,inplace=True/False);

### 3.2.8 Exchange

Exchange value of a particular data in the dataframe with a user specified value in a column or a row.

df.exchange(before, after, row/column filter,inplace=True/False)

Example:-

// Exchange a value in a specific column (replace '2' with '20' in column 'B')

df.exchange (2, 20,{B},inplace=True);

### 3.2.9 Grouping

Grouping involves splitting the data into groups, applying a function to each group independently, and then combining the results.

#### 3.2.9.1 Groupby

**- groupby()**: Splits the data into groups based on some criteria (e.g., a column value), allowing you to perform operations on each group independently then apply aggregate functions.

Example:-

// Group by 'Category' and calculate the sum of 'Values'

grouped_sum = df.groupby('Category')['Values'].sum();

### 3.2.9.2 Concat

**-concat()**:- Concatenates or joins two or more DataFrames along a particular axis (rows or columns). It allows you to stack DataFrames either vertically or horizontally.

Example:-

concat([df1, df2], axis=0);

### 3.2.9.3 Merge

**- merge()**: Merges two DataFrames based on one or more common keys or indices. This function is similar to SQL joins and is more flexible than 'concat' when merging datasets that share one or more common columns or indices.

how: Type of merge to be performed - left, right, outer, inner(default).

on: Column or index level names to join on. It must be found in both DataFrames.

Example:-

merge(df1, df2,how='left', on='key',suffixes=('_1', '_2')))

### 3.2.9.4 Join

**- join()**: Joins two DataFrames on their indices or on a specified key column. It is primarily used for combining DataFrames that have overlapping indices or that should be aligned on a specific column.

on: The column or index to join on.

how: The type of join to be performed - left(default), right, inner, outer

Example:-

df1.join(df2,how='left', on='key')

# 4. Tokens

### 4.1 Comments

Comments are used to explain and annotate code. They are ignored by the C compiler and serve to enhance code readability.

**Single-Line Comments**: Begin with // and extend to the end of the line.

For example:

```
int square(int num) {
    return num * num;  // Return the square
}
```

**Multi-Line Comments**: Enclosed within /* and */, these comments span multiple lines.

For example:

```
int factorial(int n) {
    if (n <= 1) return 1; /* returns the factorial of the number */
    else return n * factorial(n - 1);
}
```

### 4.2 Whitespace

Whitespace includes spaces, tabs, and newlines. While it does not affect the execution of code, it is crucial for code formatting and readability.

- **Spaces**: Separate tokens and improve readability.

For example:

```
int sum = a + b;
```

**Tabs**: Typically used for indentation.

For example:

```
void example() {
        int x = 10;
        int y = 20;}
```

**Newlines**: Separate lines of code.

For example:

```
int main() {
        printf("Hello, world!\n");
        return 0;
}
```

### 4.3 Keywords

**Keywords are reserved words in C that have special meanings and cannot be used as identifiers.**

- **Common Keywords**: Examples include int, return, if, else, while, and for.

For example:

```
int main() {
        int x = 5;
        if (x > 0) {
        printf("Positive number\n");
        }
        return 0;
}
```

- **4.4 Punctuators**

**Punctuators are symbols used to structure and organize code, defining syntax rules.**

**Semicolon (;)**: Terminates statements.

For example:

```
int a = 10;
int b = 20;
```

**Comma (,)**: Separates items in function calls or variable declarations.

For example:

```
int x = 5, y = 10;
printf("x: %d, y: %d\n", x, y);
```

**Brackets ([])**: Used for array indices.

For example:

```
int arr[5];
arr[0] = 10;
```

**Braces ({})**: Define blocks of code, such as function bodies or control structures.

For example:

```
void function() {
int a = 5;
```

```
        {
        int b = 10;
        printf("Inside block: %d\n", b);
        }
        printf("Outside block: %d\n", a);
    }
```

**Parentheses (())**: Group expressions and enclose function parameters.

For example:

```
        int sum(int a, int b) {
        return a + b;
    }
         int main() {
            int result = sum(5, 10);
          printf("Sum: %d\n", result);
            return 0;
     }
```

## 4.5 Identifiers

**Identifiers are names used for variables, functions, and other entities. They must follow specific naming rules.**

- **Naming Rules**: Identifiers must start with a letter or underscore and can include letters, digits, or underscores. They are case-sensitive.
  For example:

```
int counter;
                void printMessage() {
                   printf("Hello\n");
                 }
```

## 4.6 Constants

**Constants represent fixed values that do not change during the program's execution. They provide a way to define immutable values.**

- **Definition**: Constants are defined using the const keyword or #define preprocessor directive.

For example:

```
#define PI 3.14159
const int MAX_SIZE = 100;
int main() {
        printf("Value of PI: %f\n", PI);
        printf("Max Size: %d\n", MAX_SIZE);
        return 0;}
```

**Types**: Constants can be numeric, string, or boolean.

For example:

```
  const double GRAVITY = 9.81;  // Numeric constant
const char *GREETING = "Welcome";  // String constant
```

### 4.7 Operators

| Operator | Description | Associativity |
|---|---|---|
| … | ELLIPSIS | N/A |
| >> | Right Shift | Left to Right |
| << | Left Shift | Left to Right |
| ++ | Increment | N/A |
| -- | Decrement | N/A |
| && | AND | Left to Right |
| \|\| | OR | Left to Right |
| ! | NOT | Right to Left |
| ^ | XOR | Left to Right |
| <= | Less than Equal to | Left to Right |
| >= | Greater than Equal to | Left to Right |
| = = | Equal to Equal to | Left to Right |
| ! = | NOT Equal to | Left to Right |
| ; | Semi- Colon | N/A |
| = | Equal to | Right to Left |
| { } | Curly Braces | N/A |
| ( ) | Parenthesis | N/A |
| , | Comma | Left to Right |
| : | Colon | N/A |
| [ ] | Square Braces | N/A |
| . | Dot | Left to Right |
| ~ | BITWISE NOT | Right to Left |
| & | BITWISE AND | Left to Right |
| - | Subtraction | Left to Right |
| + | Addition | Left to Right |
| * | Multiplication | Left to Right |
| / | Division | Left to Right |
| % | Modulo | Left to Right |
| < | Less than | Left to Right |
| > | Greater than | Left to Right |
| \| | BITWISE OR | Left to Right |
| ? | Optional | N/A |

# 5. Sample codes

**EXAMPLE CODE1: Handling Missing Values and Aggregating Data**
**Sample input:- 'weather_data.csv':**

```
Date, Temperature, Humidity, WindSpeed, Precipitation
2024-08-01,32, ,15,0
2024-08-02, ,80,10,0.2
2024-08-03,28,78,12,
2024-08-04,30,85,8,0
2024-08-05,34,75,14,0.5
```

**Input:**

```
//reads csv file as input
input  weather_data.csv;
// Converts CSV file into data frame
df = read('weather_data.csv');
// Fills missing values with the median
df_filled = df.miss_value(fill,df.median());
// Calculate aggregate statistics
mean_temp = df_filled['Temperature'].mean();
max_wind_speed = df_filled['WindSpeed'].max();
sum_precipitation = df_filled['Precipitation'].sum();
// Print results
print("Mean Temperature:"+mean_temp);
print("Max Wind Speed:"+max_wind_speed");
print("Total Precipitation:" + sum_precipitation);
df_filled.head();
```

**Output:**

```
Mean Temperature: 31.0
Max Wind Speed: 15
Total Precipitation: 0.8
```

| | Date | Temperature | Humidity | WindSpeed | Precipitation |
|---|---|---|---|---|---|
| **0** | 2024-08-01 | 32.0 | 79.0 | 15 | 0.0 |
| **1** | 2024-08-02 | 31.0 | 80.0 | 10 | 0.2 |
| **2** | 2024-08-03 | 28.0 | 78.0 | 12 | 0.1 |
| **3** | 2024-08-04 | 30.0 | 85.0 | 8 | 0.0 |
| **4** | 2024-08-05 | 34.0 | 75.0 | 14 | 0.5 |

**EXAMPLE CODE 2 : Grouping and Aggregation with 'groupby'**
**Sample input:- 'weather_data.csv':**

```
Date, Temperature, Humidity, WindSpeed, Precipitation
2024-08-01,32, ,15,0
2024-08-02, ,80,10,0.2
2024-08-03,28,78,12,
2024-08-04,30,85,8,0
2024-08-05,34,75,14,0.5
```

**Input:**

```
//reads csv file as input
input  weather_data.csv;
```

```
// Converts CSV file into data frame
df = read('weather_data.csv');
// Fills missing values with the median
df_filled = df.miss_value(fill,method=ffill);
// Group by Humidity and calculate the mean temperature
grouped = df.groupby('Humidity').agg({'Temperature': 'mean'});
// Reset index to make 'Humidity' a column again
grouped.reset_index();
//creating an output csv file
grouped.to_csv('Output.csv', index=False)
// Creating output csv of  the grouped and aggregated data
output Output.csv
```

**Output:**

| | **Humidity** | **Temperature** |
|---|---|---|
| **0** | 75.0 | 34.0 |
| **1** | 78.0 | 28.0 |
| **2** | 80.0 | 32.0 |
| **3** | 85.0 | 30.0 |

**EXAMPLE CODE 3: Merging Datasets and Handling Missing Values**
**Sample input :- 'weather_data.csv':**

```
Date, Temperature, Humidity, WindSpeed, Precipitation
2024-08-01,32, ,15,0
2024-08-02, ,80,10,0.2
2024-08-03,28,78,12,
2024-08-04,30,85,8,0
2024-08-05,34,75,14,0.5
```

**Sample input:- 'weather_data1.csv':**

```
Date, Temperature, Humidity, WindSpeed, Precipitation
2024-08-01,32,80,15,0
2024-08-02,31,80,10,0.2
2024-08-03,28,78,12,0
2024-08-04,30,85,8,0.3
2024-08-05,34,75,14,0.5
```

**Input:**

```
//reads csv file as input
input  weather_data1.csv;
// Converts CSV file into data frame
df=df_filled
//obtained from EXAMPLE CODE 1
df2 =read('weather_data1.csv')
// Merge the two datasets on 'Date'
merged_df = pd.merge(df, df2, on='Date', how='outer',suffixes=('_1', '_2')));
// Interpolate missing values
interpolated_df = merged_df.interpolate(inplace='False');
// Print the first few rows of the merged and interpolated data
interpolated_df.head();
```

**Output:**

| | Date | Temperature_1 | Humidity_1 | WindSpeed_1 | Precipitation_1 | Temperature_2 | Humidity_2 | WindSpeed_2 | Precipitation_2 |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 2024-08-01 | 32.0 | 79.0 | 15 | 0.0 | 32 | 80 | 15 | 0.0 |
| **1** | 2024-08-02 | 31.0 | 80.0 | 10 | 0.2 | 31 | 80 | 10 | 0.2 |
| **2** | 2024-08-03 | 28.0 | 78.0 | 12 | 0.1 | 28 | 78 | 12 | 0.0 |
| **3** | 2024-08-04 | 30.0 | 85.0 | 8 | 0.0 | 30 | 85 | 8 | 0.3 |
| **4** | 2024-08-05 | 34.0 | 75.0 | 14 | 0.5 | 34 | 75 | 14 | 0.5 |