MASTERING MEWAR PROGRAMMING

Code Like Royalty

- ✓ Learn Mewar from Scratch
- ✓ Interactive Coding Lessons
- ✓ Build Real Projects

BEGINNER TO ADVANCED

ROYAL COURSE

MEWAR PROGRAMMING LANGUAGE

A Beginner-Friendly Guide to Intent-Based Programming

Evolution of Mewar Language (v1–v5)

Author
Karan Soni

Inspired by the spirit of Mewar — courage, clarity, and innovation.

About the Author

Karan Soni is the creator of the Mewar Programming Language, a modern programming language designed to make learning programming simple, clear, and meaningful. His goal is to help beginners understand programming concepts through readable syntax and powerful built-in features.

Inspired by the historic land of Mewar, known for courage, leadership, and intelligence, Karan developed this language with a focus on clarity, structured thinking, and intent-based programming. The language introduces unique concepts such as goal, explain, tell, and show, which help programmers not only execute code but also understand and analyze their programs.

Karan is passionate about programming language design, software development, and creating educational tools that simplify complex technical concepts. Through the Mewar project, he aims to build a learning environment where students and

developers can explore programming in a more intuitive and meaningful way.

The Mewar Programming Language represents his vision of combining innovation, learning, and creativity to inspire the next generation of programmers.

Introduction to the Mewar Programming Language:

Mewar is a modern, beginner-friendly programming language designed to make coding simple, powerful, and meaningful. It is built with the vision that programming should not only be about writing instructions for a machine, but also about developing clear thinking, discipline, and problem-solving strength in the programmer. With its simple English-like syntax, Mewar allows beginners to learn programming easily while still providing advanced capabilities such as matrices, pointers, storage systems, and intelligent features like goal, explain, tell, and show. These features make Mewar unique because it does not just execute code—it helps users understand and analyze their own programs.

The name “Mewar” is inspired by the glorious history of India, especially the land of courage, honor, and sacrifice. It reflects the spirit of legendary warriors such as **Maharana Pratap**, **Rana Sanga**, **Prithviraj Chauhan**, **Maharana Kumbha**. These heroes were known for their bravery, leadership, intelligence, and never-give-up attitude.

TABLE OF CONTENTS

Chapter 1

Introduction to Mewar

1.1 ‘Mewar’: An Introduction

Mewar is a human-first programming language designed around intent, clarity, and understanding. Unlike many traditional programming languages that mainly focus on giving instructions to a computer, Mewar focuses on helping the programmer clearly express what they want to achieve and why they want to achieve it. The goal of the language is to make programming easier to read, easier to understand, and easier to learn.

Most programming languages require beginners to remember complex syntax, symbols, and strict formatting rules. This can make the learning process difficult for new programmers. Mewar attempts to reduce this difficulty by using simple, readable, English-like keywords such as set, say, ask, goal, explain, tell, and show. Because of this approach, programs written in Mewar look closer to natural language, making them easier to understand even for beginners.

One of the main ideas behind Mewar is that a program should clearly communicate its purpose. Instead of writing code that only the computer understands, Mewar encourages writing code that both the computer and the programmer can easily

read. This improves readability, reduces confusion, and makes programs easier to maintain and debug.

Mewar emphasizes several important principles. Readability means that programs should be written in a way that other programmers can easily read and understand. Simplicity means the syntax should remain straightforward so beginners do not feel overwhelmed. Clear intent means the purpose of the program should be visible directly in the code. Outcome awareness means the programmer should understand what happens during execution and why the program produces certain results.

Because of these principles, Mewar is designed to be beginner-friendly, educational, structured, and expandable. Beginners can quickly start writing simple programs, while more advanced users can explore complex features such as lists, matrices, pointers, functions, and structured data.

Using Mewar, programmers can perform many fundamental programming tasks. They can compute values using arithmetic operations, display information to the user, take input from the user, make decisions using conditions, and repeat actions using loops. They can also organize programs using functions and work with structured data types.

Another unique aspect of Mewar is its support for Intent-Based Programming. Through commands such as goal, explain, tell, and show, programs can describe their purpose, analyze data, and display internal program state. These

features allow programmers not only to run programs but also to understand how those programs work internally.

For learners, this approach makes programming more meaningful. Instead of writing code blindly, students can observe how their program behaves, why certain results appear, and how the logic flows from one step to another.

In this way, Mewar is not only a programming language but also a learning environment designed to develop logical thinking, structured problem solving, and deeper understanding of programming concepts.

1.2 Structure of a Mewar Program

A Mewar program is written as a sequence of instructions that are executed one after another. Each instruction tells the system what action should be performed. In most cases, every instruction is written on a separate line so that the program remains clear and easy to read.

The structure of a program is designed to be simple and readable. Because Mewar uses English-like keywords, even beginners can quickly understand how a program is organized and how it runs.

Example:

```
set x to 10

say x
```

Out put
10

In this example, the first line creates a variable named **x** and assigns the value **10** to it. The second line prints the value stored in **x**. When the program runs, the system reads the first instruction, stores the value, and then moves to the next instruction to display the result.

Basic structure:

1. Variable declarations
2. Logic (conditions / loops)
3. Output statements

   General format:

```
set variable to value
perform operations
say result
```

   Programs are executed line by line.

Program Execution Flow

1.3 Running the First Program

Now let us run the simplest Mewar program. Writing and running small programs is the best way to understand how a programming language works. This example shows how a value can be stored in a variable and then displayed on the screen.

Example program:

```
set number to 5
say number
```

When executed, the output will be:

```
5
```

Explanation:

In the first line, the instruction set number to 5 creates a variable called number and stores the value 5 in it. A variable acts like a container that holds data which can later be used by the program.

The second line say number tells the system to display the value stored in the variable number. Since the variable contains the value 5, the system prints 5 as the output.

This example demonstrates how a simple program works in Mewar. The interpreter reads and executes each instruction one by one in the order they appear.

1.4 Tokens in Mewar

In any programming language, a program is made up of small building blocks. These building blocks are called **tokens**. A token is the smallest meaningful unit of a program that the interpreter or compiler can understand.

When a program is written, the interpreter first breaks the program into tokens and then analyzes them to understand what the program is trying to do. Without tokens, the system would not be able to interpret the instructions correctly.

Consider the following statement:

```
set x to 10
```

This statement can be divided into several tokens:

This statement can be divided into several tokens:

- set → keyword
- x → identifier
- to → keyword
- 10 → constant

Each of these parts has a specific meaning in the program. The keyword **set** tells the system that a variable is being assigned a value. The identifier **x** represents the name of the variable. The keyword **to** connects the variable with the value being assigned. The constant **10** represents the value that will be stored in the variable.

Tokens help the interpreter understand the structure of the program and determine how each instruction should be executed.

In the Mewar programming language, tokens are generally divided into the following categories:

Keywords – Reserved words that have special meaning in the language, such as set, say, if, and while.

Identifiers – Names given to variables, functions, or other user-defined elements.

Constants – Fixed values that do not change during program execution, such as numbers or text values.

Operators – Symbols or keywords used to perform operations on values, such as addition, subtraction, or comparison.

Symbols – Special characters used for structure and organization in a program.

1.5 Keywords

Keywords are reserved words that have a special meaning in a programming language. These words are predefined by the language and are used to perform specific operations or define the structure of a program. Because keywords already have a fixed meaning, they cannot be used as names for variables, functions, or other user-defined elements.

In the Mewar programming language, keywords help define actions such as creating variables, displaying output, controlling program flow, and performing advanced operations. They make the language easier to read because most of them are written in simple English-like words.

For example, the keyword **set** is used to assign a value to a variable, and the keyword **say** is used to display output on the

screen. Similarly, control keywords such as **if**, **elif**, and **else** help the program make decisions based on conditions.

The Mewar language includes several groups of keywords based on their purpose.

Output keywords are used to display information to the user. These include **say**, **show**, and **print**.

Variable-related keywords are used to create and manage variables. These include **set** and **const**.

Control keywords help manage the flow of execution in a program. These include **if**, **elif**, **else**, and **nested if**.

Loop keywords are used to repeat instructions multiple times. These include **repeat**, **while**, **for**, **do**, **step**, and **stop**.

Function keywords are used to define and execute functions. These include **function**, **return**, and **call**.

Data type keywords represent different types of data structures supported by Mewar. These include **list**, **string**, and **matrix**.

Advanced programming keywords provide more powerful features such as memory references and structured data. These include **pointer**, **record**, and **enum**.

Error handling keywords help manage runtime errors safely within a program. These include **try**, **catch**, and **error**.

Mewar also includes special keywords used for **Intent-Based Programming**, which is a unique feature of the language. These keywords include **goal**, **explain**, **tell**, and **show**. They allow programs to describe their purpose, analyze data, and explain how they work.

Example:

```
set age to 18
```

In this example, **set** and **to** are keywords used to assign the value **18** to the variable **age**.

1.6 Identifiers

Identifiers are the names given to variables, functions, and other user-defined elements in a program. They help programmers refer to data and operations in a clear and meaningful way. Instead of working directly with values, programmers use identifiers to store and access information during program execution.

In simple terms, an identifier is the name used to identify something in a program. For example, when we create a variable to store a value, we give it a name so that we can use that value later in the program.

Example:

```
set age to 25
set totalMarks to 90
```

In the above statements, **age** and **totalMarks** are identifiers. They represent variables that store values which can later be used in calculations, conditions, or output statements.

Identifiers should be meaningful so that the program becomes easier to understand. For instance, a name like **totalMarks** clearly indicates what the variable represents, while a name like **x** may not provide enough information about its purpose.

To ensure that identifiers are valid and readable, the Mewar programming language follows certain rules.

Rules for identifiers:

- An identifier must start with a letter.
- It cannot be a keyword of the language.
- It cannot contain spaces between words.
- It may include letters and numbers.
- Special characters are generally not allowed unless specified by the language rules.

Examples of valid identifiers:

- age
- total1
- studentName

Examples of invalid identifiers:

• 1age (cannot start with a number)
• set (cannot use a keyword as an identifier)
• total marks (spaces are not allowed)

Choosing clear and meaningful identifiers is an important practice in programming because it improves readability and makes programs easier to understand and maintain.

1.7 Constants

Constants are values that remain fixed and do not change during the execution of a program. Unlike variables, whose values can be modified at different stages of a program, constants always keep the same value once they are defined or used.

Constants are commonly used to represent numbers, text, or collections of values that are not expected to change while the program runs. Using constants helps make programs clearer and more reliable because the programmer knows that those values will remain the same throughout execution.

In the Mewar programming language, constants can appear directly in statements without needing to be stored in a variable first. They represent literal values written in the program.

There are several types of constants used in Mewar.

Numeric constants represent numbers. These numbers can be integers or decimal values and are commonly used in calculations.

Examples of numeric constants:

```
10
-5
3.14
```

String constants represent text. Strings are always written inside quotation marks so that the interpreter understands they are text values rather than variable names.

Examples of string constants:

```
"Hello"
"Mewar"
```

List constants represent a collection of multiple values stored together in a single structure. Lists are written inside square brackets and can contain several values separated by commas.

Examples of list constants:

```
[1, 2, 3]
[10, 20, 30]
```

Constants are often used when assigning values to variables.

Example:

```
set name to "Mewar"
set marks to [80, 85, 90]
```

In this example, **"Mewar"** is a string constant and **[80, 85, 90]** is a list constant. These constant values are stored in the variables **name** and **marks** for later use in the program.

1.8 Operators

Operators are special symbols or keywords used to perform operations on values or variables. They allow programs to carry out calculations, compare values, and apply logical conditions. In simple terms, operators help the program process data and produce results.

In the Mewar programming language, operators are used in expressions to manipulate values stored in variables or constants. Different types of operators are used depending on the task being performed.

Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations such as addition, subtraction, multiplication, and division. These operators work with numeric values.

Common arithmetic operators include:

- Addition
- Subtraction

- Multiplication
- Division
- Modulo (remainder after division)

Example:

```
set total to 10 + 5

say total
```

```
Output

15
```

In this example, the + operator adds the values **10** and **5**, and the result is stored in the variable **total**.

Comparison Operators

Comparison operators are used to compare two values. The result of a comparison is usually **true** or **false**, depending on whether the condition is satisfied.

Common comparison operators include:

- Greater than
- Less than
- Equal to
- Not equal to

- Greater than or equal to
- Less than or equal to

Example:

```
Set age to 21

if age > 18
say "Adult"
end
```

```
Output

Adult
```

In this example, the program checks whether the value of **age** is greater than **18**. If the condition is true, the program prints **"Adult"**.

Logical Operators

Logical operators are used to combine multiple conditions in a single expression. They help the program make more complex decisions.

Common logical operators include:

- And
- Or

- not

Example:

```
if age > 18 and age < 60
say "Working age"
end
```

In this example, the program checks two conditions at the same time. The message **"Working age"** will be printed only if **age is greater than 18 and less than 60**.

Operators are essential in programming because they allow programs to perform calculations, evaluate conditions, and control the flow of execution.

1.9 Miscellaneous Examples

In the previous sections, we learned about variables, constants, operators, and basic program structure in the Mewar programming language. To understand these concepts better, let us look at a few simple examples. These examples demonstrate how different elements of the language work together in a program.

Example 1 — Simple Addition

```
set a to 5
set b to 3
```

```
set sum to a + b
say sum
```

Output:
8

Explanation:
In this program, two variables **a** and **b** are created and assigned the values **5** and **3**. The third statement adds the values of **a** and **b** using the + operator and stores the result in the variable **sum**. Finally, the **say** statement prints the value stored in **sum**, which is **8**.

Example 2 — Using Strings

```
set name to "Mewar"
say name
```

Explanation:
In this example, the variable **name** stores a string value **"Mewar"**. The **say** command prints the text stored in the variable. Strings must always be written inside quotation marks so the interpreter understands that they represent text.

Example 3 — Using a List

```
set numbers to [1, 2, 3]
say numbers[1]
```

```
Output:
1
```

Explanation:
In this example, a list named **numbers** is created containing three values: **1**, **2**, and **3**. The expression **numbers[1]** accesses the first element of the list. Since Mewar uses **1-based indexing**, the first position in the list contains the value **1**, which is printed as the output.

These examples show how variables, operators, strings, and lists can be used together to perform simple tasks in a program.

1.10 Review Questions

In this chapter, we learned the basic concepts of the Mewar programming language, including program structure, tokens, keywords, identifiers, constants, and operators. The following questions will help you review and test your understanding of the concepts covered in this chapter.

Short Answer Questions

1. What is a programming language?
2. What is the Mewar programming language?
3. What is a token in programming?
4. What are keywords in Mewar?

5. What are identifiers?

6. What is a constant?

7. What are operators and why are they used in programs?

Practice Questions

1. Write a simple Mewar program to print your name.

Example idea:
set name to " YourName "
say name

2. Create two variables and add their values.

Example idea:
set a to 10
set b to 20
set sum to a + b
say sum

3. Write a program to store a list of numbers and print the first element.

4. Create a variable that stores a string and display it using the **say** command.

These questions will help reinforce the fundamental concepts introduced in this chapter and prepare you for more advanced topics in the following chapters.

Chapter 2

Data Types and Variables

2.1 Introduction

In the previous chapter, we learned about the basic structure of a Mewar program, including how programs are written, how instructions are executed, and the role of tokens, keywords, identifiers, constants, and operators. These concepts form the basic building blocks of any program.

In this chapter, we will move one step further and learn about **data types and variables**, which are essential for storing and working with information inside a program. Almost every program needs to store data, process it, and display results. To do this effectively, we must understand how data is represented and managed.

In this chapter, we will understand the following concepts:

- What a data type is
- What a variable is
- How to declare a variable
- How to store values in variables
- Important rules and guidelines for using variables

Every program works with some form of data. For example, a program may work with numbers, text, or collections of values. These pieces of information must be stored somewhere so that the program can use them when needed.

Data types help the programming language understand **what kind of information is being stored**, while variables provide a **named location where that information can be kept**. Together, data types and variables form the foundation of programming because they allow programs to store, process, and manipulate data effectively.

Understanding these concepts will help you write clearer programs and prepare you for more advanced topics later in the Mewar programming language.

2.2 What is a Data Type

A **data type** defines the kind of value that a variable can store. In programming, different types of information must be handled in different ways. For example, numbers are used for calculations, while text is used to display messages or store names. Data types help the programming language understand what kind of data is being used so that it can process it correctly.

In simple words, a data type tells us **what type of information is stored in a variable**. It determines how the value can be used, what operations can be performed on it, and how it will be interpreted by the system.

For example, a program may work with different types of data such as:

- A number
- A piece of text
- A collection of values

Each of these represents a different kind of information. Numbers are typically used for calculations, text is used for messages or labels, and collections allow multiple values to be stored together.

In many traditional programming languages, programmers must explicitly specify the data type when declaring a variable. However, the Mewar programming language simplifies this process. Mewar automatically determines the data type based on the value assigned to the variable.

For example:

```
set age to 20
set name to "Mewar"
```

In the first statement, the value **20** is recognized as a number. In the second statement, **"Mewar"** is recognized as a string (text). Because Mewar automatically understands the data type, the programmer does not need to declare the type separately.

This feature makes the language easier for beginners to learn, while still allowing programs to work with different types of data efficiently.

2.3 Types of Data in Mewar

In programming, data can appear in different forms such as numbers, text, or collections of values. Each type of data is handled differently by the programming language. In Mewar, the type of data is automatically recognized based on the value assigned to a variable.

Understanding the different types of data helps programmers choose the correct format for storing and processing information.

2.3.1 Numeric Data Type

The **numeric data type** represents numbers. Numbers are commonly used in programs to perform mathematical calculations such as addition, subtraction, multiplication, and division.

Examples:

```
set age to 20
set price to 99
```

```
say age
say price
```

In the above example, **age** and **price** store numeric values. When the program runs, the **say** command prints these values.

Numbers can appear in different forms, such as:

• Positive numbers (10)
• Negative numbers (-5)
• Decimal numbers (3.14)

Numeric values are mainly used for calculations and mathematical operations.

Example:

set a to 5 set b to 3 set result to a + b
say result
Output: 8

Explanation:
In this program, two numeric variables **a** and **b** are created with the values **5** and **3**. The third statement adds these values using the + operator and stores the result in the variable **result**. Finally, the program prints the result, which is **8**.

2.3.2 String Data Type

A **string data type** represents text or a sequence of characters. Strings are used to store words, names, messages, or any other form of textual information in a program.

In the Mewar programming language, strings must always be written inside **quotation marks** so that the system can recognize them as text values. If quotation marks are not used, the interpreter may assume that the word is the name of a variable instead of a text value.

Example:

```
set name to "Mewar"
say name
```

Output:
Mewar

Explanation:
In this example, the variable **name** stores the text **"Mewar"**. The **say** command prints the value stored in the variable, which is the word **Mewar**.

Strings are commonly used to display messages to the user, store names, or represent textual data in a program.

For example:

set message to "Welcome to Mewar Language" say message
Output: Welcome to Mewar Language

It is important to remember that **quotation marks are required for string values**. Without quotation marks, the system may treat the word as a variable name instead of text.

2.3.3 Sequence Data Types

Sequence data types are used to store multiple values in an ordered structure. In a sequence, elements are arranged in a specific order, and each element can be accessed using its position, known as an index.

In the Mewar programming language, sequence data types allow programmers to store collections of values and work with them efficiently. These data types are very useful when handling groups of related data such as numbers, characters, or structured values.

The main sequence data types available in Mewar include **List, String, and Matrix**.

2.3.4 List

A **list** is used to store multiple values in a single variable. Lists are written inside square brackets, and values are separated by commas.

Example:

```
set numbers to [1, 2, 3]
say numbers
```

Output:
[1, 2, 3]

Each value inside a list has a position called an index. In Mewar, indexing starts from **1**.

Example:

```
set numbers to [1, 2, 3]
say numbers[1]
```

Output:
1

Lists are commonly used to store collections such as marks, values, or items.

2.3.5 String

A **string** is a sequence of characters used to represent text. Strings are always written inside quotation marks.

Example:

```
set name to "Mewar"
say name
```

Output:
Mewar

Each character in a string also has an index, allowing individual characters to be accessed if required.

Example:

```
set word to "Mewar"
say char 1 of word
```

Output:
M

Strings are useful for storing names, messages, and other textual information.

2.3.6 Matrix

A **matrix** is a two-dimensional sequence of numeric values arranged in rows and columns. It is useful for representing tables, grids, or mathematical data structures.

Example:

```
matrix A = [[1,2],[3,4]]
say A
```

Output:
[[1,2],[3,4]]

Elements inside a matrix can be accessed using row and column positions.

Example:

```
say A[1][2]
```

Output:
2

Matrices are commonly used in mathematical calculations, scientific computing, and data processing.

These sequence data types allow Mewar programs to handle multiple values efficiently while maintaining a clear and structured format.

A list stores multiple values in one variable.

Lists are written inside square brackets.

Example:

```
set numbers to [1, 2, 3] say numbers

Output:

[1, 2, 3]
```

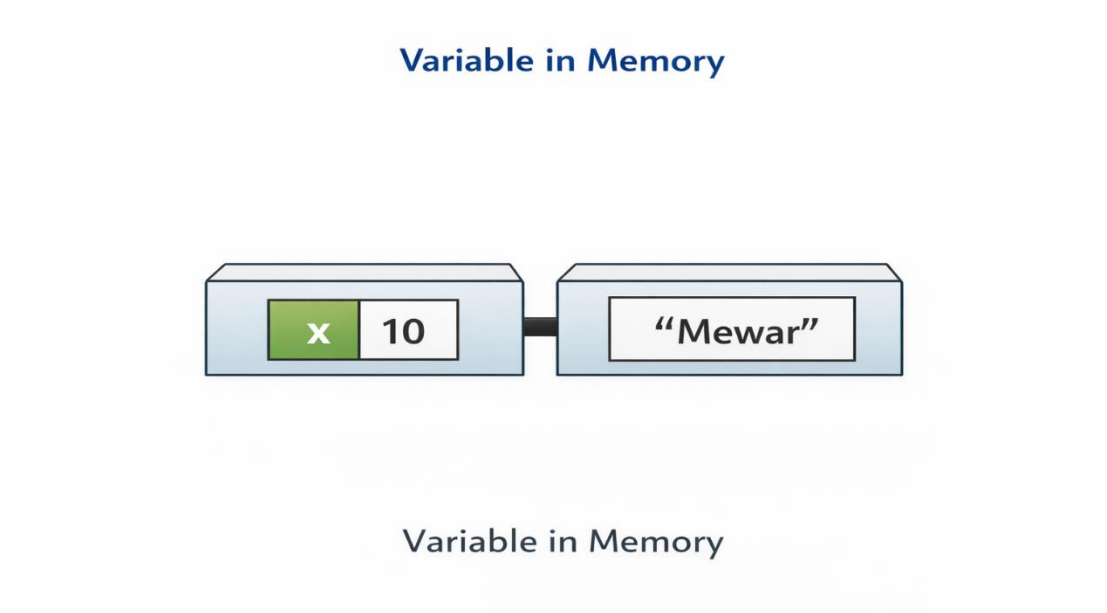
Each value inside a list has a position (index).

2.4 What is a Variable

A **variable** is a named storage location used to store data in a program. Variables allow programmers to store values that can be used later in calculations, conditions, or output statements. Instead of working directly with values, we give those values a name so that they can be easily accessed and reused throughout the program.

In simple terms, a variable acts like a container that holds information. The value stored inside a variable can be read or changed during program execution.

You can think of a variable as a **labeled box**. The label represents the name of the variable, and the box contains the value stored inside it.



Example:

```
set x to 10
```

In this example:

- **x** is the name of the variable.
- **10** is the value stored in the variable.

Whenever the program uses **x**, the system refers to the value stored in that variable. This means the program will treat **x** as the value **10** wherever it appears.

For example:

```
set x to 10
say x
```

```
Output:
10
```

Here, the **say** command prints the value stored in **x**.

Variables are very important in programming because they allow programs to store data, update values, and perform operations using that stored information.

2.5 Declaring a Variable

In the Mewar programming language, variables are created and assigned values using the **set** keyword. This process is called **declaring a variable**. Declaring a variable means creating a named storage location where a value can be stored and used in the program.

The syntax for declaring a variable in Mewar is simple and easy to understand.

Syntax:

```
set variableName to value
```

In this syntax, **variableName** represents the name of the variable, and **value** represents the data that will be stored in that variable.

Example:

```
set marks to 90
set city to "Delhi"
```

In the first statement, a variable named **marks** is created and the value **90** is stored in it. In the second statement, a variable named **city** is created and the text **"Delhi"** is stored in it.

In Mewar, the **set** keyword performs two tasks at the same time:

- It creates the variable.
- It assigns a value to that variable.

Unlike some traditional programming languages, Mewar does not require a separate declaration step to specify the data type of a variable. The interpreter automatically determines the type of data based on the value assigned.

This approach keeps the language simple and beginner-friendly, allowing programmers to focus more on program logic rather than complex declarations.

2.6 Storing Value in a Variable

A variable is used to store data that can be used later in a program. In the Mewar programming language, a variable can store different types of values depending on what the program needs. These values can represent numbers, text, collections of values, or even the result of a calculation.

The value stored in a variable can be:

- A number
- A string (text)
- A list of values
- The result of an expression or calculation

Example 1 — Storing a Number

```
set score to 100
```

In this example, the variable **score** stores the numeric value **100**.

Example 2 — Storing Text

```
set message to "Welcome"
```

Here, the variable **message** stores the text **"Welcome"**. Since it is text, it must be written inside quotation marks.

Example 3 — Storing a Calculation

```
set total to 50 + 25
say total
```

Output:
75

Explanation:
In this example, the expression **50 + 25** is calculated first. The

result of the calculation, which is **75**, is stored in the variable **total**. When the **say** command is executed, the program prints the stored value.

Storing values in variables allows programs to reuse data, perform calculations, and display results efficiently.

2.7 Updating a Variable

In programming, the value stored in a variable can be changed during the execution of a program. This process is called **updating a variable**. Updating allows a program to modify previously stored data and store a new value in the same variable.

When a variable is assigned a new value, the previous value is replaced by the new one. The variable name remains the same, but the value it stores changes.

Example:

```
set x to 5
set x to 20
say x
```

Output:
20

Explanation:
In the first statement, the variable **x** is created and assigned the value **5**. In the second statement, the variable **x** is assigned

a new value **20**. This new assignment replaces the previous value stored in **x**.

When the **say** command is executed, the program prints the current value of **x**, which is **20**.

Updating variables is very useful in programs where values change during execution, such as counting numbers in loops, updating totals in calculations, or storing new results after performing operations.

2.8 Points to Note

When working with variables in the Mewar programming language, there are several important rules and guidelines that programmers should remember. Following these points helps avoid errors and makes programs easier to read and understand.

1. **A variable must be created before using it.**
   A variable should always be defined using the **set** keyword before it is used in any operation or output statement. If a variable is used before it is created, the program may produce an error.

2. **Always use quotation marks for text values.**
   Text values must be written inside quotation marks. Without quotation marks, the interpreter may assume the text is a variable name instead of a string.

3. **Variable names cannot be keywords.**
   Reserved words of the language such as **set**, **say**, **if**, or **while** cannot be used as variable names because they already have special meanings in the language.

4. **Variable names cannot start with a number.**
   Identifiers must begin with a letter. Starting a variable name with a number will make it invalid.

5. **Variables may be case-sensitive.**
   In some systems, **age**, **Age**, and **AGE** may be treated as different variables. It is important to follow consistent naming conventions to avoid confusion.

6. **Use meaningful variable names.**
   It is always better to choose descriptive names that clearly represent the purpose of the variable. For example, **totalMarks** is more meaningful than using a short name like **t**.

2.9 Review Questions

In this chapter, we learned about **data types and variables** in the Mewar programming language. We studied how data is stored, how variables are declared, and how different types of data such as numbers, strings, and lists are used in programs. The following questions will help you review the concepts covered in this chapter.

Short Questions

1. What is a data type?

2. What is a variable?

3. How do you declare a variable in Mewar?

4. What is the difference between a number and a string?

5. What is a list?

Practice Questions

1. Write a program to store your age in a variable and print it.

Example idea:
set age to 20
say age

2. Create a variable that stores your city name and display it.

Example idea:
set city to "Delhi"
say city

3. Create a list containing three numbers and print the list.

Example idea:
set numbers to [5, 10, 15]
say numbers

4. Store the result of a calculation in a variable and print it.

Example idea:
set total to 25 + 15
say total

These questions will help reinforce your understanding of **data types, variables, and how values are stored in Mewar programs**.

Chapter 3

Input, Output, and Comments

3.1 Introduction

In the previous chapters, we learned the fundamental concepts of the Mewar programming language. These included variables, data types, and the basic structure of a program. We also learned how programs store values, perform operations, and display simple results.

In this chapter, we will learn how to make programs more useful and interactive. Most real programs need to communicate with the user. They must be able to accept information from the user, process that information, and display meaningful results.

In this chapter, we will learn how to:

• Take input from the user
• Display output on the screen
• Write comments inside a program

Input allows a program to receive information while it is running. Output allows the program to display results or messages to the user. Comments are notes written inside the program to explain what the code is doing.

These features make programs more interactive, easier to understand, and easier to maintain. By learning how to use input, output, and comments, you will be able to write programs that interact with users and clearly explain their purpose.

3.2 Input in Mewar

Input means taking data from the user while the program is running. Many programs need information from the user in order to perform tasks such as calculations, decision making, or storing data. This information is called **input**.

In the Mewar programming language, input is taken using the **ask** keyword. The **ask** command displays a message on the screen and waits for the user to enter a value. The value entered by the user is then stored in a variable so that the program can use it later.

Syntax:

set variableName to ask "Message to display"

In this syntax, the **ask** keyword shows a message to the user, and the value entered by the user is stored in the variable specified before it.

Example:

```
set number to ask "Enter a number:"
say number
```

In this example, the program first displays the message **"Enter a number:"**. The user enters a value, and that value is stored in the variable **number**. The **say** command then prints the value entered by the user.

Using input allows programs to become interactive because they can receive different values each time they are executed.

3.2.1 Example: Taking Input

Let us see a simple example of taking input from the user in the Mewar programming language. In this example, the program asks the user to enter a number and then displays that number on the screen.

Example:

```
set num1 to ask "Enter the first number:"
say num1
```

Explanation:
In the first statement, the **ask** keyword displays the message **"Enter the first number:"** on the screen. The program then waits for the user to enter a value. Whatever value the user enters is stored in the variable **num1**.

The second statement uses the **say** command to print the value stored in **num1**.

What happens during execution:

1. The message **"Enter the first number:"** appears on the screen.

2. The user enters a value.

3. The entered value is stored in the variable **num1**.

4. The program prints the stored value using the **say** command.

This example shows how input can be taken from the user and used in a program.

3.2.2 Example: Taking Two Inputs

In many programs, we need to take more than one input from the user. The following example shows how a program can take two numbers from the user, add them, and display the result.

Example:

```
set num1 to ask "Enter the first number:"
set num2 to ask "Enter the second number:"
set sum to num1 + num2
say "The Royal Sum is: " + sum
```

Explanation:
In this program, the first **ask** statement displays the message **"Enter the first number:"** and waits for the user to enter a value. The value entered is stored in the variable **num1**.

The second **ask** statement displays the message **"Enter the second number:"** and stores the user's input in the variable **num2**.

The third statement adds the values stored in **num1** and **num2**, and the result is stored in the variable **sum**.

Finally, the **say** command displays the message **"The Royal Sum is:"** along with the calculated result.

Summary of the steps:

- The **ask** command displays a prompt to the user.
- The user enters values which are stored in variables.
- The program adds the two values.
- The result is displayed on the screen.

3.3 Output in Mewar

Output means displaying information or results to the user. After a program performs calculations or processes data, the results need to be shown so that the user can see them. This process of displaying information is called **output**.

In the Mewar programming language, output is mainly produced using the **say** command. The **say** keyword tells the program to print a value, message, or result on the screen.

The value displayed can be a number, a variable, or a piece of text.

Syntax:

```
say value
```

The **value** can be a variable, a string, or the result of an expression.

Example:

```
say "Welcome to Mewar"
```

Output:

```
Welcome to Mewar
```

Example with a variable:

```
set x to 100
say x
```

Output:

```
100
```

The **say** command is one of the most commonly used commands in Mewar because it allows the program to communicate with the user by displaying messages and results.

3.3.1 Printing Text

Printing text means displaying a message or sentence on the screen. In the Mewar programming language, text messages are printed using the **say** command. Text values must always be written inside quotation marks so that the interpreter recognizes them as strings.

Example:

say "Welcome to Mewar"
Output:
Welcome to Mewar

Explanation:
In this example, the **say** command displays the text **"Welcome to Mewar"** on the screen. Since the message is written inside quotation marks, the system understands that it is a string value and prints it exactly as written.

Printing text is commonly used to display messages, instructions, or information to the user during program execution.

3.3.2 Printing Variables

In addition to printing text messages, a program can also display the value stored in a variable. This allows the program to show results of calculations or information that has been stored during execution.

In the Mewar programming language, the **say** command is used to print the value of a variable.

Example:

set x to 100 say x
Output: 100

Explanation:
In this example, the variable **x** is created and assigned the value **100**. When the **say** command is executed, the program prints the value stored in **x**.

Printing variables is very useful when displaying results of calculations or showing stored data to the user.

3.3.3 Printing Mixed Output

Sometimes a program needs to display both **text and variable values together**. This type of output is called **mixed output**.

It allows the program to combine messages with stored data and display them in a single line.

In the Mewar programming language, mixed output can be created by combining a **string** with a **variable** using the **+ operator**.

Example:

```
set name to "Mewar"
say "Hello " + name
```

Output:
Hello Mewar

Explanation:
In this example, the variable **name** stores the text **"Mewar"**. The **say** command combines the string **"Hello "** with the value stored in **name**. The **+ operator** joins the text and the variable value, and the program prints the combined result.

Mixed output is useful when programs need to display personalized messages or combine text with calculated values.

3.4 Complete Example Program

Let us now look at a complete example program that uses **input, calculation, and output** together. This program asks the user to enter two numbers, adds them, and displays the result.

Example Program:

Program to add two numbers

```
set num1 to ask "Enter the first number:"
set num2 to ask "Enter the second number:"
```

```
set sum to num1 + num2
```

```
say "The Royal Sum is: " + sum
```

Explanation:

This program performs the following steps:

1. The program first asks the user to enter the first number. The value entered is stored in the variable **num1**.

2. It then asks the user to enter the second number. The value entered is stored in the variable **num2**.

3. The program adds the two numbers using the **+ operator** and stores the result in the variable **sum**.

4. Finally, the **say** command displays the message **"The Royal Sum is:"** along with the calculated result.

This example shows how input, processing, and output work together in a simple program.

3.5 Comments in Mewar

Comments are notes written inside a program to explain what the code is doing. They are meant for the programmer or for anyone reading the code later. Comments help make programs easier to understand and maintain.

Comments are **ignored during program execution**, which means they do not affect how the program runs. The interpreter skips comments and only executes the actual instructions.

In the Mewar programming language, comments are written using the **# symbol** at the beginning of the comment line.

Syntax example:

 # This is a comment

Example:

 # This program calculates sum
set a to 5 set b to 10 say (a + b)

Explanation:
The first line is a comment that explains the purpose of the program. The interpreter ignores this line when running the

program. The remaining lines perform the actual task of adding two numbers and displaying the result.

Comments are very useful for documenting programs, explaining logic, and helping other programmers understand the code more easily.

3.6 Points to Remember

When working with input, output, and comments in the Mewar programming language, it is important to remember a few key points. These guidelines help ensure that programs work correctly and remain easy to read and understand.

1. **ask is used for input.**
   The **ask** keyword is used to take input from the user while the program is running.

2. **Input must be stored in a variable using set.**
   The value entered by the user should always be stored in a variable so that the program can use it later.

3. **say is used for output.**
   The **say** command displays messages, variable values, or results on the screen.

4. **Strings must be written inside quotation marks.**
   Text values should always be enclosed in quotation marks so that the interpreter recognizes them as strings.

5. **Comments improve readability.**
   Comments help explain what the program is doing, making it easier for others to understand the code.

6. **Always write meaningful prompt messages.**
   When using **ask**, the message displayed to the user should clearly explain what information needs to be entered.

3.7 Review Questions Short Questions

1. What is input?

2. Which keyword is used for input in Mewar?

3. How do you print output?

4. What is the purpose of comments?

5. Why is ask used with set?

Practical Questions

1. Write a program to ask the user’s name and print it.

2. Write a program that asks the user for two numbers and prints their sum.

Chapter 4

Operators in Mewar

4.1 Introduction

Operators are symbols or keywords used to perform operations on data. They allow programs to manipulate values, perform calculations, and evaluate conditions. Without operators, it would be difficult for a program to process data or produce meaningful results.

In the Mewar programming language, operators play an important role in expressions. An **expression** is a combination of values, variables, and operators that produces a result.

Operators allow programmers to perform different kinds of tasks within a program. Using operators, a program can perform mathematical calculations, compare values, combine logical conditions, and update stored values.

In Mewar, operators allow us to:

• Perform calculations such as addition, subtraction, multiplication, and division.
• Compare values to check whether conditions are true or false.
• Make decisions in control structures such as **if statements**.
• Update the values stored in variables.

• Work with logical conditions when multiple comparisons are involved.

Almost every program uses operators in some form. Whether performing simple arithmetic or evaluating complex conditions, operators are essential tools that help programs process data and control the flow of execution.

4.2 Various Operations Available in Mewar

In programming, different types of operations are

used to manipulate data and control how a program

behaves. Each operation performs a specific task,

such as performing calculations, comparing values,

or combining logical conditions.

The Mewar programming language supports

several types of operations that help programmers

work with data efficiently. These operations allow

programs to process information, make decisions,

and update values during execution.

The main types of operations available in Mewar

include:

• **Arithmetic operations** – used to perform mathematical calculations such as addition, subtraction, multiplication, and division.

• **Relational (comparison) operations** – used to compare two values and determine whether a condition is true or false. These operations are commonly used in decision-making statements.

• **Logical operations** – used to combine multiple conditions and evaluate complex logical expressions.

• **Assignment operations** – used to assign or update values stored in variables.

• **Increment and decrement operations** – used to increase or decrease the value of a variable, usually by one.

• **Special operations** – used for specific tasks or special features supported by the language.

• **Type casting** – used to convert one data type into another when necessary.

Each type of operation serves a specific purpose and helps the programmer perform different tasks while writing programs. Understanding these operations is important because they are widely used in calculations, conditions, loops, and many other programming situations.

4.3 Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations in a program. These operators work with numeric values and allow programs to perform operations such as addition, subtraction, multiplication, division, and finding the remainder.

In the Mewar programming language, arithmetic operators are commonly used in expressions to calculate results and store them in variables or display them as output.

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo	a % b

Example:

```
set a to 10
set b to 3
```

```
say (a + b)
say (a - b)
say (a * b)
say (a / b)
say (a % b)
```

Output:

```
13
7
30
3.33333
1
```

Explanation:
In this example, the variables a and b store the values 10 and 3. The program then uses different arithmetic operators to perform calculations.

The + operator adds the two values.
The - operator subtracts the second value from the first.
The * operator multiplies the values.
The / operator divides the first value by the second.
The % operator calculates the remainder when the first value is divided by the second.

Arithmetic operators are essential in programming because they allow programs to perform calculations and process numeric data.

4.4 Relational Operators

Relational operators are used to compare two values. These operators check the relationship between values and return a result that is either true or false depending on whether the condition is satisfied.

Relational operators are commonly used in decision-making statements such as if conditions, where the program must determine whether a condition is correct before executing a block of code.

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Example:

```
set a to 10
set b to 5
```

```
say (a > b)
say (a == b)
say (a != b)
```

Explanation:
In this example, the variables a and b store the values 10 and 5. The relational operators compare these values.

The expression a > b checks whether 10 is greater than 5, which is true.
The expression a == b checks whether 10 is equal to 5, which is false.

The expression a != b checks whether 10 is not equal to 5, which is true.

Output will be either true or false, depending on the result of the comparison.

4.5 Logical Operators

Logical operators are used to combine or modify conditions in a program. They are commonly used when a decision depends on more than one condition. Logical operators help programs evaluate complex expressions and determine whether the combined condition is true or false.

In the Mewar programming language, logical operators are often used inside **if statements** and other control structures.

Operator	**Meaning**
and	Logical AND
or	Logical OR
not	Logical NOT

The **and** operator returns true only if **both conditions are true**.
The **or** operator returns true if **at least one condition is true**.
The **not** operator reverses the result of a condition. If a condition is true, **not** makes it false, and if it is false, **not** makes it true.

Example:

```
set age to 20
```

```
if age > 18 and age < 60
say "Working age"
end
```

Explanation:
In this example, the variable **age** is assigned the value **20**. The condition checks two things: whether the age is greater than **18** and whether it is less than **60**. Both conditions are combined using the **and** operator.

Since **20 is greater than 18 and less than 60**, both conditions are true, so the program prints **"Working age"**.

4.8 Special Operators

The Mewar programming language also provides some **special operators** that allow programmers to perform advanced operations on data. These operators are useful when working with collections, strings, or memory references.

Membership Operator

Membership operators are used to check whether a value exists inside a collection such as a list. They help determine if an element belongs to a particular list.

Example:

```
set list to [1, 2, 3]
```

```
say 2 in list
say 5 not in list
```

Explanation:
The expression **2 in list** checks whether the value **2** exists inside the list. Since **2** is present, the result will be **true**.
The expression **5 not in list** checks whether the value **5** is not present in the list. Since **5** does not exist in the list, the result will also be **true**.

String Operations

String operations allow programs to combine or manipulate text values. In Mewar, the **+ operator** can be used to join two strings together. This process is called **string concatenation**.

Example:

```
set name to "Mewar"
say name + " Language"
```

Output:

```
Mewar Language
```

Explanation:
The **+ operator** combines the value stored in the variable

name with the text **" Language"**, producing a single combined string.

Pointer Operators

Pointer operators are used to work with memory addresses and references. A pointer stores the address of another variable instead of storing a value directly.

Example:

```
set x to 10
set p to address x
say value p
```

Explanation:
In this example, the variable **x** stores the value **10**. The statement **address x** returns the memory address of **x**, which is stored in the pointer variable **p**. The expression **value p** accesses the value stored at that address, which is **10**.

Special operators provide powerful capabilities for working with lists, text data, and memory references in Mewar programs.

4.9 Operator Precedence and Order

Operator precedence determines the order in which operations are evaluated in an expression. When an expression contains multiple operators, the program does not always evaluate

them from left to right. Instead, certain operators are given higher priority and are evaluated first.

Understanding operator precedence is important because it ensures that expressions are calculated correctly. If the order of operations is not clear, the result of a calculation may be different from what the programmer expects.

In Mewar, operators follow a specific order of precedence.

Order of precedence:

1. Parentheses ( )
2. Multiplication, Division, Modulo (*, /, %)
3. Addition, Subtraction (+, -)
4. Relational operators (>, <, >=, <=, ==, !=)
5. Logical operators (and, or, not)

Operations inside parentheses are evaluated first. After that, multiplication, division, and modulo operations are performed. Then addition and subtraction are evaluated. Relational operators are evaluated next, followed by logical operators.

Example:

```
set result to 10 + 5 * 2
say result
```

Output:
20

Explanation:
In this expression, both addition and multiplication appear. According to operator precedence, multiplication is performed before addition. Therefore, the program first calculates **5 * 2**, which gives **10**. Then it adds **10 + 10**, producing the final result **20**.

If parentheses are used, the order of evaluation can be changed. For example:

```
set result to (10 + 5) * 2
say result
```

Output:
30

Here, the expression inside the parentheses **(10 + 5)** is evaluated first, giving **15**. Then **15 * 2** is calculated, producing the result **30**.

4.10 Type Casting

Type casting means converting a value from one data type to another. Sometimes a program needs to change the form of

data so that it can be used in calculations, comparisons, or output. For example, a number stored as text may need to be converted into a numeric value before performing arithmetic operations.

In the Mewar programming language, type casting allows programmers to explicitly convert values from one data type to another when necessary.

Mewar supports **explicit type casting**, which means the programmer manually specifies the conversion.

4.10.1 Explicit Type Casting

In Mewar, type casting is done using **function-style syntax**. These functions convert values into a different data type.

The main casting functions are:

number(value)
string(value)
boolean(value)

The **number()** function converts a value into a numeric type.
The **string()** function converts a value into text.
The **boolean()** function converts a value into a logical value such as true or false.

Examples

Number Conversion

```
set x to number("123")
say x + 1
```

Explanation:
In this example, the string **"123"** is converted into a number using the **number()** function. After conversion, the program adds **1** to the value and prints the result.

String Conversion

```
set text to string(100)
say "Value: " + text
```

Explanation:
The **string()** function converts the numeric value **100** into a text value. This allows it to be combined with other text in the output.

Boolean Conversion

```
set flag to boolean(0)
say flag
```

Explanation:
The **boolean()** function converts the value **0** into a Boolean value. In most cases, **0 becomes false**, while non-zero values become **true**.

Type casting is useful when working with different types of data, especially when input values, calculations, and text operations need to interact with each other.

4.12 Miscellaneous Examples

The following examples demonstrate how different types of operators and operations can be used together in Mewar programs. These examples combine arithmetic, logical, relational, and membership operations.

Example 1 — Combined Operations

```
set a to number("10")
set b to 5

set result to (a + b) * 2
say result
```

Output:

```
30
```

Explanation:
In this example, the value **"10"** is first converted into a number using the **number()** function. The variables **a** and **b** are then added together. The result of the addition is multiplied by **2**, and the final result is stored in the variable **result**. The **say** command displays the calculated value.

Example 2 — Logical and Relational Operations

```
set age to 25

if age > 18 and age < 60
say "Valid"
end

Output

Valid
```

Explanation:
In this example, the program checks whether the value of **age** satisfies two conditions: it must be greater than **18** and less than **60**. The **and** logical operator combines these two relational conditions. Since **25** satisfies both conditions, the program prints **"Valid"**.

4.13 Practice Questions

Short Answer

1. What is an operator?
2. What is the difference between relational and logical operators?
3. What is operator precedence?
4. What is type casting?
5. What is explicit type casting?

Practical Questions

1. Write a program to add two numbers.
2. Compare two values and print the result.
3. Use logical operators in a condition.
4. Convert a string to number and perform addition.
5. Check if an element exists in a list.

Chapter 5

Control Statements in Mewar

5.1 Introduction

Control statements determine how the flow of execution moves within a program. By default, a program executes instructions **line by line from top to bottom**. However, in many situations, a program needs to make decisions, repeat certain actions, or change the order of execution. Control statements make this possible.

Control statements allow a program to behave differently depending on conditions and requirements. They help programmers control how and when specific parts of a program should run.

Using control statements, a program can:

• Make decisions based on conditions
• Repeat actions multiple times
• Control the execution of instructions dynamically

In the Mewar programming language, control statements are divided into several categories.

Conditional statements are used to make decisions in a program. They allow the program to execute certain

instructions only when specific conditions are true. Examples include **if, elif, else, and switch**.

Looping statements are used to repeat a block of code multiple times until a condition is satisfied or a certain limit is reached. Examples include **while, for, repeat, do-while, and step**.

Control flow statements are used to alter the normal flow of execution inside loops or conditions. In Mewar, the **stop** statement can be used to terminate execution when needed.

These control statements make programs more powerful by allowing them to respond to different situations and perform repetitive tasks efficiently.

5.2 If Statement

The **if statement** is used to make decisions in a program. It allows a block of code to execute only when a specified condition is true. If the condition is false, the block of code inside the **if statement** is skipped.

Syntax:

```
if condition then
    statements
end
```

Example:

```
set age to 20
if age > 18 then
    say "Adult"
end
```

Explanation:

• **set age to 20** → stores the value **20** in the variable **age**.

• **if age > 18 then** → the program checks whether the condition is true.

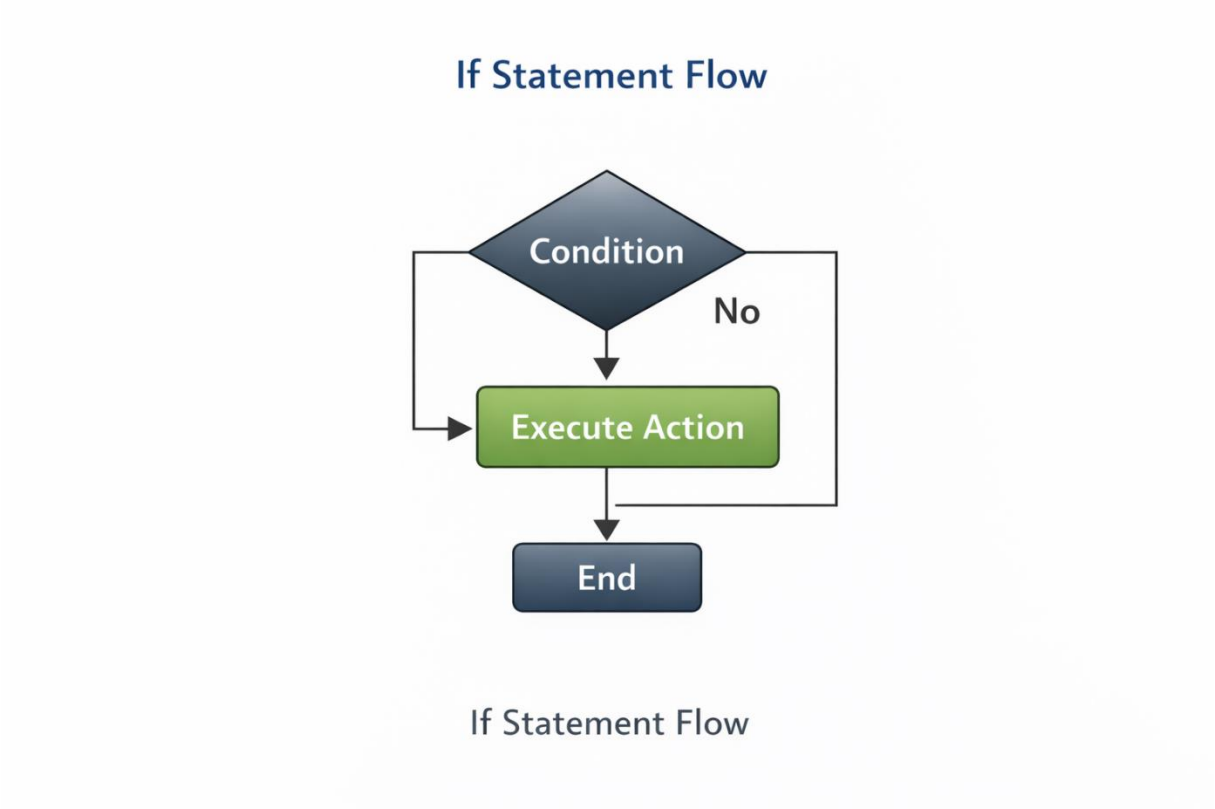
• Since **20 > 18**, the condition is true.

• **say "Adult"** → this statement is executed because the condition is satisfied.

• **end** → marks the end of the **if** block.

Output:

```
Adult
```



If Statement Flow

5.3 If…Elif Statement

The **if… elif statement** is used when a program needs to check **multiple conditions**. It allows the program to test several conditions one after another and execute the block of code associated with the first condition that evaluates to true.

If the first condition is false, the program checks the next condition using **elif**. This process continues until a true condition is found.

Syntax:

```
if condition1 then
    statements
```

```
elif condition2 then
   statements
end
```

Example:

```
set marks to 75

if marks > 90 then
   say "A"
elif marks > 60 then
   say "B"
end
```

Explanation:

• **marks = 75** → the variable **marks** stores the value **75**.
• **First condition: 75 > 90** → this condition is false.
• **Second condition: 75 > 60** → this condition is true.
• Since the second condition is true, the program executes **say "B"**.

```
Output:

B
```

5.4 If…Elif…Else Statement

The **if… elif …else statement** is used when a program needs to check multiple conditions and also provide a **default action** if none of the conditions are true. The **else block**

ensures that some code will always execute when all previous conditions fail.

The program first checks the **if condition**. If it is false, it checks the **elif condition**. If none of the conditions are true, the **else block** is executed.

Syntax:

```
if condition then
    statements
elif condition then
    statements
else
    statements
end
```

Example:

```
set num to -5

if num > 0 then
    say "Positive"
elif num == 0 then
    say "Zero"
else
    say "Negative"
end
```

Explanation:

• **num = -5** → the variable **num** stores the value **-5**.
• **First condition: num > 0** → this condition is false.
• **Second condition: num == 0** → this condition is also false.
• Since both conditions are false, the **else block** executes.
• The program prints **"Negative"**.

Output:

Negative

5.5 Nested If-Else

A **nested if-else** occurs when an **if statement is placed inside another if statement**. This structure is useful when a decision depends on another condition. The program first checks the outer condition, and if it is true, the inner condition is then evaluated.

Example:

```
set age to 25

if age > 18 then
    if age < 60 then
        say "Working"
    else
        say "Senior"
    end
```

```
else
   say "Minor"
end
```

Explanation:

• **age = 25** → the variable **age** stores the value **25**.
• **Outer condition: age > 18** → this condition is true, so the program enters the first block.
• **Inner condition: age < 60** → this condition is also true.
• Since both conditions are satisfied, the program prints **"Working"**.

If the inner condition had been false, the program would have printed **"Senior"**. If the outer condition had been false, the program would have printed **"Minor"**.

5.6 Switch Case Statement

The **switch case statement** is used when a program needs to choose one option from multiple possible values. Instead of writing many **if…elif statements**, a switch statement allows the program to match a value with predefined cases and execute the corresponding block of code.

Syntax:

```
switch value
   case option1
      statements
```

```
    case option2
        statements
    default
        statements
end
```

Example:

```
set day to 2

switch day
    case 1
        say "Monday"
    case 2
        say "Tuesday"
    default
        say "Other"
end
```

Explanation:

• **day = 2** → the variable **day** stores the value **2**.
• The **switch statement** checks the value of **day**.
• **case 1** checks if the value is **1**, which is false.
• **case 2** checks if the value is **2**, which is true.
• Since the condition matches **case 2**, the program executes **say "Tuesday"**.
• The **default case** runs only if none of the cases match.

Output:

Tuesday

5.7 Looping in Mewar

Loops are used to repeat a block of code multiple times. Instead of writing the same statements again and again, a loop allows the program to execute the same set of instructions repeatedly until a condition is satisfied.

Loops are very useful when performing tasks such as counting numbers, processing lists of data, or repeating calculations. They help make programs shorter, more efficient, and easier to manage.

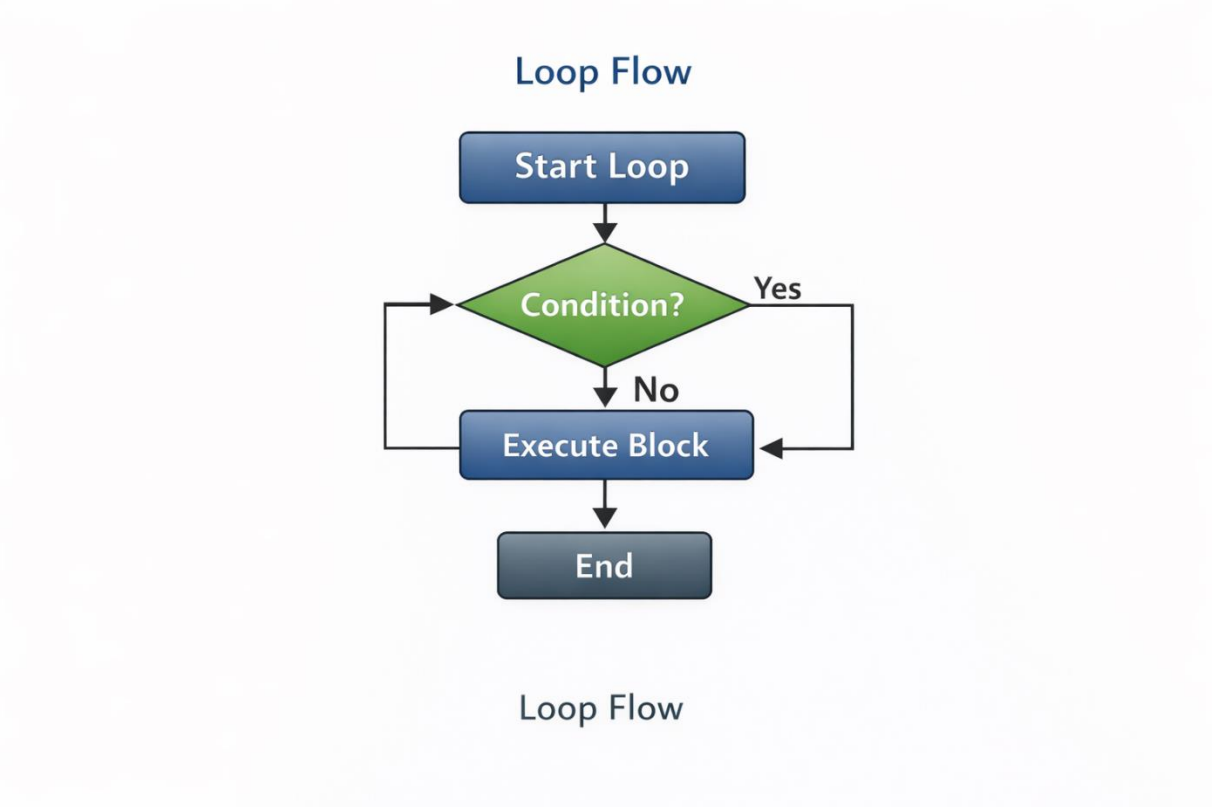
In the Mewar programming language, several types of loops are available. Each type is designed for a specific situation.

Types of loops in Mewar include:

• **while loop** – repeats a block of code as long as a specified condition is true.
• **for loop** – repeats a block of code for a fixed number of times or over a sequence of values.
• **repeat loop** – repeats a block of code until a certain condition is met or for a specified number of repetitions.
• **do-while loop** – executes the block of code at least once before checking the condition.

• **step loop** – used to control how the loop counter increases or decreases during iteration.

Loops are an important part of programming because they allow programs to perform repetitive tasks efficiently without duplicating code.



Loop Flow

5.8 While Loop

The **while loop** is used to repeat a block of code as long as a specified condition remains true. The program checks the condition before each iteration of the loop. If the condition is true, the loop continues to execute. When the condition becomes false, the loop stops.

Syntax:

```
while condition then
   statements
end
```

Example:

```
set i to 1

while i <= 3 then
   say i
   set i to i + 1
end
```

Explanation:
In this program, the variable **i** is initialized with the value **1**. The loop runs while the condition **i <= 3** is true. Inside the loop, the program prints the value of **i** and then increases its value by **1**.

Execution Steps:

Step	**i**	**Output**
1	1	1
2	2	2
3	3	3
4	4	Loop stops

5.9 For Loop

The **for loop** is used to repeat a block of code for a fixed number of times or to iterate over a sequence of values. It is commonly used when the number of iterations is known in advance.

In the Mewar programming language, the **for loop** works with a sequence generated using the **step() function**. The loop variable takes each value from the sequence and executes the statements inside the loop.

Syntax:

```
for i in step(start, end, step) then
    statements
end
```

Here:
• **start** is the starting value.
• **end** is the final value of the sequence.
• **step** determines how much the value increases in each iteration.

Example:

```
for i in step(1,5,1) then
    say i
end
```

Explanation:

• **step(1,5,1)** generates the sequence **[1, 2, 3, 4, 5]**.
• The loop variable **i** takes each value from the sequence one by one.
• For each value, the statement **say i** is executed.

```
Output:

1
2
3
4
5
```

5.10 Step Statement

The **step statement** is used with the **for loop** to control the range and the increment of values during iteration. It allows the programmer to specify the starting value, the ending value, and how much the value should increase after each iteration.

The **step() function** generates a sequence of numbers that the loop variable will follow.

Syntax:

```
step(start, end, step)
```

Where:
• **start** is the starting value of the sequence.
• **end** is the final value of the sequence.

• **step** determines how much the value increases after each iteration.

Example:

```
for i in step(1,10,2) then
    say i
end
```

Explanation:
In this example, the **step(1,10,2)** function generates a sequence starting from **1** and increasing by **2** each time until it reaches **10**.

The generated sequence is:
1, 3, 5, 7, 9

The **for loop** prints each value in this sequence.

Output:

```
1
3
5
7
9
```

5.11 Do-While Loop

The **do-while loop** is used when a block of code must run **at least once** before checking a condition. Unlike the **while loop**, which checks the condition first, the **do-while loop** executes the statements first and then checks the condition.

If the condition is true, the loop continues. If the condition becomes false, the loop stops.

Syntax:

```
do
   statements
while condition
```

Example:

```
set i to 1

do
   say i
   set i to i + 1
while i <= 3
```

Explanation:

- **i = 1** → the variable **i** is initialized with the value **1**.
- The **do block** executes first, printing the value of **i**.
- After executing the statements, the condition **i <= 3** is

checked.
• If the condition is true, the loop runs again.

Execution Steps:

Step	i	Output
1	1	1
2	2	2
3	3	3
4	4	Loop stops

When **i** becomes **4**, the condition **i <= 3** becomes false, so the loop stops executing.

5.12 Repeat Loop

The repeat loop is used to execute a block of code a fixed number of times. It is useful when the number of repetitions is already known. Instead of writing the same statement multiple times, the repeat loop allows the program to perform the task automatically.

Syntax:

```
repeat n then
    statements
end
```

Here, n represents the number of times the loop should execute.

Example:

```
repeat 3 then
    say "Hello"
end
```

Explanation:

• The repeat loop is set to run 3 times.
• During each iteration, the statement say "Hello" is executed.
• As a result, the message "Hello" is printed three times.

Output:
Hello Hello Hello

5.13 Nested Loop

A **nested loop** is a loop that is placed inside another loop. The outer loop runs first, and for each iteration of the outer loop, the inner loop executes completely.

Nested loops are commonly used when working with **tables, grids, matrices, or combinations of values**.

Example:

```
for i in step(1,2,1) then
    for j in step(1,2,1) then
        say i + " " + j
    end
end
```

Explanation:

• The **outer loop** controls the value of **i**.
• The **inner loop** controls the value of **j**.
• For each value of **i**, the inner loop runs completely from start to end.

Execution steps:

• When **i = 1**, the inner loop runs for **j = 1 and j = 2**.
• When **i = 2**, the inner loop again runs for **j = 1 and j = 2**.

Output:
1 1 1 2 2 1 2 2

This happens because the inner loop runs fully for each iteration of the outer loop.

5.14 Stop Statement (Break)

The **stop statement** is used to immediately terminate a loop before it finishes its normal execution. When the program encounters the **stop** statement inside a loop, the loop ends instantly and the program continues executing the next statements after the loop.

This statement is useful when a specific condition occurs and the program no longer needs to continue the loop.

Example:

```
for i in step(1,10,1) then
    if i == 5 then
        stop
    end
    say i
end
```

Explanation:

- The **for loop** generates values from **1 to 10**.
- For each value of **i**, the program checks the condition **i == 5**.
- When **i becomes 5**, the condition becomes true.
- The **stop statement** immediately terminates the loop.
- Therefore, the program prints only the numbers before **5**.

Output:
1 2 3 4

5.15 Miscellaneous Examples

The following examples demonstrate how control statements, loops, and conditions can be used together in Mewar programs.

Example 1 — Even Numbers

```
for i in step(1,10,1) then
    if i % 2 == 0 then
        say i
    end
end
```

Explanation:
This program generates numbers from **1 to 10** using the **for loop**. Inside the loop, the condition **i % 2 == 0** checks whether the number is even. The modulo operator **%** returns the remainder after division. If the remainder is **0**, the number is even and it is printed.

Output:
2 4 6 8 10

Example 2 — Input Based Decision

```
set num to ask "Enter number:"
if num > 0 then
    say "Positive"
else
    say "Negative"
end
```

Explanation:
This program asks the user to enter a number. The **if condition** checks whether the number is greater than **0**. If the condition is true, the program prints **"Positive"**. Otherwise, it prints **"Negative"**.

5.16 Points to Remember

When working with control statements and loops in the Mewar programming language, it is important to remember a few key

points. These guidelines help ensure that programs behave correctly and are easier to read and understand.

• **if** is used for decision making when a program needs to execute code based on a condition.

• **elif** is used to check multiple conditions when the first condition is false.

• **else** handles the default case when none of the previous conditions are true.

• **for loop** is commonly used with the **step() function** to iterate over a sequence of values.

• **while loop** continues running as long as its condition remains true.

• **repeat loop** executes a block of code a fixed number of times.

• **stop statement** immediately exits a loop when a specific condition occurs.

• **continue statement** skips the current iteration of a loop and moves to the next one.

• Always close conditional and looping blocks with the **end** keyword.

5.17 Practice Questions

Short Answer

1. What are control statements?
2. Difference between if and switch?
3. What is a loop?
4. What is stop statement?
5. What is continue?

Practical Questions

1. Print numbers from 1 to 10
2. Check even or odd
3. Print table of a number
4. Use switch to print day

CHAPTER 6

LIST, STRING AND MATRIX

6.1 INTRODUCTION

In the Mewar programming language, programs often need to work with groups of data rather than single values. To manage such data efficiently, Mewar provides **advanced data structures** that allow multiple values to be stored and manipulated easily.

These data structures help programmers organize information, perform operations on collections of data, and solve more complex problems. They make programs more flexible and efficient when dealing with large or structured datasets.

The main advanced data structures used in Mewar include:

• **List** → used to store multiple values in a single variable.
• **String** → used to store and manipulate text data.
• **Matrix** → used to store two-dimensional numerical data arranged in rows and columns.

Using these structures, a program can handle collections of values, process text information, and perform mathematical operations on structured data.

These features make the Mewar language more powerful and flexible, allowing programmers to build programs that can solve real-world problems efficiently.

6.2 LIST (COLLECTION OF VALUES)

Definition

A **list** is a collection of multiple values stored in a single variable. Lists allow programmers to group related values together and manage them easily. Instead of creating many variables to store individual values, a list stores them in a single structure.

Example:

```
set nums to [1,2,3,4]
```

In this example, the variable **nums** contains four values: **1, 2, 3, and 4**.

Lists are extremely useful when working with collections of data such as numbers, marks of students, product prices, or any group of related values.

6.2.1 Indexing (Important)

Each element in a list has a position called an **index**. The index allows the program to access specific elements from the list.

Mewar uses **1-based indexing**, which means the first element of a list starts at position **1**, not 0.

Example:

```
set nums to [1,2,3,4]
```

```
say nums[1] # 1
say nums[2] # 2
```

Explanation:

```
nums = [1,2,3,4]
```

Position → Value
1 → 1
2 → 2
3 → 3
4 → 4

So nums[1] returns 1, nums[2] returns 2, and so on.

6.2.1 Nested List (Advanced)

A list can also contain other lists inside it. This type of structure is called a **nested list**.

Example:

```
set grid to [[1,2], [3,4]]
```

This structure can be visualized as:

[
[1,2],
[3,4]
]

This works like a **table** with rows and columns.

Accessing values:

```
say grid[1][2] # 2
```

Explanation:

grid[1] → [1,2]
grid[1][2] → second element of first list → **2**

Nested lists are useful for representing **tables, grids, matrices, or structured data**.

6.3 List Operations

Mewar provides several built-in operations to modify and manage lists.

6.3.1 Append

The **append operation** is used to add a new value to the **end of a list**. It is one of the most common list operations because

programs often need to **build a list step by step while the program runs**.

When **append** is used, the new value is placed **after the last element of the list**, increasing the size of the list by one.

Example:

```
set roster to []
say "Building the roster..."

append 1 to roster
append 2 to roster

say "The final list is: " + roster
```

Explanation:

First, an **empty list** named **roster** is created.

roster = []

The program then prints a message:

Building the roster...

Next, the statement

append 1 to roster

adds the value **1** to the list.

roster = [1]

After that, the statement

append 2 to roster

adds the value **2** to the end of the list.

roster = [1,2]

Finally, the **say** statement prints the complete list.

Output:
Building the roster... The final list is: 1,2

Important Points about Append

• append always adds the value at the end of the list
• It does not replace existing elements
• It increases the size of the list by one

Example of list growth:

Start:

[]

After append 5:

[5]

After append 10:

[5,10]

After append 15:

[5,10,15]

This shows how a list can **grow dynamically while the program runs**.

Why Append is Useful

Append is very useful when programs **collect data over time**, such as:

• storing user inputs
• collecting numbers for calculations
• building a list of items
• recording results during loops

For example, a program could ask the user for numbers and keep **appending them to a list** until the user stops entering data.

6.3.2 Insert

The **insert operation** is used to add a value at a **specific position** in a list. Unlike **append**, which always adds an element at the end, **insert** places the value at the position you specify.

Syntax:

insert value at position in list

Example:

```
insert 99 at 2 in nums
```

If the list is:

nums = [1,2,3,4]

After the insert operation, the list becomes:

[1,99,2,3,4]

Explanation:

The value **99** is inserted at **position 2**. The existing elements starting from position 2 are **shifted one position to the right** to make space for the new value.

So the list changes like this:

Before:
[1,2,3,4]

After:
[1,99,2,3,4]

Important point:
Since Mewar uses **1-based indexing**, position **1 is the first element**, position **2 is the second element**, and so on.

6.3.3 Remove

The **remove operation** is used to delete a specific value from a list. When a value is removed, the remaining elements automatically shift to fill the empty position.

Syntax:

```
remove value from list
```

Example:

```
set nums to [1,2,3,4]

say "Original list:"
say nums

remove 2 from nums

say "After removing 3:"

say nums
```

After the remove operation, the list becomes:

[1,3,4]

Explanation:

The value **2** is removed from the list. After it is deleted, the remaining elements **shift left** to close the gap.

Before:

[1,2,3,4]

After:

[1,3,4]

Important point:
The **remove operation deletes the specified value**, not the position. If the value appears multiple times, the first occurrence is usually removed.

6.3.4 Pop

The **pop operation** is used to remove the **last element** from a list. It is commonly used when the program needs to remove the most recently added item.

Syntax:

pop from list

Example:

set nums to [1, 2, 3, 4]
pop from nums
say nums

After the pop operation, the list becomes:

[1,2,3]

Explanation:

The **last element (4)** is removed from the list. All other elements remain unchanged.

Before:

[1,2,3,4]

After:

[1,2,3]

Important point:
The **pop operation always removes the last element of the list**, making it useful when working with data where the most recent value needs to be removed.

6.3.5 Sort

The **sort operation** is used to arrange the elements of a list in **ascending order** (from smallest to largest). Sorting helps organize data so that it becomes easier to read, analyze, or process.

Syntax:

```
sort list
```

Example:

```
set nums to [4, 1, 3, 2]

sort nums

say "After sort:"

say nums
```

After applying the **sort** operation, the list becomes:

[1,2,3,4]

Explanation:

The sort operation rearranges the values so they appear in **increasing numerical order**.

Before sorting:

[4,1,3,2]

After sorting:

[1,2,3,4]

Important point:
The **sort operation only changes the order of elements**, it does not add or remove any values from the list.

6.3.6 Reverse

The **reverse operation** is used to reverse the order of elements in a list. This means the first element becomes the last, the last element becomes the first, and all other elements change their positions accordingly.

Syntax:

```
reverse list
```

Example:

```
set nums to [1, 2, 3, 4]

reverse nums

say "After reverse:"

say nums
```

After applying the **reverse** operation, the list becomes:

[4,3,2,1]

Explanation:

The order of the elements is completely reversed.

Before reverse:

[1,2,3,4]

After reverse:

[4,3,2,1]

Important point:
The **reverse operation only changes the order of the elements**, it does not modify or remove any values in the list.

6.3.7 Slice

The **slice operation** is used to extract a **portion of a list**. It allows the programmer to select a range of elements from the list without modifying the original list.

Syntax:

```
slice list from start to end
```

Example:

```
set nums to [10, 20, 30, 40]
slice nums from 1 to 3
say "After slice (1 to 2):"
say nums
```

Result:

[10,20,30]

Explanation:

The slice operation selects elements starting from **position 1 up to position 3**. Since Mewar uses **1-based indexing**, position 1 refers to the first element of the list.

So the elements taken are:

nums[1] → 10
nums[2] → 20
nums[3] → 30

Important point:
The **slice operation creates a subset of the list**, meaning it extracts only the selected range of elements while leaving the original list unchanged.

6.4 List Functions

Mewar provides useful built-in functions that perform quick operations on lists.

6.4.1 size(nums)

Returns the number of elements in the list.

Example:

```
set items to [1,2,3,4]
```

```
say "Total items: " + size(items)

say "Same result: " + length(items)
```

Works similar to size and returns the length of the list.

6.4.2 first(nums)

Returns the first element of the list.

Example:

```
set numbers to [5,10,15]

say "First: " + first(numbers)

say "Last: " + last(numbers)
```

Checks whether a value exists in the list.

If nums = [5,10,15]

contains(nums,10) → true

6.4.3 Tell System

One of the most powerful and unique features of the Mewar language is the **Tell System**. This system allows programmers to perform quick calculations and analysis on lists without writing loops.

In many programming languages, you would need to write loops to calculate things such as the sum, maximum value, or average of a list. Mewar simplifies this process by providing built-in **tell commands** that automatically perform these operations.

This makes programs shorter, easier to read, and more efficient.

Example list:

```
set nums to [5,10,15,20]

tell max of nums
```

Finds the **largest value** in the list.

```
Result:

20
```

```
tell min of nums
```

Finds the **smallest value** in the list.

Result:

5

```
tell sum of nums
```

Calculates the **total sum** of all elements.

5 + 10 + 15 + 20 = 50

Result:

50

```
tell average of nums
```

Calculates the **average value** of the list.

Sum = 50
Count = 4

Average = 50 / 4 = 12.5

```
tell count of nums
```

Returns the number of elements in the list.

Result:

4

Why the Tell System is Powerful

The Tell System removes the need to write loops for common list operations.

Without Tell System (traditional approach conceptually):

loop through list
add numbers manually
find max manually
calculate average manually

With Tell System in Mewar:

tell sum of nums
tell max of nums
tell average of nums

This makes Mewar programs **shorter, clearer, and easier to write**, especially when working with collections of data.

Because of this feature, lists in Mewar become extremely powerful tools for data analysis and problem solving.

6.5 STRING (TEXT DATA)

Definition

A **string** is a sequence of characters enclosed in quotation marks. Strings are used to store and manipulate text such as names, messages, sentences, or any textual information in a program.

Example:

```
set name to "Mewar"
```

In this example, the variable **name** stores the string **"Mewar"**.

Strings are widely used in programs to display messages, store user input, or work with textual data.

- **Important Notes**

Strings in Mewar have several important characteristics:

• Strings are **1-dimensional**, meaning they are treated as a single sequence of characters.
• Strings **do not support nested structures** like lists. A string cannot contain another string as a structured element the way lists can contain lists.

6.6 String Functions

Mewar provides several built-in functions to manipulate and analyze strings.

- **Case Conversion**

These functions change the letter case of a string.

6.6.1 upper(name)

Converts all characters to uppercase.

Example:

set name to "Mewar"
say upper("mewar")
Result:
MEWAR

6.6.2 lower(name)

Converts all characters to lowercase.

Example:

set city to "Mewar"
say lower(city)
Result:
mewar

6.6.3 Trim Function

The **trim()** function removes extra spaces from the beginning and end of a string.

Example:

set name to "Mewar"

```
say trim(" mewar ")
```

Result:

mewar

6.6.4 Length Function

The **length()** function returns the number of characters in a string.

Example:

```
say length("Mewar")
```

Result:

5

6.6.5 First and Last Character

These functions return the first or last character of a string.

first(name)

Returns the first character.

Example:

```
say first("Mewar")
```

Result:
M

last(name)

Returns the last character.

Example:

say last("Mewar")
Result:
r

6.6.6 Character Access

Since a string is a sequence of characters, each character has a specific position. These positions are called indexes. Using the index, we can access any individual character from the string.

In the Mewar programming language, strings use 1-based indexing. This means:

- The first character is at position 1
- The second character is at position 2
- The third character is at position 3, and so on

This is different from some other languages that start indexing from 0.

To access a character in Mewar, we use the syntax:

char position of stringVariable

Example Program

```
say "Character Access Test"
set word to "MEWAR"
say "Full word:"
say word
say "First character:"
say char 1 of word
say "Third character:"
say char 3 of word
say "Last character:"
say char 5 of word
```

Output
Character Access Test
Full word:
MEWAR
First character:
M
Third character:
W
Last character:
R

Explanation

First, the program prints a title "Character Access Test".

Then a variable named word is created with the value "MEWAR".

word = "MEWAR"

The characters in this string are arranged as:

Position	Character
1	M
2	E
3	W
4	A
5	R

Now the program accesses characters using their positions:

• char 1 of word → M (first character)
• char 3 of word → W (third character)
• char 5 of word → R (last character)

This feature allows programs to analyze or manipulate individual characters inside a string, which is useful for tasks like:

• validating input
• processing text
• extracting specific characters
• building string algorithms

6.6.7 f-String (Formatted String)

Mewar supports **formatted strings**, also called **f-strings**. An f-string allows variables, expressions, and even function calls to be placed directly inside a string. The values inside the braces {} are automatically evaluated and inserted into the text.

This makes output statements **shorter, clearer, and more powerful**, because we can combine text and calculations in a single statement.

To create a formatted string in Mewar, we place the letter **f** before the quotation marks.

Syntax:

f"text {expression}"

Everything inside the **curly braces {}** is evaluated by the program, and the result is inserted into the string.

Example Program:

```
set name to "Mewar"
```

```
set x to 5

say f"Welcome {name}"

say f"Square = {x * x}"



set city to "Udaipur"

say f"Capital: {upper(city)}"

say f"capital: {lower(city)}"
```

Output:

```
Welcome Mewar

Square = 25

Capital: UDAIPUR

capital: udaipur
```

Explanation:

First, the variable **name** is assigned the value **"Mewar"**.

```
set name to "Mewar"
```

The statement

```
say f"Welcome {name}"
```

places the value of **name** inside the string. The program replaces {name} with **Mewar**.

Result:

Welcome Mewar

Next, the program calculates a mathematical expression inside the formatted string.

say f"Square = {x * x}"

Here, **x = 5**, so the expression **x * x** becomes **25**.

Result:

Square = 25

Then the program demonstrates how **functions can also be used inside f-strings**.

say f"Capital: {upper(city)}"

The function **upper(city)** converts "Udaipur" to uppercase.

Result:

Capital: UDAIPUR

Similarly:

say f"capital: {lower(city)}"

The function **lower(city)** converts "Udaipur" to lowercase.

Result:

capital: udaipur

Important Features of f-Strings:

• They allow **variables to be inserted directly into text**
• They support **mathematical expressions**
• They support **function calls inside { }**
• They make code **more readable and easier to write**

Because of these advantages, **formatted strings are one of the most powerful ways to produce dynamic output in Mewar programs**.

6.6.8 ASCII Functions

ASCII stands for **American Standard Code for Information Interchange**. It is a system that represents characters using numeric values. Every character, letter, number, or symbol has a unique ASCII value.

For example:

Character → ASCII Value

A → 65
B → 66

a → 97
0 → 48

These numeric values allow computers to store and process characters internally.

Getting ASCII Value

In Mewar, the ASCII value of a character can be obtained using the **ascii() function**.

Syntax:

```
ascii(character)
```

Example:

```
say ascii("A")
say ascii("Z")
```

Output:

65
90

Explanation:

ascii("A") returns the ASCII value corresponding to the character **A**.

- **Converting ASCII to Character**

Mewar also allows converting numeric ASCII values back into characters using the **char() function**.

Syntax:

```
char(number)
```

Example:

```
say char(65)
say char(66)
```

Output:

A
B

Explanation:

char(65) converts the ASCII value **65** into the character **A**.

Complete Example

```
set letter to "M"
set code to ascii(letter)

say code
say char(code)
```

Output: 77 M

Explanation:

The variable **letter** stores the character **"M"**. The function **ascii(letter)** converts it into its ASCII value **77**, which is stored in the variable **code**. The program then prints the numeric ASCII value and converts it back to the character using **char(code)**.

This demonstrates how characters and numbers can be converted back and forth using ASCII functions.

- **ASCII Functions**

ASCII (American Standard Code for Information Interchange) is a numeric representation of characters.

Every character has a unique numeric value.

For example:

Character	ASCII Value
A	65

B	66
a	97
0	48

In Mewar, ASCII values can be obtained using the **ascii() function**.

- **Getting ASCII Value**

Syntax:

```
ascii(character)
```

Example:

```
say ascii("A")
say ascii("Z")
```

Output:

```
65
90
```

Explanation:

ascii("A") returns the ASCII value of character **A**.

Converting ASCII to Character

Mewar also allows converting ASCII values back to characters using the **char() function**.

Syntax:

```
char(number)
```

Example:

```
say char(65) say
char(66)
```

Output:

A

B

Explanation: char(65) converts ASCII value **65** into character **A**.

Complete Example

```
set letter to "M" set
code to ascii(letter)
say code say
char(code)
```

Output:
77 M

6.7 MATRIX (2D NUMERIC DATA)

Definition:

A matrix is a two-dimensional structure used to store numbers in the form of rows and columns. Unlike a list which stores values in a single line, a matrix organizes numbers in a table-like structure.

Matrices are widely used in mathematics, engineering, graphics, machine learning, and scientific computing.

In Mewar, matrices are defined using nested lists, where each inner list represents a row.

Example:

matrix A = [[1,2],[3,4]]

This matrix can be visualized as:

Row	Column 1	Column 2
1	1	2
2	3	4

So the structure is:

```
A =

[

 [1,2],

 [3,4]

]
```

Here:

• Row 1 → [1,2]
• Row 2 → [3,4]

This means matrix A has 2 rows and 2 columns.

6.8 Accessing Elements

Each value inside a matrix can be accessed using two indexes.

The first index represents the row, and the second index represents the column.

Syntax:

```
matrix[row][column]
```

Example:

```
say A[1][2]
```

Explanation:

A =

[[1,2],

  [3,4]]

Row 1 → [1,2]
Column 2 → 2

```
Output:

2
```

Important note:

Mewar matrices use 1-based indexing.

Index	Value
A[1][1]	1
A[1][2]	2
A[2][1]	3
A[2][2]	4

6.9 Matrix Functions

Mewar provides several built-in functions to work with matrices easily.

6.9.1 Size of Matrix

The size of a matrix refers to the number of rows and columns it contains. Knowing the size of a matrix is important because many matrix operations (such as addition, multiplication, and transpose) depend on the matrix dimensions.

In the Mewar programming language, the size of a matrix can be determined using two built-in functions:

```
rows(A)

cols(A)
```

• rows(A) returns the number of rows in the matrix.
• cols(A) returns the number of columns in the matrix.

Example:

```
matrix A = [[1,2],[3,4]]



say rows(A)

say cols(A)
```

Output:

2

2

Explanation:

The matrix A is defined as:

A = [

[1,2],

[3,4]

]

This matrix contains two rows and two columns, so its size is 2×2.

Row	Column 1	Column 2
1	1	2
2	3	4

Here:

- Row 1 contains the elements 1 and 2
- Row 2 contains the elements 3 and 4

Rows represent horizontal lines in the matrix, while columns represent vertical lines.

Matrix size is usually written in the form:

rows × columns

For example:

- 2 × 2 matrix → 2 rows and 2 columns
- 3 × 3 matrix → 3 rows and 3 columns
- 2 × 3 matrix → 2 rows and 3 columns

Understanding the size of a matrix is important because certain operations require specific matrix dimensions. For example, matrix addition requires matrices of the same size, and matrix multiplication requires compatible row and column sizes.

6.9.2 Transpose of a Matrix

The transpose of a matrix is obtained by interchanging its rows and columns. In simple terms, the elements of each row become elements of a column, and the elements of each column become elements of a row.

In the Mewar programming language, the transpose of a matrix can be calculated using the built-in function:

```
transpose(A)
```

Example:

```
matrix A = [[1,2],[3,4]]

show transpose(A)
```

This matrix has 2 rows and 2 columns.

Row	Column 1	Column 2
1	1	2
2	3	4

Transpose result:

[[1,3],[2,4]]

Explanation:

In the transpose operation:

- The element at A[1][1] remains in the same position
- A[1][2] moves to position A[2][1]
- A[2][1] moves to position A[1][2]
- A[2][2] remains in the same position

So the transformation looks like this:

Original:

1 2

3 4

Transpose:

1 3

2 4

In other words:

- Row 1 becomes Column 1
- Row 2 becomes Column 2

The matrix is effectively flipped across its main diagonal (top-left to bottom-right).

Where Transpose is Used

The transpose operation is very important in many computational and mathematical applications, including:

• Linear algebra – solving matrix equations and transformations
• Machine learning – vector and matrix manipulations
• Image processing – rotating and transforming images
• Matrix transformations – mathematical operations on multidimensional data

Because of these uses, transpose is one of the fundamental operations in matrix mathematics.

6.9.3 Determinant

The determinant is a special numerical value calculated from a square matrix (a matrix that has the same number of rows and columns). The determinant provides important information about the matrix and is widely used in mathematics and linear algebra.

In the Mewar programming language, the determinant of a matrix can be calculated using the function:

```
det(A)
```

The determinant helps determine properties of a matrix. For example, if the determinant of a matrix is 0, the matrix cannot be inverted. If the determinant is non-zero, the matrix has an inverse.

Example:

```
matrix A = [[1,2],[3,4]]
say "Matrix A:"
show A
say "det(A) = " + det(A)
```

This matrix can be written in table form as:

1	2
3	4

For a 2 × 2 matrix, the determinant is calculated using the formula:

$$\det(A) = (a \times d) - (b \times c)$$

Where the matrix is:

a	b
c	d

So in this example:

a = 1
b = 2

c = 3
d = 4

Now substitute the values into the formula:

$\det(A) = (1 \times 4) - (2 \times 3)$

First calculate the multiplications:

$1 \times 4 = 4$
$2 \times 3 = 6$

Then subtract:

$4 - 6 = -2$

Output:

-2

So the determinant of matrix A is −2.

Why Determinants Are Important

Determinants play an important role in many mathematical and computational problems. They are used in:

- Solving systems of linear equations
- Calculating the inverse of a matrix
- Geometry transformations and coordinate changes
- Computer graphics and scientific calculations

Because of these uses, the determinant is considered one of the most fundamental operations in matrix mathematics.

6.9.4 Matrix Inverse

The inverse of a matrix is another matrix that reverses the effect of the original matrix. In simple terms, when a matrix is multiplied by its inverse, the result becomes the identity matrix.

The identity matrix is a special matrix where the diagonal elements are 1 and the other elements are 0.

Example identity matrix:

1 0

0 1

If a matrix A has an inverse, then:

A × inverse(A) = Identity Matrix

This property makes matrix inverse very useful in solving mathematical problems.

In the Mewar programming language, the inverse of a matrix can be calculated using the function:

```
inverse(A)
```

Example:

Suppose we have the matrix:

A = [[1,2],
[3,4]]

To calculate the inverse, we write:

inverse(A)

The Mewar system automatically calculates the inverse matrix.

For this matrix, the inverse is approximately:

[[-2, 1],
[1.5, -0.5]]

Explanation

The inverse matrix is calculated using a mathematical formula that involves the determinant of the matrix. Only matrices with a non-zero determinant can have an inverse.

For a 2 × 2 matrix:

a b

c d

The inverse formula is:

1 / det(A) ×

d -b

-c a

In this example:

A = [[1,2],
[3,4]]

The determinant is:

det(A) = (1×4) − (2×3)
det(A) = 4 − 6
det(A) = −2

Using the inverse formula, Mewar computes the inverse automatically.

Why Matrix Inverse is Important

Matrix inverse is widely used in many advanced fields of computing and mathematics.

It is useful for:

• Solving systems of linear equations
• Matrix division operations
• Computer graphics transformations
• Scientific and engineering calculations
• Machine learning and data analysis

Because of these applications, the matrix inverse is considered a very powerful mathematical operation in programming and computational mathematics.

6.10 Matrix Operations

Matrices can perform several mathematical operations similar to numbers. These operations allow matrices to be combined, manipulated, and used in complex mathematical computations. In the Mewar programming language, matrices support operations such as addition, subtraction, multiplication, and division.

These operations are widely used in fields like linear algebra, physics simulations, machine learning, graphics, and data processing.

6.10.1 Matrix Addition

Matrix addition combines two matrices by adding their corresponding elements. For matrix addition to work, both matrices must have the same number of rows and columns.

Example:

```
matrix A = [[1,2],[3,4]]

matrix B = [[5,6],[7,8]]

say "A + B:"
```

```
set Add to A + B

show Add
```

Result:

[[6,8],
[10,12]]

Explanation:

Each element of matrix A is added to the corresponding element of matrix B.

1 + 5 = 6
2 + 6 = 8
3 + 7 = 10
4 + 8 = 12

6.10.2 Matrix Subtraction

Matrix subtraction works similarly to matrix addition. The corresponding elements of one matrix are subtracted from another matrix.

Example:

```
matrix A = [[5,6],[7,8]]

matrix B = [[1,2],[3,4]]

say "A - B:"
```

```
set sub to A - B

show sub
```

Result:

[[4,4],
[4,4]]

Explanation:

Each element of matrix B is subtracted from the corresponding element of matrix A.

5 − 1 = 4
6 − 2 = 4
7 − 3 = 4
8 − 4 = 4

6.10.3 Matrix Multiplication

Matrix multiplication is more complex than addition or subtraction. In multiplication, rows of the first matrix are multiplied with columns of the second matrix.

Example:

```
matrix A = [[1,2],[3,4]]

matrix B = [[5,6],[7,8]]

say "A * B:"
```

```
set Mul to A * B

show Mul
```

Result:

[[19,22],
[43,50]]

Explanation:

The first element is calculated as:

$(1\times5) + (2\times7) = 5 + 14 = 19$

The second element:

$(1\times6) + (2\times8) = 6 + 16 = 22$

The third element:

$(3\times5) + (4\times7) = 15 + 28 = 43$

The fourth element:

$(3\times6) + (4\times8) = 18 + 32 = 50$

Matrix multiplication is widely used in graphics transformations, neural networks, and scientific simulations.

6.10.4 Matrix Division

Matrix division is not performed directly like normal numbers. Instead, it is done by multiplying a matrix by the inverse of another matrix.

Operation:

A / B

This means:

A / B = A × inverse(B)

The program first calculates the inverse of matrix B, and then multiplies it with matrix A.

Matrix division is commonly used in:

• solving mathematical equations
• advanced linear algebra computations
• scientific and engineering calculations

6.10.5 Why Matrices Are Important

Matrices are extremely powerful data structures used in many real-world applications.

They are widely used in:

• machine learning
• artificial intelligence

• computer graphics
• physics simulations
• image processing
• robotics
• data science

Because of matrices, the Mewar programming language becomes capable of handling advanced mathematical and scientific problems efficiently.

6.11 DIFFERENCE BETWEEN LIST, STRING AND MATRIX:

In the Mewar programming language, **List, String, and Matrix** are three important data structures used to store and manage data. Although they may appear similar because they store multiple values, they are designed for different purposes and support different types of operations.

A **List** is used to store a collection of values, which may include numbers, text, or even other lists. Lists are very flexible and can hold mixed types of data.

A **String** is used to store **text data**, which is a sequence of characters such as letters, numbers, and symbols enclosed in quotation marks.

A **Matrix** is a **two-dimensional numeric structure** used to store numbers arranged in rows and columns. Matrices are mainly used for mathematical and scientific computations.

The following table shows the key differences between these three data structures:

Feature	List	String	Matrix
Type	Mixed values (numbers, text, etc.)	Text only	Numbers only
Dimension	1D or nested	1D	2D
Indexing	Yes	Yes	Yes
Nested Support	Yes (lists inside lists)	No	Fixed 2D structure
Operations	General data operations	Text-based operations	Mathematical operations

Explanation:

• **List** can store different types of values such as numbers, strings, or even other lists. Because of this flexibility, lists are often used for general-purpose data storage.

• **String** stores only textual data. It supports operations such as case conversion, trimming, substring checking, and character access.

• **Matrix** stores numeric data arranged in rows and columns. It supports mathematical operations such as matrix addition, multiplication, transpose, determinant, and inverse.

Understanding the differences between these structures helps programmers choose the **correct data structure depending on the problem they are solving**.

6.12 PRACTICE QUESTIONS :

Q1. Create a list and print its first and last elements.

Q2. Check whether "war" exists in " MewarLanguage".

Q3. Create a matrix and print its transpose.

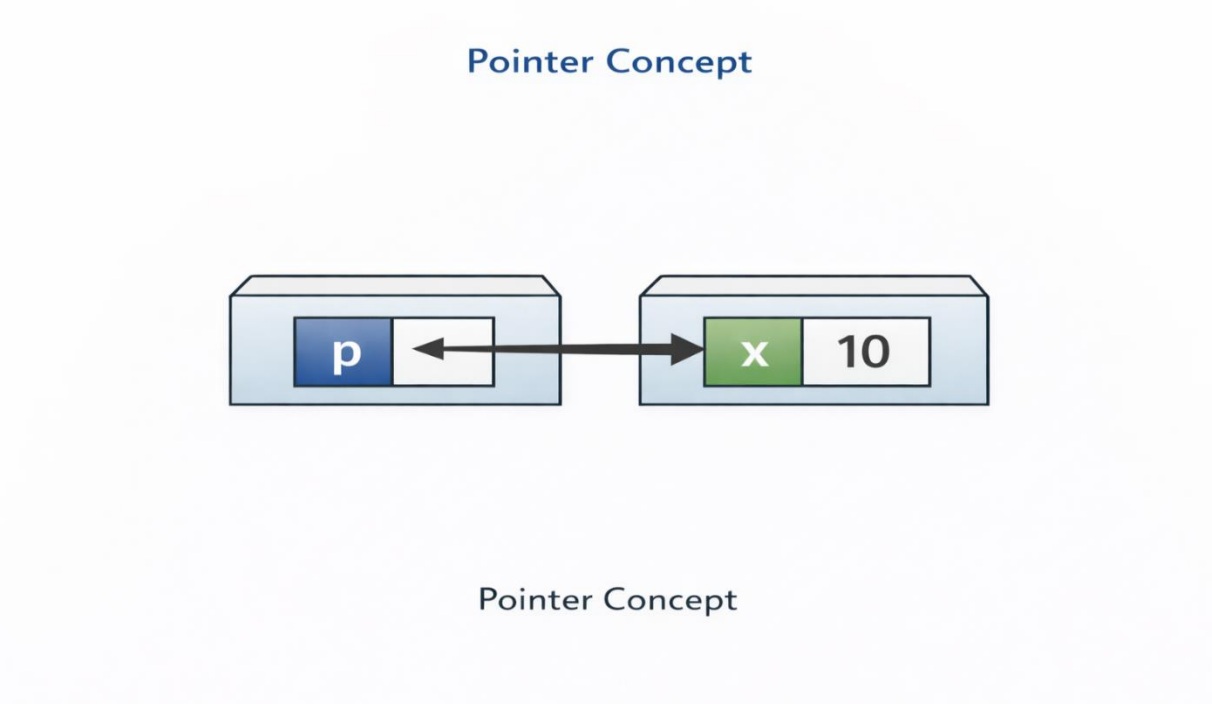
Q4. Find the sum of a list using the tell command.

Chapter 7

Pointer (Mewar Language)

7.1 Introduction to Pointer

A **pointer** is a special type of variable that stores the **address (or reference)** of another variable. Instead of storing the actual value, a pointer keeps the location in memory where that value is stored.



In simple terms, a pointer points to the memory location of another variable. This allows programs to access or modify the value indirectly through the pointer.

Pointers are very useful in advanced programming because they allow efficient memory handling, dynamic data structures, and direct access to stored values.

Simple Idea:

```
x = 10
p = address x
```

Explanation:

• The variable **x** stores the value **10**.
• The variable **p** is a pointer that stores the **memory address of x**.

So instead of holding the number **10**, the pointer **p** holds the location where **x** is stored in memory.

Conceptually it looks like this:

Variable | Value / Address
x → 10
p → address of x

If we access the value stored at that address, we can retrieve the original value of **x**.

Pointers are commonly used for:

• memory management
• efficient data manipulation
• implementing complex data structures
• referencing variables without copying their values

7.2 Pointer in Mewar

In the **Mewar programming language**, pointers are designed to be **simple and readable**. Unlike some traditional programming languages that use complex symbols (like * and &), Mewar uses clear keywords such as **address** and **value** to work with pointers.

This makes pointer concepts easier to understand, especially for beginners.

- **Creating a Pointer**

To create a pointer in Mewar, we store the **address of a variable** in another variable.

Example:

```
set x to 10
set p to address x
say value p
```

Explanation:

• **x** is a normal variable storing the value **10**.

• **address x** returns the **memory location of x**.

• **p** stores that memory location, so **p becomes a pointer to x**.

Conceptually:

Variable	**Stores**
x	10
p	address of x

So **p points to x**.

- ## Accessing Value Using Pointer

Once a pointer has been created, we can use it to **access the value stored at the memory address it points to**. In the Mewar programming language, this is done using the **value** keyword.

The **value keyword** tells the program to read the value stored at the address contained in the pointer.

Example:

```
set x to 10
set p to address x
say value p
```

Output:

10

Explanation:

First, suppose the variables were defined like this:

set x to 10

set p to address x

Here:

- The variable **x** stores the value **10**.
- The pointer **p** stores the **memory address of x**.

Conceptually it looks like this:

Variable | Stored Value

x → 10

p → address of x

When the statement

say value p

is executed, the program performs the following steps:

1. It checks the pointer **p**.
2. It reads the **address stored in p**.
3. It goes to that memory location.
4. It retrieves the value stored there.

Since the address belongs to **x**, the value retrieved is **10**.

Therefore the output printed is:

10

Important Idea:

The **pointer itself does not store the value**, it stores the **location of the value**. Using **value p**, the program accesses the **actual data stored at that location**.

- ## Modifying Value Using Pointer

A pointer can also be used to **modify or update the value of the original variable** it points to. Since the pointer stores the

address of the variable, changing the value at that address automatically changes the original variable.

In Mewar, this is done using the **value keyword** with the **set statement**.

Example:

```
set value p to 50
say x
```

Output:

50

Explanation:

Suppose the variables were defined earlier like this:

set x to 10
set p to address x

Here:

• The variable **x** stores the value **10**.
• The pointer **p** stores the **memory address of x**.

Conceptually:

Variable | Value

x → 10

p → address of x

Now the statement

set value p to 50

does the following:

1. The program looks at pointer **p**.
2. It reads the **address stored in p**.
3. It goes to that memory location.
4. It replaces the value stored there with **50**.

Since that memory location belongs to **x**, the value of **x** changes.

So when the program runs:

say x

the output becomes:

50

Conceptually:

Before modification:

Variable	Value
x	10
p	address of x

After modification:

Variable	Value
x	50
p	address of x

Important idea:

The pointer **does not change its address**, but it changes the **value stored at that address**. As a result, the original variable is updated directly.

7.3 Pointer Operations

In the Mewar programming language, several operations can be performed using pointers. These operations allow programmers to **access memory addresses, retrieve values from pointers, modify variables indirectly, and redirect pointers to different variables**.

- **Address Operator**

The **address operator** is used to obtain the **memory address of a variable**. This address can then be stored in a pointer.

Syntax:

```
set p to address x
```

Explanation:

• **x** is a normal variable containing a value.
• **address x** returns the memory location where **x** is stored.
• The pointer **p** stores this memory address.

Example:

```
set x to 10
set p to address x
```

Conceptually:

Variable | Value
x → 10
p → address of x

Here, **p points to x**.

2. Value Operator (Dereference)

The **value operator** is used to retrieve the **actual value stored at the memory address contained in a pointer**. This process is called **dereferencing**.

Syntax:

```
value p
```

Example:

```
say value p
```

Output:

10

Explanation:

- The pointer **p** stores the address of **x**.
- **value p** accesses the value stored at that address.
- Since **x = 10**, the program prints **10**.

3. Pointer Assignment

Pointers can also be used to **modify the value stored in the original variable**. This is done using the **set value** statement.

Syntax:

```
set value p to 100
```

Example:

```
set value p to 100
say x
```

Output:

100

Explanation:

• The pointer **p** contains the address of **x**.
• **set value p to 100** updates the value stored at that address.
• As a result, the value of **x** becomes **100**.

- ## Pointer to Another Variable

A pointer can also be redirected to **point to a different variable**. This means the pointer will now store the address of a new variable.

Example:

```
set y to 20
set p to address y
```

Explanation:

• A new variable **y** is created with the value **20**.
• The pointer **p** now stores the **address of y** instead of **x**.

Conceptually:

Before:

Variable	Value
x →	10
p →	address of x

After:

Variable	Value
y →	20
p →	address of y

Now the pointer **p points to y instead of x**, and any operation using **p** will affect **y**.

7.4 Mewar vs C Pointers

Pointers exist in many programming languages, but their syntax and complexity can vary greatly. The Mewar programming language was designed to make pointer concepts **simpler and more readable**, especially for beginners. In contrast, the C programming language uses **symbol-based syntax**, which can be more difficult for new programmers to understand.

The following table compares pointers in **Mewar** and **C**.

Feature	Mewar Pointer	C Pointer
Syntax	address x	&x
Access Value	value p	*p
Safety	Safe and controlled	Risky if misused
Memory Control	No manual memory management	Full manual memory control
Complexity	Easy to understand	More complex

Explanation:

• **Syntax**
In Mewar, the address of a variable is obtained using the keyword **address**.
In C, the symbol **&** is used to get the memory address.

• **Accessing Value**
Mewar uses the clear keyword **value** to access the value stored at the pointer.
C uses the symbol * (called the dereference operator).

• **Safety**
Mewar pointers are designed to be safer because memory management is handled automatically.
In C, incorrect pointer usage can lead to errors such as segmentation faults or memory corruption.

• **Memory Control**
C provides full control over memory allocation and deallocation, which is powerful but risky.
Mewar hides this complexity to keep programming simpler.

• **Complexity**
Mewar pointer syntax is easier to read and understand.
C pointer syntax is more powerful but requires deeper knowledge of memory management.

Example Comparison

Mewar Language

```
set x to 10
set p to address x
set value p to 20
```

Explanation:

• **x** stores the value **10**
• **p** stores the address of **x**
• **set value p to 20** changes the value of **x** through the pointer

After execution, **x becomes 20**.

C Language

```
int x = 10;
int *p = &x;
```

*p = 20;

Explanation:

• **x** stores the value **10**
• **p** stores the address of **x** using **&x**
• ***p = 20** modifies the value stored at that address

Both programs perform the **same concept**, but the Mewar syntax is **cleaner, easier to read, and more beginner-friendly** compared to the symbolic pointer syntax used in C.

7.5 Important Notes

When working with **pointers in the Mewar programming language**, there are some important rules and behaviors that programmers should understand. These rules help prevent errors and clarify how pointers interact with variables.

1. Pointer Must Point to a Valid Variable

A pointer can only store the address of a variable that already exists. If the variable has not been defined, the program will produce an error.

Example:

```
set p to address x # if x is not defined → error
```

Explanation:

The program tries to obtain the address of **x**, but if **x** has not been created, the system does not know where it exists in memory. Therefore, the pointer cannot point to it.

Correct usage:

```
set x to 10
set p to address x
```

Now the pointer **p** correctly stores the address of **x**.

2. Only Pointers Support the value Keyword

The **value keyword** is used only with pointers. It retrieves the value stored at the memory address contained in the pointer.

Example:

```
set x to 10
say value x # ERROR
```

Explanation:

Here, **x** is a normal variable, not a pointer. Since it does not store an address, the **value keyword cannot be used with it**.

Correct usage:

```
set x to 10
set p to address x
say value p # Output: 10
```

3. Pointer Stores Reference, Not Copy

A pointer stores the **reference (address) of a variable**, not a copy of its value. This means that if the original variable changes, the value accessed through the pointer also changes.

Example:

```
set x to 10
set p to address x

set x to 99
say value p
```

Output:

99

Explanation:

• The pointer **p** stores the address of **x**.
• When the value of **x** changes to **99**, the pointer still points to the same memory location.
• Therefore, **value p** returns **99**.

4. Pointer Overwrite

A pointer can be redirected to another variable by assigning it a new address. When this happens, the pointer stops pointing to the old variable and begins pointing to the new one.

Example:

```
set a to 10
set b to 20

set p to address a
set p to address b # now p points to b
```

Explanation:

Initially:

Variable | Value
a → 10
b → 20
p → address of a

After the second assignment:

Variable | Value
a → 10
b → 20
p → address of b

Now the pointer **p points to b instead of a**. Any operation performed using **p** will affect **b**, not **a**.

7.6 Miscellaneous Examples

The following examples demonstrate how pointers can be used in different situations in the Mewar programming language. These examples help illustrate how pointers access, modify, and transfer values between variables.

Example 1: Basic Pointer

This example shows how a pointer can store the address of a variable and access its value.

```
set x to 10

set p to address x

say value p
```

Explanation:

- The variable x stores the value 10.
- The pointer p stores the address of x.
- value p accesses the value stored at that address.

Output:

```
10
```

Example 2: Modify via Pointer

A pointer can modify the value of the original variable.

```
set x to 5

set p to address x

set value p to 100

say x
```

Explanation:

- x initially stores 5.
- p points to x.
- set value p to 100 changes the value stored at the address of x.

```
Output:

100
```

This shows that modifying the value through a pointer directly updates the original variable.

Example 3:

Pointers can be used to copy values between variables indirectly.

```
set a to 10

set b to 20

set p1 to address a

set p2 to address b

set value p2 to value p1

say b
```

Explanation:

- p1 points to a.
- p2 points to b.
- value p1 retrieves the value of a, which is 10.
- set value p2 to value p1 assigns that value to b.

Output:

10

So the value of b becomes 10.

7.7 Review Questions

The following questions will help you test your understanding of **pointers in the Mewar programming language**.

Short Answer Questions

1. What is a pointer?

2. What does the **address** keyword do in Mewar?

3. What is the purpose of the **value** keyword?

4. What happens when you execute set value p to 50?

5. Can a pointer store the value of a variable directly? Explain.

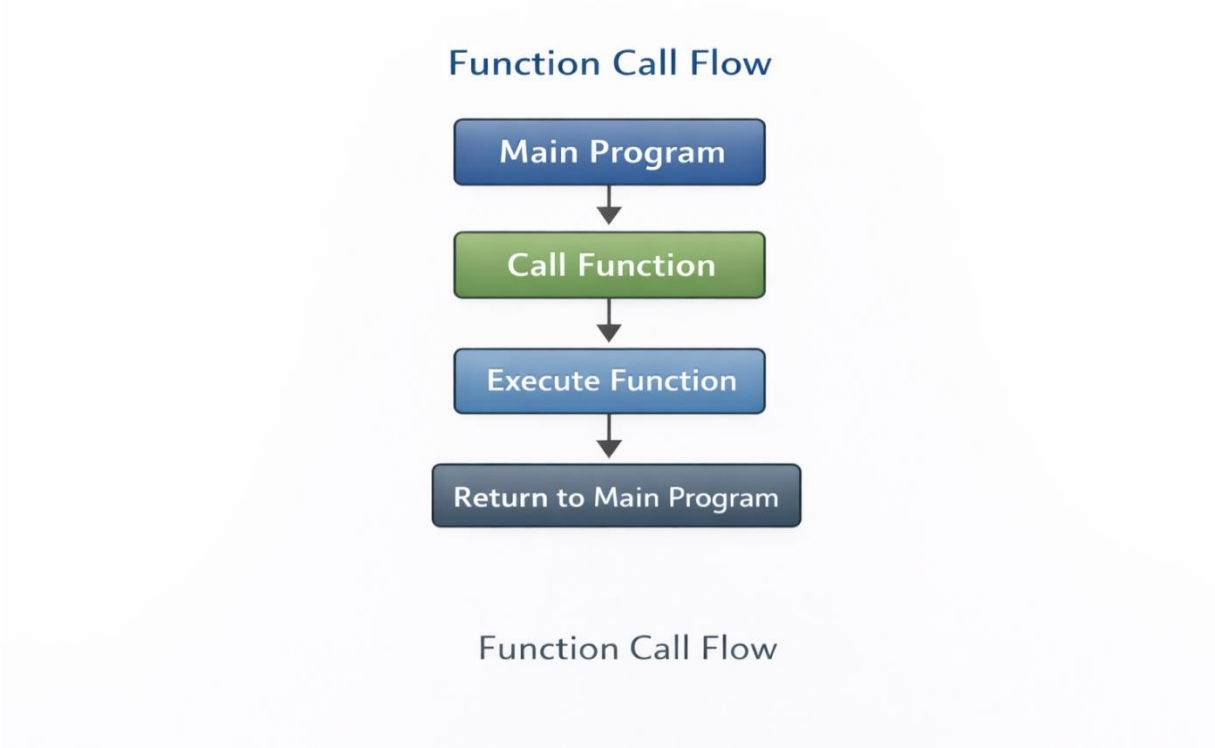
6. What happens if a pointer tries to access a variable that does not exist?

Chapter 8

Functions in Mewar

8.1 Introduction

A **function** is a reusable block of code that performs a **specific task**. Instead of writing the same set of instructions multiple times in a program, a function allows the programmer to write the code once and reuse it whenever needed.



Function Call Flow

Functions help organize programs into smaller and more manageable parts. Each function focuses on a particular task, making the program easier to understand and maintain.

Functions are very important in programming because they improve the overall structure and readability of the code.

Functions help in:

• Code reuse – The same function can be called multiple times without rewriting the code.

• Clean program structure – Large programs can be divided into smaller logical parts using functions.

• Easy debugging – Errors can be found and fixed more easily because each function performs a specific task.

Example idea:

A function that calculates the square of a number can be written once and used multiple times in a program.

By using functions, programmers can write modular, organized, and efficient programs.

8.2 Function Syntax

In the Mewar programming language, a **function** is defined using the keyword **function**. The function contains a block of code that performs a specific task and can optionally return a value.

A function definition includes:

• The **function name**
• Optional **parameters** (inputs)
• The **code block** that performs the task
• An optional **return statement** to send a result back
• The **end** keyword to close the function

Syntax:

```
function function_name with param1, param2 then
  # code
  return value
end
```

Explanation:

• **function** → keyword used to define a function
• **function_name** → the name of the function
• **with param1, param2** → parameters that the function receives as input
• **then** → begins the function body
• **# code** → the statements that perform the task
• **return value** → sends a result back to the caller
• **end** → marks the end of the function

Example:

```
function add with a, b then
  set result to a + b
  return result
end
```

Explanation:

• The function **add** receives two parameters: **a** and **b**.
• It calculates their sum and stores it in **result**.
• The **return statement** sends the result back to the program.

Functions allow programs to perform tasks repeatedly without rewriting the same code.

8.3 Function Without Parameters

A **function without parameters** is a function that does not receive any input values. It simply performs a specific task whenever it is called.

Such functions are useful when the same action needs to be repeated multiple times in a program without requiring any external data.

Example:

```
function greet then

  say "Welcome to Mewar"

end

call greet
```

Explanation:

1. The function **greet** is defined. At this stage, the interpreter stores the function in memory but does not execute it yet.

2. When the statement **call greet** is executed, the interpreter jumps to the function definition.

3. The instruction inside the function is executed:

say "Welcome to Mewar"

4. After executing the code, the function reaches the **end** statement and the control returns back to the main program.

Output: Welcome to Mewar

Important Idea:

A function can be **called multiple times** in a program. Each time the function is called, the same block of code executes again. This helps avoid repeating the same code manually and keeps programs clean and organized.

8.4 Function With Parameters & Argument

Functions can also accept **input values** so they can perform operations on different data each time they are called. These inputs are handled using **parameters** and **arguments**.
A **parameter** is a variable used in the **function definition** to receive input values.
An **argument** is the **actual value passed to the function** when it is called.

Example:

```
function add with a, b then
  return a + b
end
set result to call add with 5, 3
say result
```

Explanation:

The function **add** is defined with two parameters: **a** and **b**.
When the statement
call add with 5, 3
is executed, the values **5** and **3** are passed to the function.
These values are called **arguments**.
The parameters receive the values:
a = 5
b = 3
The function executes the statement:
return a + b
which calculates:
5 + 3 = 8
The value **8** is returned to the main program and stored in the variable **result**.
Finally, the program prints the result.

```
Output:
8
```

Important Idea:

- **Parameters** appear in the function definition.
- **Arguments** appear when the function is called.

• The return statement sends the calculated result back to the program.

8.5 Function With Return Value

A **return value** is the output produced by a function and sent back to the place where the function was called. In the Mewar programming language, this is done using the **return** keyword.

The return statement allows a function to perform a calculation and send the result back to the main program.

Example:

```
function square with x then
  return x * x
end
set result to call square with 4
say result
```

Explanation:

The function **square** is created to calculate the square of a number.

• The function receives one parameter **x**.

• Inside the function, the expression **x * x** calculates the

square of the number.

• The **return** statement sends this result back to the caller.

Flow of Execution:

1. The statement

call square with 4

calls the function and passes the value **4**.

2. The parameter receives the value:

x = 4

3. The function executes the calculation:

return 4 * 4

4. The result **16** is returned to the program and stored in the variable **result**.

5. Finally, the program prints the value of **result**.

Output: 16

8.6 Function Without Return

A function without a return value performs an action but does not send any value back to the caller. These functions are often used for tasks such as printing messages, displaying results, or performing operations that do not require a returned value.

In such functions, the return keyword is not used. The function simply executes its instructions and then ends.

Example:

```
function show_message with name then
  say "Hello " + name
end
call show_message with "Veer"
```

Explanation:

The function show_message is defined with one parameter called name.

When the statement

call show_message with "Veer"

is executed, the value "Veer" is passed to the function.

The parameter receives the value:

name = "Veer"

Then the function executes the instruction:

say "Hello " + name

which prints the greeting message.

Output:
Hello Veer

Key Idea:

A function without a return value performs an action only. It does not produce a result that needs to be stored or used elsewhere in the program. After executing its statements, the function simply finishes and control returns to the main program.

8.7 Function With Condition

A function can also include **conditional statements** to make decisions based on input values. By using conditions such as **if–else**, the function can return different results depending on the situation.

This allows functions to become more **dynamic and intelligent**, because their output changes depending on the input provided.

Example:

```
function check_age with age then
  if age >= 18 then
    return "Adult"
  else
    return "Minor"
  end
end

set status to call check_age with 20
say status
```

Explanation:

The function **check_age** receives one parameter called **age**.

Inside the function, an **if condition** checks whether the value of **age** is greater than or equal to **18**.

If the condition is true, the function returns **"Adult"**.
If the condition is false, the function returns **"Minor"**.

Flow of Execution:

1. The function is called:

call check_age with 20

2. The parameter receives the value:

age = 20

3. The condition is evaluated:

20 >= 18 → true

4. Since the condition is true, the function executes:

return "Adult"

5. The returned value **"Adult"** is stored in the variable **status**.

6. The program prints the value of **status**.

Output: Adult

Key Idea:

Functions combined with **conditional logic** allow programs to make decisions and return different results based on input values. This is widely used in real programs such as **age verification, grading systems, and decision-making applications**.

8.8 Function Calling Function

A **function call** is the process of executing a function using the **call** keyword. In many programs, one function may need

to use the result produced by another function. This is called **function calling another function**.

This technique helps in **breaking complex tasks into smaller reusable functions**, making programs more organized and easier to maintain.

Example:

```
function add with a, b then
  return a + b
end

function double_sum with x, y then
  set temp to call add with x, y
  return temp * 2
end

set result to call double_sum with 3, 4
say result
```

Explanation:

The program defines **two functions**:

1. **add**
   This function receives two numbers and returns their sum.
2. **double_sum**
   This function calls the **add** function, then multiplies the result by **2**.

Flow of Execution:

1. The program calls the function:
   call double_sum with 3, 4
2. The parameters receive the values:

x = 3
y = 4

3. Inside **double_sum**, the function **add** is called:
   call add with x, y
4. The **add** function executes:
   return 3 + 4
   Result:
   7
5. The returned value **7** is stored in the variable **temp**.
6. The function then calculates:
   return temp * 2
   7 * 2 = 14
7. The value **14** is returned to the main program and stored in **result**.

Output: 14

Key Idea:

A function can call another function to reuse its logic. This helps in creating **modular, structured, and reusable programs**, which is an important principle in software development.

8.9 Important Rule (VERY IMPORTANT)

In the Mewar programming language, a **function call that returns a value cannot be used directly inside the say statement**. This is because the interpreter or parser treats the expression as a **variable name instead of a function call**.

For this reason, the returned value from a function must first be **stored in a variable**, and then that variable can be printed.

Incorrect Example (This will NOT work):

```
say call square with 4
```

Explanation:

When the interpreter reads this statement, it tries to treat **call square with 4** as a variable rather than a function call. Because of this, the program will produce an error or unexpected behavior.

Correct Way:

```
set result to call square with 4

say result
```

Explanation:

1. The function **square** is called with the argument **4**.
2. The returned value from the function is stored in the variable **result**.

3. The program then prints the value stored in **result** using the **say** statement.

Example Function:

```
function square with x then
  return x * x
end

set result to call square with 4
say result

Output:
16
```

Key Idea:

Always **store the returned value of a function in a variable before printing it**. This rule ensures that the Mewar interpreter correctly processes function calls and avoids parsing errors.

8.10 Function with Pointer (Advanced)

In the Mewar programming language, a **function can also work with pointers**. When a pointer is passed to a function, the function can **directly modify the original variable** because it has access to the variable's **memory address**. This technique is known as **pass-by-reference**.

In pass-by-reference, the function receives the **reference (address) of the variable**, not a copy of its value. Therefore, any change made through the pointer **affects the original variable**.

Example:

```
function update with ptr then
  set value ptr to 200
end

set x to 10
set p to address x

call update with p

say x
```

Explanation:

First, the variable **x** is created and assigned the value **10**.

set x to 10

Next, a pointer **p** is created that stores the **address of x**.

set p to address x

This means **p points to x**.

Then the function **update** is called:

call update with p

Inside the function, the statement

set value ptr to 200

changes the value stored at the memory address contained in **ptr**.

Since **ptr holds the address of x**, the value of **x** becomes **200**.

Flow of Execution:

p → points to x

call update(p)

→ set value p to 200

→ x becomes 200

Output: 200

Key Idea:

Passing a pointer to a function allows the function to **modify the original variable directly**. This technique is called **pass-by-reference** and is commonly used when a function needs to update variables outside its own scope.

8.11 Function with Matrix

In the Mewar programming language, a **function can also accept a matrix as an input parameter**. This allows the function to perform different **matrix operations** such as addition, multiplication, scaling, or transformation.

Passing a matrix to a function makes it possible to create **reusable mathematical operations** that can work on different matrices.

Example:

```
function double_matrix with m then
  return m * 2
end

matrix A = [[1,2],[3,4]]

set B to call double_matrix with A
show B
```

Explanation:

The function **double_matrix** is defined with one parameter called **m**.

This parameter represents a **matrix** passed to the function.

Inside the function, the statement:

return m * 2

multiplies every element of the matrix **m** by **2**.

Next, a matrix **A** is created:

matrix A = [[1,2],[3,4]]

This matrix looks like:

1	**2**
3	4

Then the function is called:

set B to call double_matrix with A

Here:

• The matrix **A** is passed to the function.
• The parameter **m** receives the matrix **A**.

So inside the function:

m = [[1,2],[3,4]]

Now the operation is performed:

m * 2

Each element of the matrix is multiplied by **2**.

Calculation:

$1 \times 2 = 2$

$2 \times 2 = 4$

$3 \times 2 = 6$

$4 \times 2 = 8$

The resulting matrix becomes:

[[2,4],

[6,8]]

This result is returned from the function and stored in **B**.

Output:

[2,4]

[6,8]

Key Idea:

Functions that accept matrices allow programmers to **create reusable mathematical operations**. These functions can be used for tasks such as:

• scaling matrices

• performing matrix transformations

• implementing mathematical algorithms

• building scientific or engineering applications.

8.12 Common Errors

When working with **functions in the Mewar programming language**, certain mistakes can cause errors during program execution. Understanding these common errors helps programmers write correct and reliable code.

1. Calling a Function Without Assignment

A function that returns a value must first store that value in a variable before printing it.

Incorrect Example:

```
say call add with 2, 3   # ERROR
```

Explanation:

The interpreter may treat **call add with 2, 3** as a variable instead of a function call, which causes an error.

Correct Example:

```
set result to call add with 2, 3

say result
```

Here the returned value is stored in **result**, and then printed.

2. Wrong Number of Arguments

A function must receive the **correct number of arguments** that match its parameters. If arguments are missing or extra arguments are passed, the function will not execute correctly.

Incorrect Example:

```
call add with 5   # missing value
```

Explanation:

If the function **add** expects two parameters (for example a and b), passing only one argument will cause an error.

Correct Example:

```
call add with 5, 3
```

Both parameters now receive values.

3. Calling an Undefined Function

A function must be **defined before it is called**. If the program tries to call a function that does not exist, an error occurs.

Incorrect Example:

```
call test   # not defined
```

Explanation:

Since the function **test** has not been defined, the interpreter does not know what instructions to execute.

Correct Example:

```
function test then
  say "Function running"
end

call test
```

Now the function is defined before it is called.

8.13 Practice Questions

1. Write a function to reverse a string
2. Write a function to sum a list
3. Write a function to multiply two matrices
4. Write a function using pointer to change value
5. Write a function to check prime number

Chapter 9

Error Handling, Escape Support, Const, Record & Enum

9.1 Error Handling (Definition)

Error handling is a mechanism used to safely manage **runtime errors** without causing the program to crash. Runtime errors are problems that occur while the program is executing, such as division by zero, invalid operations, or accessing undefined values.

In the Mewar programming language, error handling is implemented using the structured control blocks:

test → fail → else → close

This structure allows a program to detect errors, respond to them appropriately, and continue running safely.

Syntax

```
test
  # risky code
```

```
fail err then
  # error handling
else
  # runs if no error
close
  # always runs
end
```

Explanation of the blocks:

• **test** – Contains the main code that might cause an error.

• **fail** – Executes if an error occurs during the test block.

• **else** – Executes only when no error occurs.

• **close** – Executes regardless of whether an error occurred or not.

• **end** – Marks the end of the error handling structure.

Example

```
test
  say "Enter 0 to trigger an error."
  set num to ask "Enter a number (0 = crash): "

  if num == 0 then
```

```
    set boom to step(1, 5, 0)
  end

  say "No error happened in test."
fail err then
  say " Error handled: " + err
else
  say " test block completed with no error."
close
  say " Log: test block closed."
end
```

Explanation

The program starts with the test block, which contains code that might produce an error.

The program prints a message asking the user to enter a number.

The statement:

set num to ask "Enter a number (0 = crash): "

takes input from the user.

Case 1: User enters 0

If the user enters 0, the condition becomes true:

if num == 0

Then this statement runs:

set boom to step(1, 5, 0)

Here, the step function uses 0 as the step size, which is invalid because a loop cannot move forward with a step of zero. This produces a runtime error.

When the error occurs:

• The test block stops execution.
• The program jumps to the fail block.
• The error message is stored in the variable err.

Output example:

Error handled: invalid step value

Log: test block closed.

Case 2: User enters a number other than 0

If the user enters a number like 5, the condition:

if num == 0

is false, so the risky code does not run.

The program continues normally and prints:

No error happened in test.

Since no error occurred:

• The fail block is skipped
• The else block executes

Output example:

No error happened in test.

test block completed with no error.

Log: test block closed.

Block Purpose Summary

Block	**Purpose**
test	Main code where an error may occur
fail	Executes if an error occurs
else	Executes if no error occurs
close	Always executes

9.2 Escape Support (Definition)

Escape sequences are special character combinations used inside strings to represent characters that are difficult or impossible to type directly. They usually begin with a **backslash** \ followed by another character.

Escape sequences allow programmers to include **new lines, quotation marks, or special symbols** inside a string without causing syntax errors.

Common Escape Characters

Escape	Meaning
\n	New line
\"	Double quote inside a string
\'	Single quote inside a string
\\	Backslash character

Example

```
say "Hello\nWorld"
say "He said \"Welcome\""
say 'It\'s Mewar'
```

Output

```
Hello
```

```
World
He said "Welcome"
It's Mewar
```

Explanation

1. **New Line (\n)**

```
say "Hello\nWorld"
```

The \n escape sequence moves the text to the **next line**.

```
Result:
Hello
World
```

2. **Double Quote (\")**

```
say "He said \"Welcome\""
```

Normally, double quotes would end the string.
Using \" allows the quote to appear **inside the string**.

```
Result:
He said "Welcome"
```

3. **Single Quote (\')**

```
say 'It\'s Mewar'
```

The escape \' allows the apostrophe to appear inside a single-quoted string.

```
Result:
It's Mewar
```

9.3 Const Variable (Definition)

A **const variable** is a variable whose value **cannot be changed after it is initialized**. Once a value is assigned to a constant variable, it becomes **fixed for the entire program**. Const variables are useful when a value should remain **permanent and protected from accidental modification**, such as mathematical constants or configuration values.

Syntax

```
const variable_name to value
```

Example:

```
const pi to 3.14
say pi
Output:
3.14
```

Explanation:

• The keyword **const** is used to declare a constant variable.

• The variable **pi** is assigned the value **3.14**.

• Since it is declared as a constant, its value **cannot be changed later** in the program.

Invalid Reassignment (Error)

```
set pi to 10
```

Explanation:

This statement tries to **change the value of pi**, but since **pi was declared as a constant**, the program will generate an **error**.

Constants protect important values from being modified accidentally.

9.4 Record (Definition)

A **record** is a user-defined data structure used to store **multiple related values together** in a single variable. Each value inside the record is stored in a **field**, and each field represents a specific property of the record.

Records help organize related information into a single structured object. This makes programs easier to manage and understand when dealing with complex data.

Syntax

```
record Name then
  field1
  field2
end
```

Explanation:

- **record** → keyword used to define a record structure
- **Name** → name of the record type
- **field1, field2** → variables that belong to the record
- **end** → marks the end of the record definition

Example

```
record Warrior then
  name
  power
  alive
end

set w to Warrior
set w.name to "Pratap"
set w.power to 100
set w.alive to true

say w.name
```

Explanation

First, a record named **Warrior** is defined with three fields:

- **name**
- **power**
- **alive**

Next, a variable **w** is created using the **Warrior** record.

set w to Warrior

Then values are assigned to the fields:

set w.name to "Pratap"

set w.power to 100

set w.alive to true

This creates a structured object that stores all related data together.

Conceptually:

Field	**Value**
name	Pratap
power	100
alive	true

Finally, the program prints the value of **w.name**.

Output:

Pratap

Key Points

• A **record groups multiple related values into one structure**.

• Each value is accessed using **dot notation** (e.g., w.name).

• Records are **similar to structures (struct) in the C programming language**.

• They are useful for storing complex data such as **user profiles, game characters, or database-like records**.

9.5 Enum (Definition)

An **enum** (short for **enumeration**) is a data structure used to define a **collection of fixed named constants**. These constants represent a set of predefined values that a variable can take.

Enums are useful when a variable should only contain **one value from a limited set of options**. Instead of using random numbers or strings, enums provide **clear and readable names**.

Syntax

```
enum Name then
  VALUE1
  VALUE2
  VALUE3
end
```

Explanation:

• **enum** → keyword used to define an enumeration

• **Name** → name of the enum type

• **VALUE1, VALUE2, VALUE3** → constant values in the enumeration

• **end** → marks the end of the enum definition

Example

```
enum Color then
  RED
  GREEN
  BLUE
end

say Color.RED
say Color.GREEN
```

Explanation

The enum **Color** defines three constant values:

• RED

• GREEN

• BLUE

These values are accessed using **dot notation**.

Color.RED

Color.GREEN

```
Output:
RED
GREEN
```

Internally, the system automatically assigns **numeric values** to each constant.

Conceptually:

Enum Value	Internal Number
RED	0
GREEN	1
BLUE	2

However, programmers usually use the **names instead of numbers**, which makes code easier to understand.

Key Points

• Enums define **a fixed set of constant values**.

• Values are **automatically numbered internally**.

• They are useful when a variable should only have **specific**

9.6 Difference: Record vs Enum

Feature	Record	Enum
Purpose	Store data	Fixed values
Type	Object	Constant group
Example	Warrior	Color

9.7 Important Notes

Error handling prevents crashes.

Escape sequences improve strings.

Const variables are immutable

Record stores structured data

Enum defines fixed values

FINAL SUMMARY

Concept	**Use**
Error Handling	safe execution
Escape	string formatting
Const	fixed value
Record	structured data
Enum	fixed constants

9.8 Review Questions

These questions will help test your understanding of **Error Handling, Escape Sequences, Const Variables, Records, and Enums** in the Mewar programming language.

Short Answer Questions

1. What is **error handling** in programming?
2. Which keywords are used for error handling in Mewar?
3. What is the purpose of the **fail** block?
4. What does the **close** block do in error handling?
5. What are **escape sequences**?
6. What does the escape sequence **\n** represent?
7. What is a **const variable**?
8. What happens if you try to change the value of a const variable?
9. What is a **record** in Mewar?
10. How are fields of a record accessed?
11. What is an **enum (enumeration)**?
12. Why are enums useful in programming?

Chapter 10

Introduction to Intent-Based Programming

10.1 What is Intent-Based Programming

Intent-Based Programming is a programming approach where programs are written not only to perform instructions but also to **express the intention or goal behind those instructions**.

In traditional programming languages, code usually focuses on **what to do step by step**. In Intent-Based Programming, the program also communicates **why the operation is being performed**.

This means the program can describe its purpose, track goals, and provide explanations about its internal behavior.

In the **Mewar programming language**, this concept is implemented through features that allow programs to:

• **Explain themselves** – the program can describe what it is doing.
• **Track goals** – the program can define objectives and

monitor progress.

• **Analyze data** – the program can interpret results rather than just compute them.

• **Show internal state** – the program can reveal how its data changes during execution.

Core Idea

Traditional Programming Languages:

“Do this.”

Example idea:

set total to a + b

The program simply performs the operation.

Mewar (Intent-Based Programming):

“Do this + Understand why.”

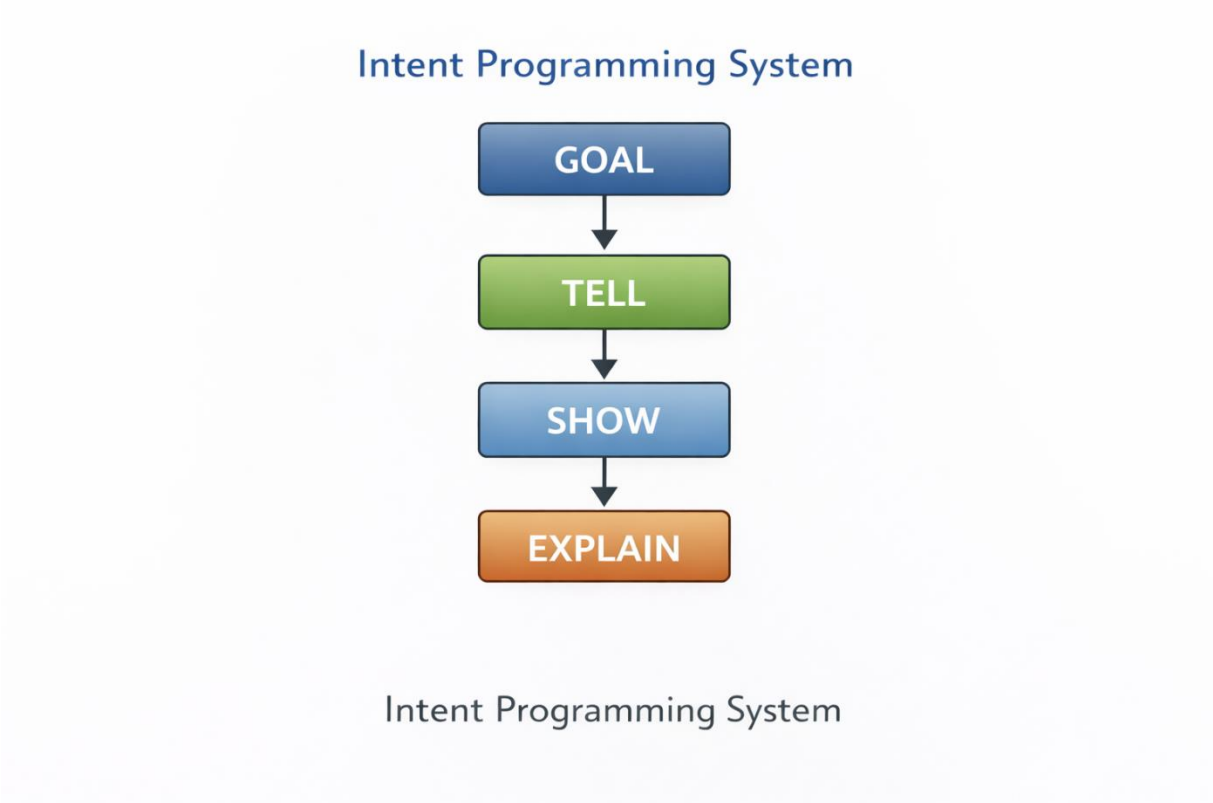
Example idea:

```
goal calculate_total

set total to a + b

explain "Calculating total value of inputs"
```

Here the program not only performs the calculation but also **expresses the intention behind it**.

10.2 The Four Core Concepts

Intent-Based Programming in the Mewar language is built around four core concepts. These concepts allow programs to express their purpose, reasoning, analysis, and internal behavior. Instead of only executing instructions, the program can also describe what it is doing and why.



The four core concepts are:

1. goal → What we want to achieve

2. explain → Why or how something works

3. tell → Analyze data and produce insights

4. show → Display the internal state of the program

10.2.1 goal — What We Want

The **goal** statement is used to define the **main objective or intention of a program**. It clearly describes what the program is trying to achieve. By specifying a goal, the program becomes easier to understand because the reader immediately knows the purpose of the code.

In Intent-Based Programming, defining the goal helps the program communicate its **intended outcome**, not just the instructions it performs.

Example:

```
goal sum == 10
goal "Sum must be correct"

set a to 3
set b to 4
set sum to a + b

explain program
```

Explanation

- **Goal Declaration**

goal "Sum must be correct"

This statement defines the **intention of the program**. It tells the reader that the purpose of the program is to **ensure the calculated sum is correct**.

In Intent-Based Programming, this makes the code **self-descriptive**.

- **Variable Initialization**

set a to 3

set b to 4

Two variables are created:

• **a = 3**
• **b = 4**

These values will be used in the calculation.

- **Performing the Calculation**

set sum to a + b

The program adds the values of **a** and **b**.

Calculation:

3 + 4 = 7

The result **7** is stored in the variable **sum**.

- **Explaining the Program**

explain program

This statement asks the program to **describe what it has done**. The system can explain the goal, the variables used, and the calculation performed.

10.2.2 Explain — Why / How It Works

The explain statement is used to describe why a particular operation is performed or how the program works. It allows the program to provide a clear explanation of its logic, making the code easier to understand.

In Intent-Based Programming, the explain statement acts like built-in documentation. Instead of writing separate comments, the program itself can describe its behavior and reasoning.

Example:

```
set x to 10
set y to 20
set sum to x + y

explain sum
```

Explanation

- **Variable Initialization**

set x to 10
set y to 20
Two variables are created:

• x = 10
• y = 20

- **Performing the Calculation**

set sum to x + y
The program adds the two numbers:

10 + 20 = 30

The result 30 is stored in the variable sum.

- **Explaining the Result**

explain sum

This statement tells the system to explain what the variable sum represents.

The explanation could conceptually mean:

• sum is the result of adding x and y
• x = 10
• y = 20
• sum = 30

So the program is not only computing

10.2.3 tell — Analyze Data

The **tell** command is used to **analyze data and produce useful information or insights**. Instead of manually writing loops or calculations, the tell statement allows the program to quickly compute summaries such as totals, averages, maximum values, and other statistics.

In Intent-Based Programming, **tell represents the analytical ability of the program**. It allows the program to interpret data and report meaningful results.

Example:

```
tell sum of numbers

tell average of numbers
```

Explanation

Suppose we have a list of numbers:

set numbers to [10, 20, 30, 40]

Now when the program executes:

tell sum of numbers

The system calculates the total of all elements:

10 + 20 + 30 + 40 = 100

Result: 100

Next:

tell average of numbers

The program calculates the average value.

Average = (10 + 20 + 30 + 40) / 4
Average = **25**

Result:

25

What tell Can Analyze

The **tell command** can analyze lists or collections and compute values such as:

- **sum** → total of all elements
- **average** → average value
- **max** → largest value
- **min** → smallest value
- **count** → number of elements

Example:

```
tell max of numbers

tell min of numbers

tell count of numbers
```

10.2.4 show — Display Internal State

The show command is used to display the internal state of variables, lists, matrices, or other data structures during program execution. It allows the programmer to see the current value stored inside a variable.

This is useful for understanding how data changes step by step while the program runs.

Example:

```
show numbers
show result
```

Explanation

Suppose we have the following program:

```
set numbers to [10, 20, 30]

set result to 60

show numbers

show result
```

When the program runs, the show command prints the current internal values of the variables.

```
Output:

[10, 20, 30]

60
```

This allows the programmer to observe the actual state of the program's data at that moment.

Why show is Useful

The show statement helps developers to:

- observe how variables change during execution
- debug programs easily
- verify that calculations are correct
- inspect complex data structures like lists or matrices

Because it reveals the internal state of the program, it acts like a built-in debugging tool.

10.3 Real Power of the Intent System

The Intent System is what makes the Mewar programming language unique. Instead of writing programs that only execute instructions, Mewar allows programs to express intention, reasoning, analysis, and internal state.

This makes programs more intelligent, understandable, and easier to maintain.

With the Intent System, a program can not only perform tasks but also understand its purpose and explain its behavior.

Core Features and Their Roles

Feature	Result
goal	Self-check
explain	Self-learning
tell	Data analysis
show	Debugging

goal → Self-Check

The goal statement defines what the program intends to achieve.
It helps the program stay aligned with its objective and makes it easier to verify whether the program's result satisfies the intended goal.

Example:

```
goal "Sum must be correct"
```

This clearly declares the purpose of the program.

explain → Self-Learning

The explain statement allows the program to describe how and why it performs an operation.

Example:

```
explain "Calculating total from input values"
```

This enables the program to behave like self-documenting code, making it easier for others to understand.

tell → Data Analysis

The tell command analyzes data and produces meaningful insights automatically.

Example:

```
tell sum of numbers

tell average of numbers
```

This allows the program to interpret and summarize data without writing complex logic.

show → Debugging

The show command displays the internal state of variables and data structures.

Example:

```
show numbers

show result
```

This helps developers inspect program values and debug problems easily.

10.4 Important Notes

The **Intent System** features such as **goal, explain, tell, and show** are unique characteristics of the Mewar programming language. These features are designed to make programs **more understandable, expressive, and intelligent**.

Unlike many traditional programming languages that only focus on executing instructions, Mewar allows programs to describe their purpose, explain their behavior, analyze data, and reveal their internal state.

10.5 Practice Questions

These practice questions help you understand and apply the concepts of **Intent-Based Programming** in the Mewar programming language.

Short Answer Questions

1. What is **Intent-Based Programming**?
2. What is the purpose of the **goal** statement?
3. How does the **explain** statement help programmers?
4. What type of operations can the **tell** command perform?

5. Why is the **show** command useful during program execution?

6. What is the main difference between **traditional programming** and **Intent-Based Programming**?

Practical Questions

1. Write a program that declares a goal and calculates the sum of two numbers.

Example idea:

```
goal "Calculate total"

set a to 5
set b to 10
set total to a + b

show total
```

2. Write a program that calculates a result and explains the operation using the **explain** statement.

Example idea:

```
set x to 10
set y to 20
set sum to x + y
explain sum
```

3. Create a list of numbers and analyze it using the **tell** command.

Example idea:

```
set numbers to [10,20,30,40]

tell sum of numbers
tell average of numbers
```

4. Write a program that displays the internal state of variables using the **show** command.

Example idea:

```
set a to 3
set b to 4
set result to a * b

show a
show b
show result
```

5. Write a small program that uses **goal, explain, tell, and show** together.

Example idea:

```
goal "Analyze numbers"

set numbers to [5,10,15]

tell sum of numbers
show numbers

explain "Program analyzes and displays data"
```

These exercises help you practice how **Mewar programs can express intention, explanation, analysis, and internal state**, which is the core idea of **Intent-Based Programming**.

Get Ready to Code Like a King!

Welcome to Mewar, a clear and powerful programming language designed for readability and intent-based coding. In this book, you'll master Mewar from the ground up, coding real-world projects as you learn.

Your Royal Journey Includes:

- ✓ Foundational Mewar Concepts
- ✓ Real Projects with Step-by-Step Guidance
- ✓ Best Practices & Tips for Better Code
- ✓ Advanced Features for Experts

> *A must-have guide for anyone who wants to learn Mewar and write robust, clear code*