TABLE OF CONTENTS

CHAPTER 1

Introduction

OVERVIEW

Interpreters have a special power – they take input and make it into output, making decisions in a simple way. But if you look closer, there's a complicated process of analyzing and calculating happening behind the scenes.

Consider a formula as straightforward as `10 + 2 * 2 + 10` your mind analyzes it as 24 through mathematical precedence. But how does an interpreter decode such a formula? Keep that question in mind.

Now, let me introduce you to DE It's an interpreted, functional, dynamic, high-level programming language. It is Javascript-Python-like, combining the two language syntaxes to create a straightforward and easy-to-understand language.

This document will serve as your comprehensive guide. We'll explain DE rules, showcase its capabilities, and introduce you to resources for hands-on exploration through a dedicated playground website.

Throughout this documentation, we're going to break down DE's design philosophy, and its features. By the end of this journey, you'll have a solid understanding of DE's fundamentals, and how it works.

CHAPTER 2

Exploring DE's Data Type & Basics

Welcome! In this section, we'll take a closer look at DE's basics and data type understand its core elements and also compare it with other languages like Javascript and Python.

In DE there are data types built in by default:-

- Numeric Types (int64, float64, decimal)
- Text Type (str)
- Sequence Types (Array, range)
- Mapping Type (dictionary)
- Boolean Type (bool)
- None Type (null)

Before diving into this section, I strongly recommend visiting the official playground website. You can access it using the following:

```
https://delang.mostafade.com
```

2.1 - Numeric Data Types and Mathematical Operations

This section explores the core building blocks of data handling in computing, encompassing numeric data types, and mathematical operations.

Numeric data types, such as integers (int), floating-point numbers (float), and decimals, play a crucial role in dealing with numerical values. They are the foundation for performing mathematical operations like addition, subtraction, multiplication, division, and more. These operations are not limited to numbers alone but can also be applied to strings, allowing for various text manipulations.

2.1.1 - Numeric Data Types

In DE, numeric data types are the foundation for handling numbers. They help us represent numerical values, and there are three main types to know: integers (int), floating-point numbers (float), and decimals. These types have specific purposes and characteristics, and they're crucial for various calculations and data operations.

Integer(int)

Integers are whole numbers without any fractional or decimal parts. They can be either positive, negative, or zero. Integers are typically used when precision to the decimal point is not required. In many programming languages, integers have a fixed range, meaning they can only represent values within a specific range, and attempting to store a value outside that range can lead to overflow or underflow.

In DE Integer numbers are represented as int64 type in Go and the range starts from `-9223372036854775808` to `9223372036854775807`.

Floating-Point Numbers (float):

Floating-point numbers, often referred to simply as floats, are used when precision to the decimal point is necessary. Floats can represent real numbers, including integers, with a fractional part. They are expressed using a decimal point, or in scientific notation. However, it's important to note that floats are approximations, and they may not always represent exact values due to the way they are stored in the computer's memory.

In DE they are represented as float64 type in Go language It can accept up to 17 digits and attempting to store a value outside that range will round the last digits after that.

```
1.922337203685475580732233332; // 1.9223372036854756
```

Floating Point: Issues and Limitations

Floating-point numbers are represented in computer hardware as base `2` (binary) fractions. Most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction `1 / 3`. You can approximate that as a base 10 fraction: `0.3` or `0.33` or `0.333` and so on. No matter how many digits you're willing to write down, the result will never be exactly `1 / 3` but will be an increasingly better approximation of `1 / 3`.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value `0.1` cannot be represented exactly as a base 2 fraction. In base 2, `1 / 10` is the infinitely repeating fraction

```
0.0001100110011001100110011001100110011001100110011...
```

For a more in-depth explanation, I highly recommend visiting https://docs.python.org/3/tutorial/floatingpoint.html

With that said, now you have an idea about why we want to avoid using float in our calculation, especially if we are dealing with money and precise calculations that can't allow any miscalculation or wrong approximation. Now let me introduce you to decimals.

Decimals

Decimals are used when high precision for decimal numbers is required, especially in financial or scientific calculations where rounding errors can have significant consequences. Unlike floats, decimals provide a fixed and exact representation of decimal numbers. They are less prone to rounding errors and are suitable for situations where accuracy is paramount.

Unlike other languages, DE comes with a built-in decimals system and to create decimals all you have to do is use the decimal built-in function, let's take an example

```
decimal(1.21113); // 1.21113
```

Let’s see the difference between float and decimals in action

```
1.2 - 1 // 0.19999999999999996
decimal(1.2) - decimal(1) // 0.2
```

But what if we want to control the precision of the digits, let’s say we want to round the number to be 2 digits.

You can achieve that by changing the `_getDecimalData` dictionary, which is a global object that contains `{'prec': 8, 'divPrec': 8}`, where prec is used for rounding any operation except the division and the mod operation, to round that you need to change the divPrec, let’s take an example to demonstrate that

```
decimal(1.21113) + decimal(2.22113); // 3.43226
_getDecimalData['prec'] = 3;
decimal(1.21113) + decimal(2.22113); // 3.432

decimal(300) / decimal(1.2121); // 247.50433133
_getDecimalData["divPrec"] = 10;
decimal(300) / decimal(1.2121); // 247.5043313258
```

Note that the allowed range for prec is from 1 to 8 and for divPrec is from 1 to 28 and attempting to store a value outside that range can lead to run time error.

2.1.2 - Mathematical Operations

In DE, the order in which mathematical operations are evaluated is essential for accurate calculations. DE follows standard mathematical operation precedence, ensuring that calculations are performed with the correct priority. This section sheds light on DE's handling of operation precedence, enhancing your understanding of how DE processes expressions.

Operation Precedence in DE

DE adheres to the natural order of mathematical operation precedence, as commonly known in mathematics. The precedence rules dictate that operations within parentheses are evaluated first, followed by multiplication and division, and finally addition and subtraction. This ensures that calculations are carried out in a manner consistent with mathematical conventions.

Consider the following DE expressions:

```
4 + 2 * 3; // 10
(4 + 2) * 3; // 18
```

In the first expression, DE follows the precedence rule and evaluates multiplication before addition, resulting in 10.

In the second expression, the parentheses alter the precedence, making the addition occur first, yielding 18.

Adition

The addition in DE for numbers like any programming language, you can add two integer numbers, two float numbers, an integer with float number or visa versa

```
1 + 10; // 11
1.2 + 1.2; // 2.4
```

Subtraction

The same thing applies to the subtraction process

```
1 - 10; // 9
1.2 - 1.2; // 0
```

Multiplication

The same thing applies to the multiplication process

```
1 * 10; // 10
1.2 * 1.2; // 1.44
```

Division

The same thing applies to the division process

```
1.0 / 10.0 // 0.1
1 / 10; // 0
```

Related to the division, if you divide two numbers with the same type, the result will be the same type.

As you can see in the above example dividing 1 over 10 will result in 0, the result should be 0.1 but because the data type is int64 it cuts the decimal digit from the result, so either the type should be float64 1.0 / 10.0 or one of them is float64 1 / 10.0, this might change in the future so the data type that return should depends on the result of the division.

Note it's recommended to use decimals for any calculation for precision.

2.2 - Strings

A string is a sequence of characters, which can include letters, numbers, symbols, and even spaces. They are used to represent and manipulate text, making them an essential component of many applications, from simple text processing to complex natural language processing.

Strings can be created by enclosing text within single (' ') or double (" ") quotation marks.

```
"Hello";
'World!';
```

Once you have a string, you can perform various operations on it, such as:

- Concatenation: Combining two or more strings together.

```
"Hello" + " " + 'World!'; // Hello World!
```

- Length: Finding the number of characters in a string.

```
len(name3); // 12
```

- Loop over characters in the string, **we will cover this later**.

Note that strings are often treated as immutable, meaning their contents cannot be changed after they are created. This immutability has several advantages, including data integrity and efficiency.

Once you create a string and assign a value to it, you cannot change the characters or content of that string. If you want to modify the string, you typically create a new string with the desired changes.

2.3 - DE's Variables and Declarations

DE's syntax for declaring variables closely resembles that of JavaScript. Let's explore how DE's variable declarations compare to other popular programming languages like Python.

By default, when you declare a variable, it becomes part of the global scope object unless you explicitly define it within a local environment, such as loops or functions. Variables within a local scope are isolated, which means they cannot be accessed from outside of that scope.

For instance, if you declare a variable within a function, that variable is considered local to that function. You won't be able to access it from outside the function, and it won't interfere with other variables of the same name in different parts of your code.

This concept of variable scope and isolation is crucial for maintaining data integrity and preventing unintended side effects in your code. It allows you to control where and how variables are used and helps ensure that your code behaves as expected.

DE lexical scoping (Static scoping)

In DE, the scope of a variable is determined by its location in the source code at the time of writing. DE follows a consistent and predictable scoping rule, which makes it a lexically scoped language.

- **Local Scope:** Variables defined within the current function are in a local scope (isolated). They have the highest priority and are the first to be looked up when you reference a variable inside the function.

- **Enclosing (Non-local) Scopes**: If a variable is not found in the local scope, DE will search in the enclosing (non-local) scopes, such as the scopes of any outer functions that enclose the current function.

- **Outer Scope:** If the variable is not found in any of the enclosing scopes, DE will look at the outer scope. Outer scope variables are those defined at the top level of your script or module.

- **If the variable couldn't be found in any scope, this will throw a run time error.**

Block declaration(environment / Scope)

The environment is a Go map also known as a hash map or hash table, it stores the DE values so we can access them afterwards when declaring a variable, it basically stores in the hash map like this

```
env.store = {
     "PI": Integer{Value: 3.14159}
}
```

We will cover the environment(object) concept in the upcoming chapters.

In DE, you use the let and const keywords to declare variables. These keywords are inherent to the language and serve the purpose of specifying whether a variable is mutable or immutable. Let's take a closer look at each of these keywords.

let for Mutable Variables

In DE, you can declare a variable using the let keyword, which denotes a mutable variable. This means you can assign and reassign values to the variable over its lifespan. Here's how you can define a variable named `"count"` and update its value:

```
let count = 0
count = count + 1
```

Comparison with Python

```
count = 0
count = count + 1
```

I understand that Python's syntax may appear simpler since it doesn't require specifying `let` keywords. However, I believe that this approach is more effective in distinguishing between mutable and immutable variables, a feature not present in Python. Therefore, I have opted for the approach used in JavaScript.

const for Immutable Variables

DE also supports constant variables using the const keyword. This signifies that the variable's value remains unchanged after initialization. Here's how you can declare a constant variable named PI:

```
const PI = 3.14159;
```

Comparison with Python

Python's equivalent to a constant variable is achieved by using uppercase letters for variable names, indicating that the value should not be changed conventionally, but from Python perspective, it’s not a constant so technically you can change it

```
PI = 3.14159
```

DE's approach to variable declarations strikes a balance between the straightforwardness of Python and the familiarity of JavaScript.

2.4 - Sequence Types

In the DE language, specialized sequence types are available to efficiently handle collections of data. These sequence types offer powerful methods for organizing and working with data.

2.4.1 - Array Data Structure

Arrays are custom-designed data structures that enable you to store collections of elements, such as numbers, strings, or even custom data types. Elements within DE arrays are ordered and can be accessed using index positions, much like other programming languages. DE arrays are versatile, allowing you to manage and manipulate datasets and implement various data structures.

Defining Arrays in DE

Defining an array in DE is simple and intuitive. Here's an example

```
const arr = ["DE", "Awesome", "!!"];
```

In this example, an array containing three elements: `"DE"`, `"Awesome"`, and `"!!"`. The elements are enclosed in square brackets and separated by commas.

Array Operations

Arrays in DE offer various operations to manage and manipulate their contents.

- You can access elements by their index

  ```
  arr[0]; // "DE"
  ```

- update values

```
arr[0] = "Lang";
arr[0]; // "Lang"
```

- add new elements
- iterate over the array using loops

Nested Arrays

You can create nested arrays, which are arrays that contain other arrays as their elements. This allows you to represent structured data with multiple levels or dimensions.

Let's say you want to create a nested array to store information about a group of students, including their names, ages, and grades. Here's how you can do it in DE:

```
const students = [
    ["Mostafa", 22, [85, 90, 78]],
    ["Aya", 26, [99, 92, 98]],
]
```

In this example, the students array contains three elements, each of which is itself an array. The inner arrays represent individual students and include their name, age, and an array of grades. You can access and manipulate this structured data using nested indexing.

For instance, to access aya's age, you can use:

```
const ayaAge = students[1][1]
```

Pointer References

In the DE language, when it comes to working with variables and data structures like arrays, it's important to understand how pointer references behave. Pointer references essentially point to the memory location of data, and this behaviour can lead to unexpected results if not managed carefully.

Consider the following example

```
const arr1 = [1, 2, 3];
const arr2 = arr1;
arr1[0] = "test"
```

At first glance, it might appear that arr2 should remain unaffected because it's assigned the value of arr1. However, due to pointer references, both arr1 and arr2 now point to the same memory location, resulting in the following:

- After `arr1[0] = "test"`, arr1 becomes `["test", 2, 3]`.
- Surprisingly, arr2 also becomes `["test", 2, 3]`.

This behaviour occurs because when arr2 is assigned the value of arr1, it's not a copy of arr1. Instead, both arr1 and arr2 reference the same memory location. Therefore, changes made to one are reflected in the other.

If you intend to create a separate copy of an array to avoid this behaviour, you should use a built-in function called `copy()`

```
const arr1 = [1, 2, 3];
const arr2 = copy(arr1);
arr1[0] = "test";
arr1; // ["test", 2, 3]
arr2; // [1, 2, 3]
```

2.4.2 - Range Function

the Range function is a built-in function that provides a convenient way to create an array that contains a sequence of numbers within a specified range. This function is particularly useful for creating loops, iterating through data, and performing repetitive tasks. The Range function allows you to specify a range of numbers.

Here's how the Range function works in DE

```
let myRange = range(1, 6)
```

In this example, we create a range that starts at 1 and ends at 6. The result is a sequence of numbers: `[1, 2, 3, 4, 5, 6]`. The ending value is inclusive, meaning it is included in the range.

You can then use this range in a loop to perform actions or iterations over the specified range, we will cover this soon.
Also, you can specify any range whether it's a negative or positive

```
range(-5, 5); // ⇒ [-5, -4, ..., 0, 1, ..., 4, 5]
```

2.5 - Mapping Type (dictionary)

In the DE language, the mapping type, often referred to as a dictionary or hash is a versatile and fundamental data structure. It allows you to store and manage key-value pairs, associating unique keys with corresponding values. This data structure is immensely valuable for tasks that involve looking up values based on specific keys, such as creating data structures, implementing efficient search algorithms, and organizing information.

Key-Value Pairs

A dictionary, or mapping consists of key-value pairs. Each key is unique within the dictionary and maps to a specific value. This key-value pairing allows you to associate information in a structured manner.

One of the primary advantages of a dictionary is its efficient retrieval O(1) of values based on keys. This data structure is designed to provide quick access to values, even when dealing with large datasets.

Flexible Data Storage

Dictionaries in DE are highly flexible in terms of the data they can store. You can use keys and values of various data types, including strings, numbers, and even complex objects. This flexibility makes dictionaries suitable for a wide range of applications.

Here's how the mapping type (dictionary) works in DE:

```
const _dict = {
    "name": "Mostafa",
    "age": 30,
```

```
    "city": "Amman"
}
```

In this example, _dict is created as a mapping that contains key-value pairs. Each key is unique and associated with a specific value. For instance, name is a key mapped to the value Mostafa, age is a key mapped to the value 30, and city is a key mapped to the value Amman.

You can access values in the dictionary using their keys:

```
_dict["name"]; // "Mostafa"
```

Why We Hash?

In DE, which is built on the Go programming language and leverages the Go map type to represent dictionaries, hashing plays a critical role in ensuring efficient data retrieval. Understanding the significance of hashing is essential, especially when considering the behaviour of pointers in Go.

The Go Language Foundation

DE, while being its language, is built upon the Go programming language, which serves as its foundation. Go, like many other languages, uses the map data structure to represent dictionaries. However, Go's behaviour with pointers introduces unique challenges when working with objects.

Challenges with Pointers in Go

In Go, when you work with pointers to objects, such as strings, each pointer references a specific memory location. Even if two pointers contain the same content (e.g., both pointers contain "name"), they are not considered equal because they point to different memory locations. This means that using one pointer to access data will not provide access to data pointed to by another, even if their content is identical. This behaviour is inherent to Go's pointer system.

Hashing for Equality and Efficiency

To address the challenges posed by Go's pointer behaviour, DE implements a HashKey system. This system generates unique hash keys for objects, ensuring that hash keys are comparable and equal for objects with the same content, such as strings containing "name." The primary goal is to make sure that hash keys for identical content objects match while remaining distinct from the hash keys of different object types like integers and booleans.

The HashKey Solution in DE

In DE, the HashKey system resolves the issue of unequal pointers and allows consistent access to data. It does this by creating hash keys that are based on various properties of an object, including its content. These hash keys are consistent and allow for efficient and predictable data retrieval.

2.6 - Boolean Type (bool)

In the DE language, the Boolean type, represented as bool, is a fundamental data type used to work with logical values. Boolean values can have one of two possible states: true or false. These values are essential for expressing and evaluating logical conditions and making decisions within your code.

Here are key points to understand about the Boolean type in DE:

Two Possible Values

The Boolean type has two distinct values: true and false. These values represent the truth or falsity of a logical statement.

Logical Operations

Booleans are commonly used in logical operations and comparisons. You can use Boolean values to evaluate conditions and control the flow of your program.

Conditional Statements

Boolean values are integral to conditional statements like if, else, and while. These statements allow you to execute code based on whether a condition is true or false.

Real-World Uses

Boolean values are everywhere in programming. They control who can access a website, whether you've correctly entered your password, or if your game character can jump. They're essential for automating decisions and making your programs more interactive.

```
const isUserLoggedIn = true;

if isUserLoggedIn: {
    if userAge >= 18: {
        logs("Welcome to the website!");
    }
} else {
    logs("Please log in to access the website.");
}
```

2.7 - None Type (null)

In the DE language, the null is a fundamental data type that represents the absence of a value or the lack of a meaningful value. It's a special placeholder used to indicate that a variable or object does not currently possess any data.

Absence of Value

The Null type is used to indicate the absence of a valid value. It is typically employed when a variable or object needs to be initialized but doesn't have a meaningful value to assign yet.

```
let name;
name; // null
```

CHAPTER 3

Flow Controls & Functions

In this chapter, we're going to focus on two key pillars of programming: Flow Control and Functions. Flow control helps you steer the direction of your code, making it dynamic and responsive, while functions are like building blocks that allow you to organize and reuse code. These two concepts are fundamental in the world of programming, and mastering them will unlock new possibilities for your coding journey. So, let's dive in and explore how flow control and functions can supercharge your programs.

3.1 - Flow Controls

Let's explore how the flow controls are constructed in DE, along with a brief comparison to similar constructs in other languages.

Conditional Statements

if Statement: Basic Decision-Making

The if statement is the fundamental tool for branching your code. It allows you to check if a condition is true and execute a specific block of code when the condition holds. You can think of it as the binary decision-maker in your code, enabling you to take one path if the condition is met and another if it's not.

Here's an example that demonstrates the if statement in action:

```
if 4 > 2: {
    logs("DE Awesome!!");
}
```

In this example, condition `4 > 2` is evaluated. Since the condition is true, the block of code within the curly braces after it is executed, resulting in the message `"DE Awesome!!"` being logged to the console.

else Statement: Adding Alternatives

Sometimes, a single decision isn't enough. That's where the "else" statement comes in. It complements the "if" statement, offering an alternative path when the condition is not met. This opens the door to more nuanced decision-making in your code, ensuring that even when the initial condition fails, your program can still take a different route.

```
if 4 > 2: {
     logs("DE Awesome!!");
} else {
     logs("DE Still Awesome!")
}
```

Comparison with Python:

```
if 4 > 2:
    print("DE Awesome!!")
else:
    print("DE Still Awesome!")
```

Comparison with Javascript:

```
if (4 > 2) {
    console.log("DE Awesome!!");
} else {
    console.log("DE Still Awesome!");
}
```

You might think that Python makes it appear simple, but here's the catch! In Python, spacing plays a crucial role, and a missing space can disrupt proper execution. Unlike DE, you don't need to worry about spacing; just include the curly brackets `"{}"` and you're good to go. Similarly, in JavaScript, the use of parentheses `"()"` can seem less intuitive and straightforward by comparison.

for Loop

The for loop is your key to performing repetitive tasks with exact control. It enables you to execute a block of code for each item in a sequence. In this section, we'll explore the syntax and functionality of the for loop, teaching you how to create precise and efficient iterations. Whether you're processing data, performing calculations, or automating tasks, the for loop is a valuable tool for you to use.

In DE the for loop is similar to the one in Python, the only difference is that in DE you have access to the index variable which is something you have to use a function in Python to get it.

for Loop Syntax

In DE for loop consists of three parts:

- **Initialization:** This is where you set an initial identifier that will hold the value of each item in the expression (Array).
- **Expression:** Any iterable expression (Array, String) that you can loop over.
- **Body:** The main body of the loop, the piece of code that you want to run on every iterator for each item in the expression.

Here are different ways to write a for loop in DE

Iteration without Index:

```
for num in iterable(collection): {
 ...
}
```

Iteration with Index:

```
for index, num in iterable(collection): {
```

```
 ...
}
```

Ignoring the Index:

```
for _, num in iterable(collection): {
 ...
}
```

In these examples, iterable(collection) represents the collection you want to iterate over. The for loop allows you to process each element in the collection, either with or without access to its index, depending on your specific requirements.

collection refers to the iterable data structure you want to loop through.

Iterating Through Lists

In DE for loop you can set a for loop that is traversing lists. For example

```
numbers= [1, 2, 3]
for num in numbers: {
     logs(num)
}
```

Note: logs function is a built-in function that you can use to print the output to the screen.

The above loop iterates through the numbers list and prints each number. DE's for loop can also be used to iterate through strings to examine individual characters.

Iterating Through Strings

In DE, strings are fundamental data types, and they often need to be processed character by character. To achieve this, we use for loops to iterate through each character in a string. This process is essential for tasks like text analysis, data extraction, and string manipulation

```
text = "Hello, World!"
```

```
for char in text: {
     logs(char)
}
```

Iterating Through Range-Based

DE provides the `range()` function, allowing you to create an array with a range of numbers. You can use it in conjunction with a for loop to control the number of iterations.

```
for number in range(1, 6): {
   logs(number)
}
```

Note: This loop generates numbers from 1 to 6 and prints them.

Note: The loop continues to execute as long as the iterable expression contains items to process.

Nested for Loops

DE like any other language also supports nested for loops, allowing you to create more complex patterns and handle multidimensional data.

```
for num in [1, 2, 3]: {
    logs("Start From Number ⇒  " + num);
    for num in [10, 20, 30]: {
        logs(num);
    }
}
```

Using the break Statement

The break statement is a crucial tool in DE that allows you to exert control over your loops. It serves as a powerful means to exit a loop prematurely, based on a specified condition. The break statement is particularly valuable when you want to stop the execution of a loop as soon as a particular condition is met.

```
for item in iterable: {
    if condition: {
        break;  # Exit the loop
```

```
    }
}
```

- iterable represents the sequence or collection you're looping through.
- condition is the trigger that, when met, leads to the execution of the break statement

The primary purpose of the break statement is to terminate a loop prematurely. This is often used when you've found what you're looking for, and there's no need to continue iterating. For example, when searching for a specific item in a list, you can use break to stop as soon as you find it.

```
const numbers = [1, 2, 3, 4, 5]
const target = 4

for number in numbers: {
    if number == target: {
        logs("Target found!");
        break;
    }
    logs(number); // 1 2 3
}
```

Using the Skip Statement

The skip statement is a powerful tool in DE that offers you precise control over loop execution. Unlike the break statement, which exits the loop entirely, skip allows you to skip the current iteration of a loop and move directly to the next one. This is particularly useful when you want to skip specific items in an iterable or apply different processing logic to different elements.

```
for item in iterable: {
    if condition: {
```

```
        skip; # Skip the current iteration
    }
}
```

skip is often used when you want to skip certain items in an iterable based on a condition. For instance, when processing a list of values, you can use "continue" to bypass values that don't meet specific criteria.

```
const numbers = [1, 2, 3, 4, 5];

for number in numbers: {
    if number % 2 == 0: {  // Skip even numbers
        skip;
    }
    logs("Odd number: " + number);
}
```

When dealing with more intricate loops that involve multiple conditions, skip helps you maintain clarity and readability. You can apply it to efficiently manage complex processing scenarios within the loop.

```
for item in items: {
    if !(meets_condition_1): {
        skip;
    }
    if !(meets_condition_2): {
        skip;
    }
    // Process the item
}
```

During Loop

The during statement is a versatile loop construct in DE that allows you to execute a block of code repeatedly as long as a specified condition remains true. Unlike the for loop, which iterates through a predetermined collection, the during loop continues execution based on the evaluation of a condition. This makes it an ideal choice when the exact number of iterations is uncertain or when you want to perform tasks while a particular condition holds true.

** Related to the naming, I'm pretty sure that you are wondering why not call it while like any other language? What is the point?

To be honest, I didn't have a specific reason for the naming, I just wanted a different name, so I went to GPT to give me some alternative names for while one of them was during so I picked it up and implemented.

The during statement follows a straightforward syntax:

```
during condition: {
    // Code
}
```

condition represents the expression that determines whether the loop should continue. As long as this condition evaluates to true, the loop persists.

Working Mechanism:

- The during loop begins by evaluating the initial condition.
- If the condition is true, the code block within the loop is executed.
- After executing the code block, the condition is re-evaluated.
- If the condition remains true, the loop continues; otherwise, it terminates.

Note: The "skip" and "break" statements function in the same way as they do in a standard "for" loop, with no distinctions.

during loops are ideal for creating interactive menu systems that allow users to choose from various options. Here is a simple example

```
during true: {
    logs("1. Option 1");
    logs("2. Option 2");
    logs("3. Exit");
    let choice = input("Select an option: ");
    if choice == "1": {
        logs("You select option 1");
    }

    if choice == "2": {
        logs("You select option 2");
    }

    if choice == "3": {
        break;  // Exit the loop
    }
}
```

Infinite Loops and Termination

One cautionary aspect of during loops is the potential for infinite loops if the termination condition is not met. To prevent infinite loops, it's essential to ensure that the condition eventually becomes false during the loop's execution.

Note:- Every statement within the flow control structures adheres to the global and local scope concept, as described in DE's Variables and Declarations section

3.2 - DE’s Functions

Functions are the building blocks of any programming language, enabling code organization and reusability.

In DE, functions are different from traditional named functions; they are anonymous by default. This means that in DE, to work with functions, you must assign them to variables. These anonymous functions, often referred to as "anonymous function literals," grant you the flexibility to create and employ functions on-the-fly as needed.

Defining Anonymous Functions

```
const f = fun() {
    logs("DE!");
}
```

This snippet demonstrates the creation of an anonymous function that, when invoked, logs "DE!" to the console. The function is assigned to the variable f, making it accessible for execution.

Advantages of Anonymous Functions

- **Dynamic Function Creation:** Anonymous functions provide the flexibility to create functions on-the-fly, enabling your code to adapt to specific requirements as they arise.

- **Conciseness:** They are ideal for shorter, task-specific operations, reducing the need for lengthy named function definitions.

- **First-Class Citizen Status:** In DE, functions are treated as first-class citizens, meaning they can be passed as arguments to other functions, returned from functions, and assigned to variables just like any other data type.

Function Invocation

To execute an anonymous function in DE, you simply invoke it by calling the assigned variable. For instance, you can call the f function defined earlier as follows:

```
f(); // This will log "DE!" to the console.
```

The parentheses after the function name indicate that the function should be executed. In this case, f would log "DE!" to the console.

Function Scoping

It's essential to understand that variables defined within an anonymous function are typically local to that function. This ensures that they don't interfere with variables outside of the function, adhering to DE's scoping rules.

CHAPTER 4

Built-In Functions

DE come with a range of built-in functions designed to simplify common tasks. These functions are ready to use out of the box, saving you time and effort. This section introduces you to some of DE's essential built-in functions and how they can enhance your coding experience.

What Are Built-In Functions?

Built-in functions, often referred to as "standard library functions" or "library functions," are predefined functions that come bundled with the DE language. They are available for use without additional setup or external libraries. These functions encapsulate common operations and functionalities, allowing you to perform various tasks straightforwardly and efficiently.

logs Function

You have already seen this function in the previous chapters, and you likely know what it does for you. It enables you to print data to the console, allowing you to monitor variables, expressions, and intermediate results during code execution.

```
const message = "Hello, DE!";
logs(message); // Output: Hello, DE!
```

len Function

The len function comes to the rescue when you need to know the length of an array or the number of characters in a string.

```
const myArray = [1, 2, 3, 4];
const arrayLength = len(myArray); // Returns 4
const myString = "DE";
const stringLength = len(myString); // Returns 2
```

first Function

The first function fetches the first item in an array without altering the original array. It's handy for quick access to array elements.

```
const arr = [5, 6, 7];
const firstItem = first(arr); // Returns 5
logs(arr); // Returns [5, 6, 7]
```

last Function

The last function retrieves the last item in an array without modifying the original array. It's perfect for grabbing the final element.

```
const arr = [8, 9, 10];
const lastItem = last(arr); // Returns 10
logs(arr); // Returns [8, 9, 10]
```

skipFirst Function

The skipFirst function discards the first item in an array and generates a fresh array, leaving the original array intact.

```
const arr = [11, 12, 13];
const newArr = skipFirst(arr);
logs(newArr); // Returns [12, 13];
logs(arr); // Returns [11, 12, 13], (arr) remains unchanged
```

skipLast Function

The skipLast function dismisses the last item in an array, crafting a new array while preserving the original one.

```
const arr = [14, 15, 16];
const newArr = skipLast(arr);
logs(newArr); // Returns [14, 15];
logs(arr); // Returns [14, 15, 16], (arr) remains unchanged
```

push Function

The push function adds an item to the end of an existing array, expanding its size dynamically. Adding to the existing array means that the original array will be changed.

```
const arr = [1, 2];
push(arr, 3);
logs(arr); // Output [1, 2, 3]
```

pop Function

The pop function is your go-to for removing and receiving the last item from an array, affecting the original array.

```
const arr = [4, 5, 6];
const poppedItem = pop(arr); // poppedItem is 6, and arr is now [4, 5].
```

typeof Function

The typeof function identifies the data type of the input you provide, whether it's a string, number, or other type.

```
const myVar = "DE";
const varType = typeof(myVar); // Returns "STRING".
```

copy Function

The copy function is your companion for creating deep copies of data structures, whether they're arrays, hashes, or strings. It ensures that you work with separate copies rather than references to the original data.

```
const originalArray = [1, 2, 3];
const copiedArray = copy(originalArray);
copiedArray[0] = 10;

logs(originalArray); // Outputs [1, 2, 3] (original array
remains unaffected).

logs(copiedArray); // Outputs [10, 2, 3] (copied array is
modified).
```

We already explained the pointer reference issue, if you would like a reminder, please refer to Pointer References

input Function

The input function takes input from the user as a string, similar to Python's input function. It's a valuable tool for interactive programs that require user interaction.

```
const userName = input("Enter your name: ");
logs("Hello, " + userName + "!");
```

Type Conversion Functions

int Function

The int function converts the input to an integer. If the input is a floating-point number or a decimal, it truncates to the integer part.

```
const intFromString = int("42"); // Result: 42
const intFromFloat = int(1.5); // Result: 1
const intFromDecimal = int(decimal(2.7)); // Result: 2
```

float Function

The float function converts the input to a floating-point number.

```
const floatFromString = float("3.14"); // Result: 3.14
const floatFromDecimal = float(decimal("3.14"))// 3.14
```

Note that if you tried to convert int to float, the return will look like int, but its type will be “FLOAT”

```
const floatFromInt = float(7); // Result: 7
logs(typeof(floatFromInt)); // Result: FLOAT
```

bool Function

The bool function converts the input to a boolean value. Most values are considered true, except for specific cases like zero and empty strings.

```
const boolFromString = bool("true"); // Result: true
const boolFromZero = bool(0); // Result: false
```

str Function

The str function is used to convert an input to a string data type. It's a versatile tool for handling various data types and ensuring they are represented as strings.

```
const numToStr = str(42); // Result: "42"
const boolToStr = str(true); // Result: "true"
```

CHAPTER 5

Advanced Features

In addition to its approachable syntax, DE empowers developers with advanced features that enhance code flexibility. This section delves into features like Higher-Order Functions (HOFs) and Closures, SIF, etc.

5.1 - Higher-Order Functions (HOFs)

Higher-Order functions HOFs are a powerful concept in programming. In DE, HOFs allow functions to take other functions as arguments or return functions as results. This elegant capability enables you to build more abstract and adaptable code structures.

Example

```
let add = fun(x, y) {
    return x + y;
}

let HOF = fun(f, x, y) {
    return f(x, y);
}

HOF(add, 1, 2);
```

In this example, we have two functions: `"add"` and `"HOF"`. The `"add"` function takes two parameters, `"x"` and `"y"`, and returns their sum. The `"HOF"` function takes three arguments: a function f and two values, `"x"` and `"y"`. Inside `"HOF"`, it calls the function `"f"` with `"x"` and `"y"` as arguments.

Finally, we call `"HOF(add, 1, 2)"`, passing the `"add"` function as the first argument and the values 1 and 2 as the second and third arguments. This results in 3 because `"add(1, 2)"` returns 3.

Use Cases

Higher-order functions find use in various scenarios:

- **Function Composition:** You can compose multiple functions together using higher-order functions to create new functions.

- **Dependency Injection:** In dependency injection patterns, higher-order functions are used to inject dependencies into functions.

Benefits

Higher-order functions provide several benefits:

- **Modularity:** You can create small, specialised functions that can be combined to build more complex functionality. This promotes code modularity and reusability.

- **Abstraction:** HOFs allow you to abstract away common patterns and behaviours, making your code more concise and easier to understand.

- **Flexibility:** You can pass different functions to higher-order functions, changing their behaviour dynamically. This flexibility is valuable for handling various use cases

5.2 - Closures

Closures are a fundamental concept in DE that enable functions to "remember" and access variables from their containing scope even after that scope has exited. This behaviour allows for powerful and flexible coding patterns.

```
let f = fun(x) {
    return fun(y) {
        return x + y;
    };
};

let apply = f(1);

apply(2);
```

In this example, we define a function f that takes a parameter x. Inside f, another anonymous function is defined and returned. This inner function takes a parameter y and returns the sum of x and y.

We then create a new variable apply, and set it equal to the result of calling f(1). apply effectively "remembers" that it was created in the scope of f(1) with x equal to 1.

Finally, when we call apply(2), it uses the remembered value of x, which is 1, and adds it to the new parameter y, resulting in 3.

How Closures Work

Closures "capture" variables from their containing scope. In our example, the inner function captures the variable `"x"` from the outer function `"deFun"`. This captured variable `"x"` is retained even after `"deFun"` has finished executing.

Use Cases

Closures are incredibly versatile and find use in various scenarios:

- **Data Encapsulation:** Closures allow you to create private variables or functions, as they are only accessible within the closure. This supports data encapsulation and helps prevent unintended modifications.

- **Callbacks:** Closures are frequently used in callback functions, allowing you to pass functions as arguments to other functions and maintain the state.

- **Factories:** You can use closures to create factory functions that generate instances of objects with shared behaviour and state.

Benefits

Closures provide an elegant way to manage and encapsulate data and behaviour in your code. They facilitate the creation of modular and reusable code while maintaining data integrity.

5.3 - Self-Invoking function

In DE, self-invoking functions are a powerful feature that allows you to execute a function immediately after it's defined, without the need for a separate function call. This concept is also known as an Immediately Invoked Function Expression (IIFE) in other programming languages.

You can create a self-invoking function using the following syntax:

```
fun() {
  // Function body
}()
```

fun(): This is the declaration of the function. You define the function just like you would with any other function.

{}: Inside the curly braces, you can place the code that you want to be executed immediately.

(): These parentheses immediately follow the function definition and invoke the function.

Example

Let's look at a practical example of a self-invoking function in DE:

```
fun() {
  let result = 10 + 5;
  logs("The result is:", result);
}()
```

In this example, the fun function is defined and immediately invoked. It calculates the sum of 10 and 5 and then prints the result. You'll notice that there's no need to call fun() separately; it runs automatically as soon as it's defined.

Use Cases

Self-invoking functions are commonly used for tasks that need to be executed immediately or for creating isolated scopes. Some common use cases include:

- **Isolating Variables:** You can use self-invoking functions to create a scope where variables are isolated from the global scope. This is useful for preventing variable name clashes.

- **Initialization:** They are handy for initializing values or setting up configurations as soon as your script starts running.

- **Immediate Execution:** Whenever you need a piece of code to execute right away, without being explicitly called elsewhere in your program.

Benefits

The primary advantage of self-invoking functions is encapsulation. They allow you to create a private scope for variables and functions, reducing the risk of naming conflicts and keeping your code clean and modular.

5.4 - Recursion function

In DE, recursive functions are a powerful feature that allows a function to call itself during its execution. This elegant technique can be harnessed to solve complex problems by breaking them down into simpler, self-referential subproblems. DE fully supports recursion, enabling you to implement recursive solutions for a wide range of tasks.

At the heart of recursion lies the concept of self-reference. A recursive function calls itself with modified arguments to solve a part of the problem. This process continues until a base case is reached, which is a condition that terminates the recursive calls. Recursive functions are structured with two main components:

- **Base Case:** The base case defines the condition under which the recursion stops. It's a critical element to prevent infinite loops and ensure the termination of recursive calls.

- **Recursive Case:** The recursive case involves calling the function itself with modified arguments. Each recursive call should move closer to the base case, ensuring that the problem gets simpler with each iteration.

Example

Let's explore a classic example of recursion: calculating the factorial of a number. The factorial of a non-negative integer n, denoted as n!, is the product of all positive integers from 1 to n.

```
const factorial = fun(n) {
    if n == 0:
        return 1; // Base case: factorial of 0 is 1
    } else {
        return n * factorial(n - 1);
    }
}

const num = 5;
const res = factorial(5);

logs(res); // Returns: 120
```

In this example, the base case is when n is equal to 0, and the recursion stops by returning 1. For any other value of n, the function recursively calls itself with a smaller argument, moving towards the base case.

Use Cases

- **Solving problems** that exhibit a recursive structure, such as calculating factorials.
- **Traversing** hierarchical data structures like trees.
- **Implementing** search algorithms like binary search.

CHAPTER 6

Tools and Methods: Brief Discussion

Welcome! Here, we'll take a peek behind the scenes of DE's creation. We'll explore the tools, technologies, and methods that have helped shape DE into the user-friendly language you've come to know.

So, let's jump in and explore the building blocks that have contributed to DE's creation.

6.1 - Go Language: The Core of DE’s Design

At the heart of DE's creation lies the deliberate choice of the Go programming language. Go, often referred to as Golang, serves as the foundation for building DE due to its balanced blend of simplicity, efficiency, and concurrency support.

Why Using Go

- **Clean Syntax:** Go's elegant and uncomplicated syntax aligns seamlessly with DE's commitment to readability and approachability. The clean and minimalistic design of Go enhances the clarity of DE code, making it more accessible to developers and easier to maintain.

- **Efficiency:** Go's optimized performance ensures that DE executes code swiftly, minimizing resource consumption.

- **Concurrency:** The concurrent programming paradigm offered by Goroutines empowers DE to manage multiple tasks concurrently, enabling responsiveness and interactive features.

6.2 - Lexical Analysis: Decoding DE's Source Code

In DE, the core structure uses something called lexical analysis. This tool breaks down the original code into important parts. It's done by a tool called the lexical analyzer, which carefully picks out tokens that makeup DE's coding structure.

Tokenization: Breaking Down the Code

Lexical analysis performs tokenization, which involves dividing the source code into discrete tokens. These tokens encompass keywords, identifiers, symbols, and more. By categorizing the code's elements, tokenization sets the stage for the subsequent steps of interpretation.

6.3 - Parsing: Constructing the Abstract Syntax Tree (AST)

Parsing takes the baton from lexical analysis and transforms the tokens into an Abstract Syntax Tree (AST). This tree-like structure captures the code's syntactic hierarchy, offering a blueprint that the interpreter can follow for precise execution.

Navigating the AST: Guiding Interpretation

The AST acts as an intermediary between the parser and the interpreter. Its hierarchical arrangement enables the interpreter to traverse the code systematically, node by node. This traversal leads to the execution of the corresponding actions during the evaluation process.

Interpreter and Execution Engine: Bringing Code to Life

The interpreter, custom-crafted for DE, collaborates with the execution engine to give life to the AST. It takes the parsed and structured code and executes it, producing tangible outcomes that developers can observe.

6.4 - Tree Traversal: Unveiling DE's Syntax

Within DE's interpretation realm lies the essential technique of tree traversal, a method that underpins DE's ability to execute code accurately and systematically. This process involves navigating the Abstract Syntax Tree (AST) generated from parsed code, enabling the interpreter to execute code instructions step by step.

Navigating the AST: A Step-by-Step Journey

DE's interpreter embarks on a journey through the AST, systematically visiting each node. As it traverses, the interpreter extracts information and performs the corresponding actions dictated by the code's structure. This sequential approach ensures the accurate execution of DE's code.

6.5 Pratt Algorithm (TDOP)

DE's adept handling of operator precedence is empowered by the Pratt algorithm, also known as Top Down Operator Precedence (TDOP). This algorithm is a cornerstone in parsing expressions, assigning precise precedence levels to operators, and managing their interactions seamlessly.

Parsing Expressions with Precision

When faced with an expression, the Pratt algorithm skillfully identifies the appropriate parsing rule, guided by the operator's precedence and associativity. This dynamic approach ensures that operators are evaluated in the correct order.

Do you Remember this example `10 + 2 * 2 + 10` ? We will visit it soon in the upcoming section while explaining the (TDOP), so stay tuned

6.6 - The Objects

We have to take a look at the important concept of DE (Objects). Yes, you read it correctly DE has an object. But wait, did I mention before that DE is a functional programming language? Now I'm saying it's object-oriented! No, I'm not saying that and it is not object-oriented, but we still need an object system, or let's say a value system.

The reason is that we still need a system that can represent the values our AST (Abstract syntax tree) represents or values that we generate when evaluating the AST in memory. As we discussed the AST is just a data structure (Tree) that represents our code to be ready to evaluate.

Let's say we're evaluating the following DE code which is a valid DE syntax:

```
let a = 5;
// ...
a + a;
```

As you can see, we're binding the integer 5 to the name a. What matters is that when we come across the `a + a` expression (Anything that produces a value) later we need to access the value a. To evaluate `a + a` we need to get to the 5. In the AST it's represented as an `*ast.Integer`, but how are we going to keep track of and represent the 5 while we're evaluating the rest of the AST?

There are a lot of different choices when building an internal representation of values in an interpreted language. And there is a lot of wisdom about this topic spread throughout the codebases of the world's interpreters and compilers. Each interpreter has its own way of representing values, always slightly differing from the solution that came before, adjusted for the requirements of the interpreted language.

Why the variety? For one, the host languages differ. How we represent a string of your interpreted language depends on how a string can be represented in the language the interpreter is implemented in. An interpreter written in Go can't represent values the same way an interpreter written in C++ can.

In DE the choice was to follow this, represent every value we encounter when evaluating code as an Object, an interface of our design. Every value will be wrapped inside a struct, which fulfils this Object interface in Go.

```
package object
type ObjectType string
type Object interface {
    Type() ObjectType
    Inspect() string
}
```

And here is how to represent integers:

```
type Integer struct {
    Value int64
}
```

So in simple terms, Object is a good way to keep track of values we encounter when executing DE source code.

6.7 - The Read-Eval-Print Loop (REPL) Implementation

DE's Read-Eval-Print Loop (REPL) is where the magic happens – it's the interactive playground where you can experiment with code in real time.

Bringing Interactivity to Code

The REPL relies on the concept of a loop, which constantly prompts the user for input, evaluates that input, and then displays the result. This iterative process is the foundation of the interactive coding experience DE offers.

Capturing User Input with Ease

To capture user input in real-time and provide immediate feedback, DE utilizes the `"github.com/eiannone/keyboard"` package. This package enables the program to read keyboard input directly, allowing you to type out code and see the output.

The keyboard package simplifies the process of capturing individual keystrokes and handling special key combinations, providing a smooth and intuitive interaction with the REPL.

I will explain in detail how the REPL was implemented in DE in the upcoming sections, so keep reading.

6.8 - Backend Server: Enabling Frontend Interaction

Playground backend server, meticulously implemented using Go's built-in `"net/http"` package, serves as a crucial link between the frontend playground and the core of the language.

By defining specific endpoints, the server enables the frontend to communicate effectively. This communication includes sending code to be executed, receiving results, and managing the interactive experience.

At its core, the server utilizes Go's powerful `"net/http"` package to handle incoming HTTP requests and generate responses. This ensures a robust and reliable connection between the user interface and the backend logic.

The backend server plays a pivotal role in executing code sent from the frontend. The code is processed and evaluated within the DE interpreter, and the results are then communicated back to the frontend for display.

We will dive into the server in the server section, so stay tuned!

6.9 - Frontend Tools

The frontend playground of DE is a dynamic environment where code meets execution, enabling developers to experiment and see results in real time. The playground's functionality and aesthetics are powered by two main tools: Solid JS and Tailwind CSS.

Solid JS

At the core of the frontend playground's interactivity lies Solid JS, a reactive JavaScript library. Solid JS brings reactivity and performance optimization to the forefront, ensuring that code changes are instantly reflected in the user interface. Its reactive nature minimizes unnecessary re-renders, making the playground responsive and smooth.

Tailwind CSS

Crafting the visual appeal of the frontend playground is Tailwind, a utility-first CSS framework. Tailwind's approach allows for quick and flexible styling through pre-defined classes, speeding up frontend development. This framework enables the creation of a sleek and consistent design, enhancing the user experience.

The combination of Solid and Tailwind results in a frontend playground that is not only highly functional but also visually engaging.

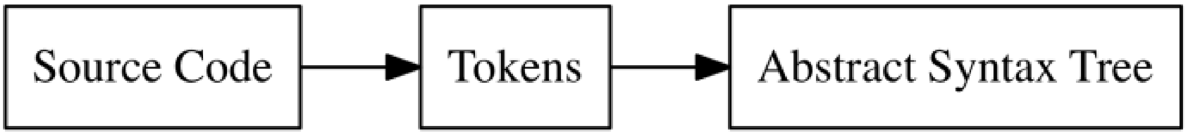
CHAPTER 7

The Interpreter: Unraveling DE's Code Execution

At the heart of DE's functionality lies its interpreter. This fundamental component is responsible for taking your carefully crafted code and turning it into executable actions. Let's dive into the inner workings of the interpreter and see how it transforms code into action.

7.1 - Lexical Analysis: The First Step

When it comes to interpreting code, the process begins with transforming raw source code into a more digestible format. Although plain text is convenient for human eyes, it's not the ideal medium for a programming language to comprehend. That's where lexical analysis, or lexing, steps in to bridge the gap.



Transitioning to Tokens

Picture this: your source code as a message and the interpreter as its recipient. But instead of words, the interpreter requires tokens – basic, recognizable units of code. Lexical analysis is like translating your code into tokens, making it easier for the interpreter to comprehend. Think of tokens as building blocks: they consist of keywords, identifiers, operators, literals, and symbols.

The Role of the Lexer

The initial transformation, from source code to tokens, is facilitated by a lexer (also known as a tokenizer or scanner). It's the lexer's job to scan through your code and create tokens based on recognized patterns. Tokens are then passed on to the parser for further processing, where they are used to construct the Abstract Syntax Tree (AST).

Tokenization in DE

The tokenization process is a critical step in the compilation and interpretation of DE code. Let's delve deeper into how the lexer operates and what tokens represent.

Scanning Source Code

Imagine you feed the lexer the following source code: `let x = 5 + 5;`
The output of the lexer could resemble the following tokens:

```
[
    LET,
    IDENT("x"),
    ASSIGN,
    INT(5),
    PLUS,
    INT(5),
    SEMICOLON
]
```

- LET corresponds to the let keyword.
- IDENT("x") represents an identifier, specifically the variable name x.
- ASSIGN corresponds to the assignment operator "=".
- INT(5) is an integer token representing the number 5.
- PLUS corresponds to the addition operator +.
- INT(5) is another integer token for the number 5.
- SEMICOLON represents the semicolon, marking the end of the statement.

Token Categories

Tokens are organized into specific categories that the interpreter can recognize. Some common token categories in DE include:

- **IDENT:** Identifiers for variables and function names.
- **INT:** Integers representing numerical values.
- **STRING:** Strings enclosed in double or single quotes.
- **Operators:** Tokens for operators like ASSIGN, PLUS, MINUS, and more.
- **Delimiters:** Tokens for delimiters such as COMMA, SEMICOLON, LEFTPAR, LEFTBRAC, COLON, and more.
- **Keywords:** Reserved words like FUNCTION, LET, TRUE, and others that have specific meanings in DE.

```
IDENT  = "IDENT"
INT    = "INT"
STRING = "STRING"

// Operators
ASSIGN      = "="
PLUS        = "+"
EXCLAMATION = "!"

// Delimiters
COMMA       = ","
SEMICOLON   = ";"
LEFTPAR     = "("
LEFTBRAC    = "{"
COLON       = ":"

// Keywords
FUNCTION = "FUNCTION"
LET      = "LET"
TRUE     = "TRUE"
```

Tokenization Process

In simple terms, the lexer examines the code token by token and organizes it in a way that the interpreter can make sense of. This process involves matching code components to predefined categories, allowing the interpreter to understand their significance. The snippet below illustrates how specific characters in the source code are recognized and mapped to tokens:

```
case '+':
    tok = newToken(token.PLUS, l.currentChar)

case '-':
    tok = newToken(token.MINUS, l.currentChar)

case '{':
    tok = newToken(token.LEFTBRAC, l.currentChar)

case '}':
    tok = newToken(token.RIGHTBRAC, l.currentChar)

case '(':
    tok = newToken(token.LEFTPAR, l.currentChar)

case ')':
    tok = newToken(token.RIGHTPAR, l.currentChar)

case ':':
    tok = newToken(token.COLON, l.currentChar)
```

This mapping process ensures that the code is parsed and categorized accurately, laying the foundation for the subsequent stages of interpretation and execution.

Tokenization is a crucial step in the DE compiler's journey from source code to executable instructions, and it forms the bridge between human-readable code and machine-understandable instructions.

Testing the Lexer

To ensure the correctness of the lexer and the accuracy of the tokenization process in DE, rigorous testing is performed using Go's testing framework. Let's take a look at an example of a test case and how it works.

Test Case: Lexing let Variables

```
func TestLexingLetVariables(t *testing.T) {
	input := `let x = 5;`
	tests := []struct {
		expectedType    token.TokenType
		expectedLiteral string
	}{
		{token.LET, "let"},
		{token.IDENT, "x"},
		{token.ASSIGN, "="},
		{token.INT, "5"},
		{token.SEMICOLON, ";"},
	}

	l := lexer.New(input)
	testLexer(t, l, tests)
}
```

In this test case, we:

- Define the input, which is the DE statement containing the let variable declaration.

- Create an array of expected token types and their corresponding literals. These expected tokens represent what we anticipate the lexer will produce when processing the input.

- Create an instance of the lexer l and feed it the input.

The **testLexer** Function
is Responsible for running the actual tests. It iterates through the expected tokens, compares them with the tokens produced by the lexer, and reports any discrepancies.

```
func testLexer(t *testing.T, l *lexer.Lexer, tests
[]struct {
	expectedType    token.TokenType
	expectedLiteral string
}) {
	for idx, val := range tests {
		tok := l.NextToken()
		if tok.Type != val.expectedType {
			t.Fatalf(
				"
				 Failed at index [%d] - tokenType wrong.
				 expected=%q but got=%q
				",
				idx,
				val.expectedType,
				tok.Type,
			)
		}
		if tok.Literal != val.expectedLiteral {
			t.Fatalf(
				"
				 Failed at index [%d] - literal
				 wrong. expected=%q but got=%q
				",
				idx,
				val.expectedLiteral,
				tok.Literal,
			)
		}

	}
}
```

The testLexer function performs the following steps:

- **Iterates** through the expected tokens and retrieves the next token from the lexer using `l.NextToken()`.

- **Compares** the token type and literal from the lexer with the expected values.

- If there is a mismatch, it reports an error specifying the index where the discrepancy occurred.

These tests serve as a crucial quality control step in the development of DE, ensuring that the lexer correctly identifies tokens and their properties as expected.

7.2 - Parsing and Abstract Syntax Tree (AST)

In the world of programming, you've likely encountered the term `"parser"` in phrases like `"parser error"` or `"parsing this input."` But what exactly is a parser, and how does it work? At its core, a parser is a software component that takes input data, often in the form of text, and creates a structured representation, like a parse tree or abstract syntax tree (AST). This structural representation helps validate the syntax and understand the code's organization.

But before diving into the world of parsers, how they work and what strategies we use, we have to take a look at the AST, which is a crucial part of many interpreters.

Abstract Syntax Trees (ASTs)

Abstract Syntax Tree (AST) is a crucial concept often utilized by interpreters and compilers. An AST serves as an essential bridge between human-readable code and a computer's understanding of that code.

Essentially, it's a hierarchical, tree-like data structure representing the syntactic structure of the source code.

Let's break this down further:

- **Hierarchical Structure:** An AST is organized as a hierarchy, akin to a family tree. At the top, there's a root node, and beneath it, there are branches, each representing different aspects of the code's structure.

- **Representation of Syntax:** The AST captures the structure of the code, including how different elements are related to one another. This encompasses everything from variables and functions to loops, conditions, and more.

- **Human-Readable to Computer-Friendly:** While humans write code in a way that makes sense to them, computers need a more structured and organised format to comprehend it. The AST acts as this intermediary by structuring the code's logic in a manner that a computer can process efficiently.

For instance, consider a simple mathematical expression like `3 * (5 + 2)`. To a human, this looks straightforward, but for a computer, it's vital to understand the order of operations—first, add 5 and 2, then multiply the result by 3. The AST breaks down this expression into nodes, highlighting these relationships and ensuring that the correct calculations are performed.

Key Components of an AST:

- **Nodes:** These are the fundamental building blocks of an AST. Each node represents a distinct element in the code, such as a variable, operator, function call, or control structure.

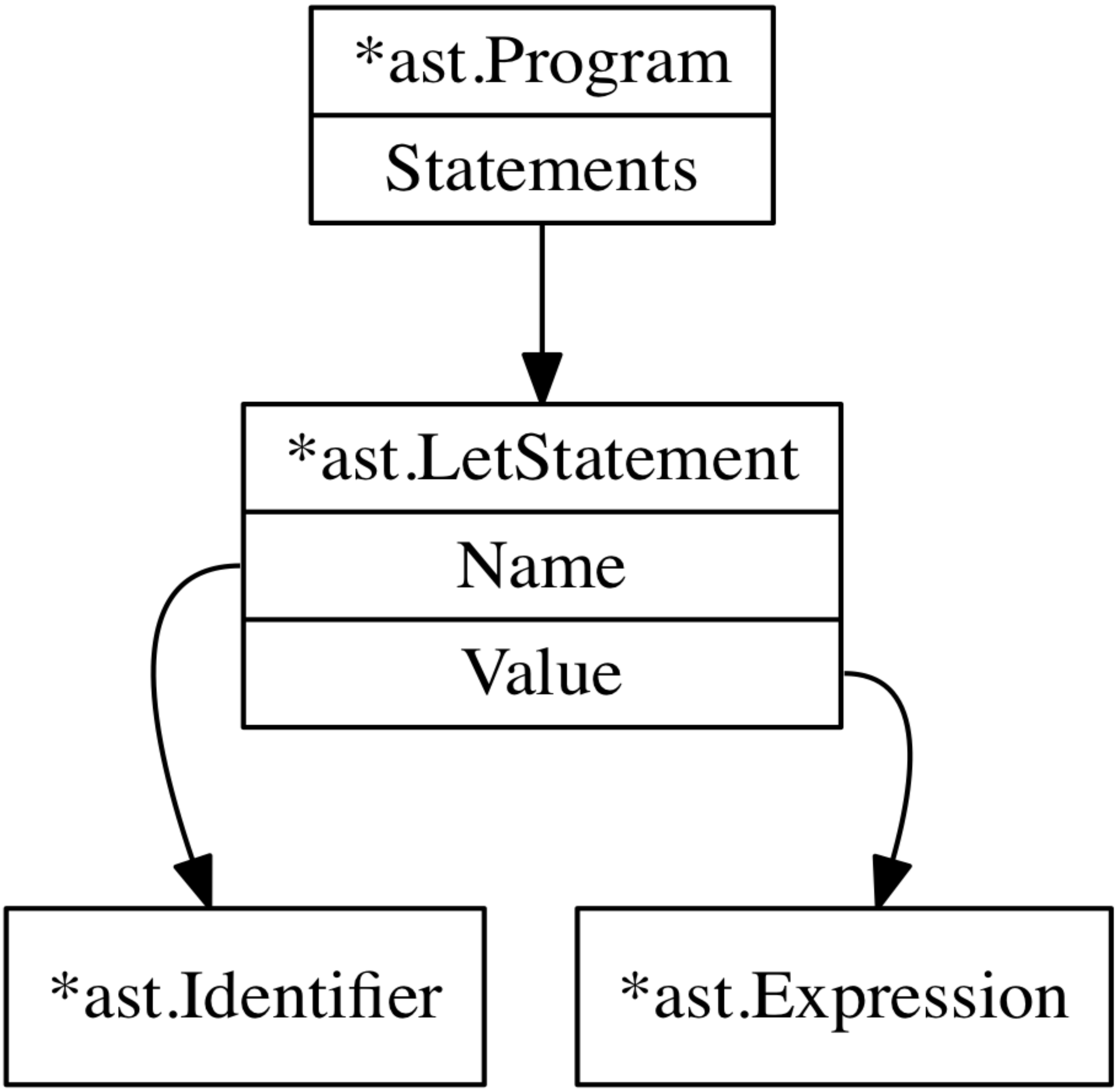
- **Edges:** These connections link nodes together, denoting how elements in the code are related and ordered. For example, an edge might indicate that a particular function is called within a loop.

- **Root Node:** At the very top of the hierarchy, the root node provides an entry point to the entire AST. It represents the entire code or script.

- **Leaves:** These are the nodes at the furthest ends of the tree, representing the most granular elements in the code. In our mathematical expression example, the numbers 3, 5, and 2 are leaves.

By constructing an AST from source code, interpreters can efficiently navigate, analyze, and execute the code. It ensures that the code adheres to the correct syntax and semantics of the programming language. Moreover, the AST simplifies tasks like variable scoping, error checking, and code optimisation.

Here's a visual representation of what the AST will resemble while parsing the code `let x = 5;`



Parsing Strategies

Parsing programming languages can follow two primary strategies: top-down parsing and bottom-up parsing. Our focus will be on a specific type of top-down parsing: recursive descent parsing. Let’s understand each of them in detail.

Top-Down Parsing

Top-down parsing is one of the two primary parsing strategies used to analyze and understand programming languages. In this approach, the parser starts at the root of the Abstract Syntax Tree (AST) and proceeds by examining the code structure from top to bottom, following the hierarchical organization of the language's grammar. This method is akin to reading and comprehending a document from the title or main heading and then exploring its subsections.

Recursive Descent Parsing

Recursive descent parsing is a specific type of top-down parsing we'll employ in our interpreter. It begins by constructing the root node of the AST and then systematically descends through the code structure. This method is recursive in nature, as it involves calling parsing functions recursively to navigate the code's various components.

Our parsing approach is based on Vaughan Pratt's innovative technique, the "top-down operator precedence" or Pratt parser. This parser has the ability to handle operator precedence and associativity with remarkable efficiency. Instead of using traditional grammar rules, it analyses operators individually and allows for a step-by-step descent through the code structure.

With that said, let’s now explain Context-Free Grammar (CFG) and Pratt Algorithm.

Context-Free Grammars (CFGs)

Context-Free Grammars are a fundamental concept in formal language theory and play a pivotal role in specifying the syntax of programming languages, as well as in parsing and interpreting code. CFGs provide a structured and systematic way to describe the syntax of a language, allowing us to define the set of valid sentences or expressions in that language.

Here's an overview of key aspects of CFGs

- Formal Definition: CFGs consist of a set of production rules that define how strings of symbols are generated. A CFG comprises the following components:

    - A set of terminals, which are the actual symbols appearing in the language (e.g., keywords, variables, operators).
    - A set of non-terminals, which are placeholders representing syntactic constructs (e.g., expressions, statements).
    - A start symbol, denoting the initial non-terminal from which the language's valid constructs are generated.
    - A set of production rules, specifying how non-terminals can be replaced with sequences of terminals and non-terminals.
- **Derivation:** Derivation in CFGs involves repeatedly applying production rules to transform a start symbol into a valid sentence or expression in the language. This process continues until no non-terminals remain.
- **Parse Trees:** A parse tree is a graphical representation of the derivation of a sentence from the start symbol using the production rules. Each node in the tree represents either a non-terminal or a terminal, and the tree's structure reflects the hierarchical organization of the sentence.
- **Ambiguity:** CFGs can exhibit ambiguity, where a sentence can have multiple valid parse trees or interpretations. Ambiguity can complicate parsing and lead to different behaviours for the same input.
- **Use in Language Specification:** Programming language designers use CFGs to formally specify the syntax of a language. The rules define how valid programs should be structured, making it essential for creating language grammars.

Pratt Parsing Algorithm

Pratt parsing, named after its creator Vaughan Pratt, is a parsing algorithm designed for handling expressions and operators with varying precedences. Pratt parsing stands out for its simplicity, efficiency, and flexibility, making it a popular choice for expression parsing in interpreters.

Here's an overview of key features of the **Pratt parsing** algorithm:

- **Tokenization:** The first step in Pratt parsing is tokenization, where the input expression is broken down into a sequence of tokens. Tokens include operators, operands (such as variables or literals), and parentheses. We already talked about it in the lexer section.

- **Precedence and Associativity:** In Pratt parsing, each operator is associated with a precedence level, which determines the order in which operators are applied. Operators with higher precedence are evaluated before those with lower precedence. Additionally, operators can have left-associative, right-associative, or non-associative properties, which dictate how they are grouped when they have the same precedence.

- **Parsing Function:** The core of the Pratt algorithm is the parsing function, which is defined for each operator. The parsing function handles the parsing and evaluation of expressions involving the operator. There is one parsing function for each unique operator, and the function's behaviour depends on the operator's precedence and associativity.

- **Recursive Descent:** Pratt parsing is a recursive descent parsing method, which means it recursively calls itself to parse sub-expressions. When the parsing function encounters an operator with higher precedence, it recursively calls itself to parse the right-hand side of the expression.

- **Expression Trees:** Pratt parsing constructs an expression tree as it processes the input expression. The tree's nodes represent operators, and the tree's structure reflects the order of evaluation based on precedence and associativity. The final tree is a hierarchical representation of the parsed expression.

- **No Ambiguity:** Pratt parsing excels at handling expressions without ambiguity. Unlike traditional context-free grammars that may require more complex techniques to resolve ambiguity, Pratt parsing's precedence and associativity definitions explicitly dictate the order of operations, leaving no room for ambiguity.

- **Error Handling:** Pratt parsing can easily handle error detection and recovery. When an invalid expression is encountered, the algorithm can gracefully handle the error and provide meaningful error messages.

- **Extensible:** Pratt parsing is highly extensible. New operators and their parsing functions can be added without major changes to the existing code, making it suitable for languages with a wide range of operators.

You might be asking yourself a question now, why anyone would favour the Pratt algorithm over context-free grammar? Well, before explaining why I choose to go with Pratt, let's compare them together first.

Pratt vs. CFGs

Operator Precedence and Associativity Control

- **Pratt Algorithm:** Pratt parsing is specifically designed for handling expressions with varying operator precedences and associativities. You can precisely define how each operator should be parsed and how they interact with each other in terms of precedence and associativity. This level of control simplifies expression parsing, especially in languages with many operators of different priorities.

  Context-Free Grammars: CFGs are less intuitive for expressing operator precedence and associativity. While it's possible to define precedence in CFGs, it often involves more complex rules and can lead to parsing ambiguities.

Ambiguity Resolution

- **Pratt Algorithm:** Pratt parsing inherently resolves ambiguity when it comes to operator precedence. Expressions are parsed in a way that adheres to the operator's precedence level, preventing ambiguous interpretations. This makes error detection and reporting more straightforward.

  Context-Free Grammars: CFGs can introduce ambiguity when operators have the same precedence. Resolving this ambiguity requires additional rules or techniques like operator precedence parsers, which may complicate the grammar and the parsing process.

Ease of Extension

- **Pratt Algorithm:** Pratt parsing is highly extensible. Adding new operators is relatively simple because you only need to define their precedence and associativity and implement their parsing functions. This makes it an attractive choice for languages with evolving sets of operators.

  Context-Free Grammars: Extending CFGs to accommodate new operators can be more challenging. New operators may require modifications to the grammar rules, which can be cumbersome and error-prone.

Error Handling

- **Pratt Algorithm:** Pratt parsing provides clear error messages when parsing issues arise due to its predictable nature. Error handling is built into the algorithm.

  Context-Free Grammars: Handling errors in CFG-based parsers can be more challenging, and error messages may not be as informative, especially when resolving parsing ambiguities.

Perhaps you asking yourself now why I chose Pratt over CFG? Well, I chose Pratt over CFG mainly for simplicity. But you might wonder why not use the available tools for CFG, which is a valid point. The thing is, this is also a graduate project, and I wanted to deeply understand how parsers work. I aimed to build a parser from scratch, not rely on existing ones. While using existing parsers is great for production-ready languages to minimize errors and ensure robustness, I wanted to learn by creating a new parser using the Pratt algorithm, which offers the most benefits for my specific goals.

With that said, do you recall the example we mentioned earlier: `10 + 2 * 2 + 10` ? It might have taken a few chapters, but now that we're here, it's time to dive into the (TDOP).

7.3 - Pratt parsing (TDOP) Algorithm

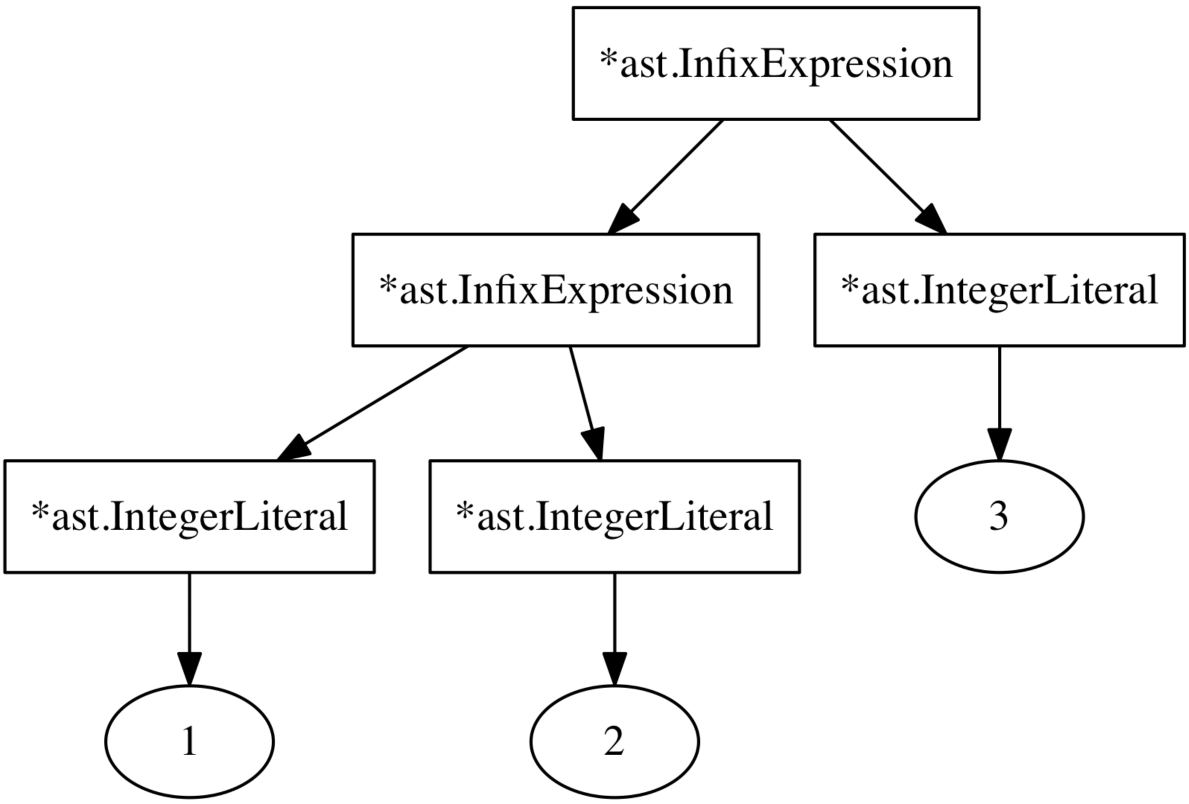
Finally, we've arrived at the heart of our discussion: understanding the `"TDOP"` in a straightforward manner. Let's simplify things with an example. If you grasp this example, you'll be well-equipped to comprehend the formula we explored earlier. Consider this expression: `1 + 2 + 3`

The big challenge here is not to represent every operator and operand in the resulting AST, but to nest the nodes of the AST correctly. What we want is an AST that (serialized as a string) looks like this: `((1 + 2) + 3)`

Now, in our tree, we're aiming to have two special `*ast.InfixExpression` nodes. Think of these nodes as containers for operations like addition. The higher one in the tree, which is like the parent, holds the integer literal 3 as its "Right" child, and its "Left" child is another `*ast.InfixExpression` node. This second node then holds the integer literals 1 and 2 as its "Left" and "Right" children respectively.

In this manner, the AST is structured hierarchically to represent the expression's intended order of operations. Each node carries specific operator and operand information, creating a tree-like structure that mirrors the way we naturally perceive mathematical expressions. This is at the core of how Pratt parsing works, allowing us to accurately construct an AST for various expressions, no matter how complex they may be.

To understand how the previous example becomes the AST, take a look at the image below.



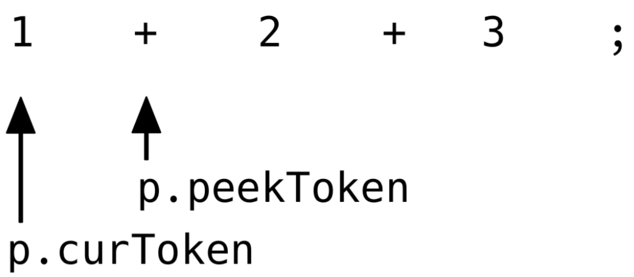
What's exciting is that our parser actually achieves this! When we feed the code "1 + 2 + 3;" into the parser, it understands the structure we want, and it carefully assembles the AST to reflect this. Now, let's explore how it does this step by step.

TDOP Steps

As the parser springs into action when we input `"1 + 2 + 3;"`, it follows a set of steps:

1. The parser starts by examining `"1"`, our first piece. It's a numeric value, and the parser knows how to handle it as an `*ast.IntegerLiteral`

2. The parser looks ahead and realizes there's more to the expression. It identifies the first `"+"` as an operator. This is done by the `curToken` and `peekToken` they keep track of what we examine now and what next.

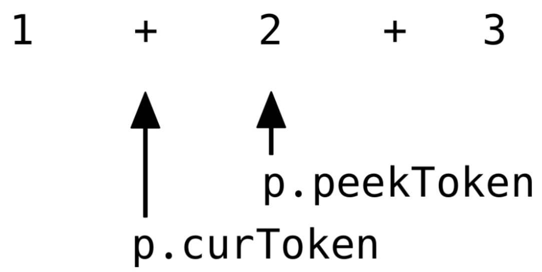


3. The parser enters a loop that's essential for correct nesting. It checks the precedence of the current operator `"+"` against the next one. If the next operator has higher precedence, the loop works to nest expressions correctly.

4. Inside the loop, the parser identifies the `"+"` operator and uses its associated parsing function `parseInfixExpression`. This function understands how to handle infix expressions – operations that involve two values and an operator in between.

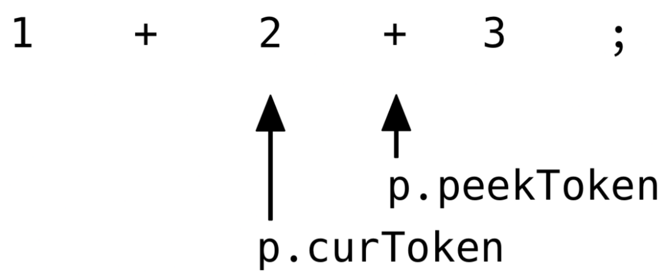
5. This function, named `parseInfixExpression`, assembles an `*ast.InfixExpression` node. This node stores the `"+"` operator, the left side (which is our `"1"`), and it expects a right side, which it fetches by calling `parseExpression` again.

6. The parser re-enters the `parseExpression` function, this time for the right side of the `"+"` operator. It identifies `"2"` and creates an `*ast.IntegerLiteral` for it.

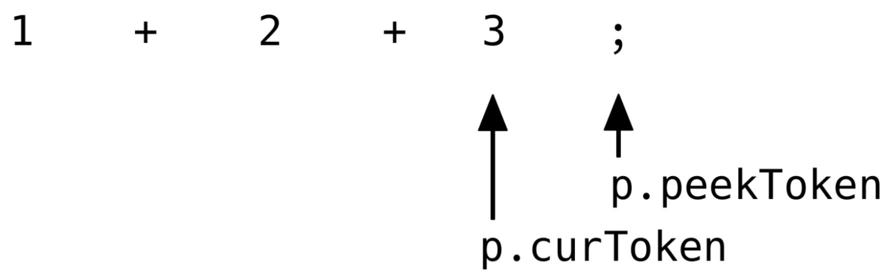


7. With the loop done, the parser assembles the complete `*ast.InfixExpression`, connecting the left and right sides. The result is nested correctly: `(1 + 2)`.

8. Now, the parser re-enters the loop to handle the final `"+"` and `"3"` The process repeats, resulting in another `*ast.InfixExpression`: `(1 + 2) + 3`



9. The parser understands that the expression has been fully processed, and the loop ends.



And that's it! The parser has created the exact tree structure we were aiming for: `((1 + 2) + 3)`. This structured representation is crucial for the interpreter to understand and evaluate the code's intent.

Here is how `parseExpression` looks in DE implemented in Go

```
func parseExpression(precedence int) ast.Expression {
	// Handle Prefix Expression
	leftExp := prefixParseFunc()

	for precedence < peekPrecedence {
		infixParseFunc := p.infixParseFuns[p.peekToken.Type]
		if infixParseFunc == nil {
			return leftExp
		}

		p.nextToken()

		// Handle Prefix Expression
		leftExp = infixParseFunc(leftExp)
	}

	return leftExp
}
```

I don't want to delve extensively into the Golang implementation details. However, if you're interested in exploring the actual implementation, you can explore the complete DE implementation on its GitHub repository

https://github.com/Mostafa-DE/delang

For the sake of simplicity, I've skipped numerous detailed aspects of the implementation. I wanted to avoid overwhelming you with specifics, such as how the (TDOP) is implemented in Go.

Now let’s take a look at the precedence in DE and how they managed. Operator precedence is managed using a predefined set of constants and a mapping structure. These constants help define the order in which operators are evaluated. The higher the constant's value, the higher the operator's precedence. This setup ensures that expressions are evaluated correctly.

```go
const (
	_                int = iota
	LOWEST           // Lowest precedence
	AND_OR           // and or
	EQUAL            // ==
	LESS_GREATER     // > <
	SUM_SUB          // + -
	MUL_DIV_MOD      // * / %
	PREFIX           // -X !X
	CALL             // myFunction(X)
	INDEX            // array[index]
)

var precedences = map[token.TokenType]int{
	token.EQUAL:          EQUAL,
	token.NOTEQUAL:       EQUAL,
	token.LESSTHAN:       LESS_GREATER,
	token.GREATERTHAN:    LESS_GREATER,
	token.LESSTHANEQ:     LESS_GREATER,
	token.GREATERTHANEQ: LESS_GREATER,
	token.AND:            AND_OR,
	token.OR:             AND_OR,
	token.PLUS:           SUM_SUB,
	token.MINUS:          SUM_SUB,
	token.SLASH:          MUL_DIV_MOD,
	token.ASTERISK:       MUL_DIV_MOD,
	token.MOD:            MUL_DIV_MOD,
	token.LEFTPAR:        CALL,
	// array indexing has the highest precedence
	token.LEFTSQPRAC:     INDEX,
}
```

- **LOWEST**: The lowest precedence for operators.
- **AND_OR**: Represents logical AND (&&) and logical OR (||) operators.
- **EQUAL**: Stands for equality operators (== and !=).
- **LESS_GREATER**: Covers comparison operators (<, >, <=, and >=).
- **SUM_SUB**: Represents addition (+) and subtraction (-) operators.
- **MUL_DIV_MOD**: Includes multiplication (*), division (/), and modulus (%) operators.
- **PREFIX**: Denotes prefix operators like unary minus (-X) and logical NOT (!X).
- **CALL**: Represents function calls (myFunction(X)).
- **INDEX**: This is the highest precedence level and pertains to array indexing (array[index]).

Now before moving to something else, I would like to show you an actual parsing that I did for parsing the for loop.

The idea was to give the developer the ability to use one of the loop forms shown below

```
for num in []: {...}
for index, num in []: {...}
for _, num in []: {...}
```

And here is the algorithm to parse the for-loop.

```
Steps to parse for loop statement:
    - current token is "for"

    - Step 1:
        -Check if the current token is not an underscore
            - Check if the next token is a COMMA
                - Yes,
                    - Assign the current token to the index identifier
                    - Skip the COMMA
                    - Check if the current token is an underscore
                        - Yes,
                            - then throw an error

                        - No,
                            - Assign the current token to the variable identifier

                - No,
                    - Assign the current token to the variable identifier

        - Else, check if the current token is an underscore
            - Yes, Check if the next token is a comma
                - No, then throw an error

                - Yes,
                    - Skip the COMMA
                    - Check if the current token is an underscore
                        - Yes, then throw an error
                        * You cannot skip both the index and the variable identifier
                        * Use "during" loop instead

                    - Check if the current token is an IN token
                        - Yes, then throw an error

                    - Assign the current token to the variable identifier

        - Else, throw an error
        * variable identifier is required after "for" keyword


    - Step 2:
        - Check if the current token is an IN token
            - No, then throw an error

            - Yes, skip the IN token
            - Parse the iterable expression

    - Step 3:
        - Check if the current token is a COLON token
            - No, then throw an error

            - Yes, skip the COLON token

    - Step 4:
        - Check if the current token is a LEFTBRAC token
            - No, then throw an error

            - Yes, skip the LEFTBRAC token
            - Parse the block statement
}
```

Certainly, this is just one instance, and I've excluded numerous similar parsing scenarios. If you're interested in exploring more parsing decisions for other statements in DE, feel free to check out the DE repository on GitHub.

https://github.com/Mostafa-DE/delang/blob/master/parser/parser.functions.go

7.4 - The Evaluator

In this section, we're going to delve into the next phase: the evaluator. It's like the brain of our interpreter, where we make sense of the code we've parsed. Imagine it as the part of the program that reads the AST (remember, the structured representation of the code) and figures out what it means.

Tree-Walking Interpreter

A `"tree-walking interpreter"` This sounds complex, but it's essentially like having someone walk through the branches of a tree, stopping at each node and figuring out what to do next. In our case, the `"tree"` is the AST, and the `"walking"` is the process of going through it step by step.

Our goal is to interpret the code `"on the fly"` which means we won't be turning it into some other kind of code before running it. This is how early programming languages used to work. We'll take the AST and directly figure out what it should do without any extra steps.

What's an Evaluator?

An `"evaluator"` is just a function that takes something (in our case, an AST node) and figures out what it means. Imagine you have a set of instructions, and someone follows those instructions step by step to get a result. That's what the evaluator does. It's like reading a recipe and cooking a meal.

Breaking Down the Pseudocode

Now, we're not going to dive into super-detailed code. Let's keep things simple. Here's a rough idea of how the evaluator works:

```
fun eval(astNode) {
    if (astNode is integerLiteral) {
        return astNode.integerValue

    } else if (astNode is booleanLiteral) {
        return astNode.booleanValue

    } else if (astNode is infixExpression) {
        leftEvaluated = eval(astNode.Left)
        rightEvaluated = eval(astNode.Right)

        if astNode.Operator == "+" {
            return leftEvaluated + rightEvaluated

        } else if ast.Operator == "-" {
            return leftEvaluated - rightEvaluated
        }
    }
}
```

Making Sense of the Pseudocode

Let's break this down. Imagine you're reading a recipe. If it says "add 2 cups of sugar" you know what to do. In our code, if we encounter an integer (a whole number), we return that number. If we see a boolean (true or false), we return that value.

But here's the cool part: when we find an infix expression like `"a + b"`, we dive deeper. We evaluate what's on the left (getting the value of `"a"`) and what's on the right (getting the value of `"b"`). Then we follow the recipe (operator) to mix them together, just like adding ingredients in a recipe.

Let’s see how we do this in Go, making a decision based on the AST.

```go
func Eval(node ast.Node, env *object.Environment) {
    switch node := node.(type) {
        case *ast.Program:
            // Root node of every AST our parser produces
            return evalProgram(node.Statements, env)

        case *ast.ExpressionStatement:
            return Eval(node.Expression, env)

        case *ast.Integer:
            return &object.Integer{Value: node.Value}

        case *ast.IfExpression:
            return evalIfExpression(node, env)

        case *ast.Identifier:
            return evalIdentifier(node, env)
}
```

As you can see we basically evaluate each node we encounter in the Go language and return the result immediately.

Each statement in DE has its own way of evaluating depending on the expected behaviour, let's see what we evaluate in DE

Expressions Evaluation

- **Literal Expressions:**
    - Integer Literals: Directly represent their integer values.
    - Boolean Literals: Represent true or false values.

- **Identifier Expressions:**
    - Retrieve the value bound to the identifier from the current scope.

- **Prefix Expressions:**

- Evaluate the right-hand side expression and apply the specified unary operator (e.g., negation, logical NOT) to the result.
- **Infix Expressions:**
  - Evaluate the left and right expressions and apply the specified binary operator (e.g., addition, multiplication) to the results.

- **If Expressions:**
  - Evaluate the condition expression.
  - If the condition is true, evaluate the consequence expression; otherwise, evaluate the alternative expression.

- **Function Call Expressions:**
  - Evaluate the function expression to get the function object.
  - Evaluate the arguments.
  - Call the function with the evaluated arguments.

Statements Evaluation

- **Let Statements:**
  - Evaluate the right-hand side expression.
  - Bind the result to the identifier in the current scope.

- **Return Statements:**
  - Evaluate the return value expression.
  - Exit the current function, returning the result.

- **Expression Statements:**
  - Evaluate the expression and discard the result (useful for side effects).

- **Block Statements:**
  - Create a new scope.
  - Evaluate each statement in the block within the new scope.

- **For Loop Statements:**
  - Evaluate the iterable expression.

- For each item, bind it to the specified identifiers in the loop header and evaluate the loop's body.

- **during Loop Statements:**
  - Evaluate the condition expression.
  - While the condition is true, evaluate the loop's body.

Scope Management

- **Scope Creation:**
  - Functions and blocks create new scopes.

- **Variable Binding and Lookup:**
  - Let statements bind values to identifiers in the current scope.
  - Identifiers are looked up in the current scope and, if not found, in outer scopes.

Error Handling

- **Runtime Errors:**
  - Detect and handle runtime errors, such as division by zero or using undefined variables.

I won't dive too deep into the details of how each piece of code is handled in DE. Instead, let's explore how a for loop is processed in DE, similar to what we covered in the parsing section.

The reason for this, explaining everything in full detail would need its own separate document, especially when dealing with unique situations and specific details for each case. If you're curious about more examples and implementations, check out the DE repository, particularly the evaluator package, where you can find all the DE implementations.

```
https://github.com/Mostafa-DE/delang/tree/master/evaluator
```

Evaluate for loop

In DE, the for loop is a crucial element for iterating over arrays or strings. The evalForStatement function manages the loop's evaluation, setting up a local environment to handle variable declarations and maintain identifier uniqueness.

The loop's body is then assessed, and an expression after the for statement is expected. The evaluation process differentiates between array and string types. For arrays, the arrayLoop function efficiently traverses each element, while stringLoop handles individual characters in a string. Both functions handle index and variable identifiers with precision, executing the block statement within the loop.

I thought about how to explain things in detail, and I realized the best way is to show you the actual code in Go with comments on important parts. You don't have to be familiar with Go; just read the comments, and you'll understand the main points.

Alternatively, feel free to skip the code and jump directly to the explanations below. This will provide you with a clear understanding of how things operate.

```go
func evalForStatement(
  node *ast.ForStatement,
  env *object.Environment

) object.Object {

    // Create a new local environment for the for loop
    localEnv := object.NewLocalEnvironment(env)

    // Handle the index identifier
    var idxIdent string
    if node.IdxIdent != nil {
        idxIdent = node.IdxIdent.Value
    }

    // Handle the variable identifier
    var varIdent string
    if node.VarIdent == nil {
        return throwError(
          "Expected a variable identifier after the for statement"
        )
    } else {
        varIdent = node.VarIdent.Value
    }

    // Ensure that the index identifier and variable identifier are not the same
    if idxIdent == varIdent {
        return throwError(
          "Index identifier and variable identifier cannot be the same"
        )
    }

    // Get the body of the for loop
    body := node.Body

    // Check for the presence of an expression after the for statement
    if node.Expression == nil {
        return throwError(
          "Expected an expression after the for statement"
        )
    }
```

At this point, we are done with the varIdent and idxIdent and we will start on the iterable collection (Array or String). See the next image.

```go
    // Evaluate the expression
    eval := Eval(node.Expression, localEnv)

    if isError(eval) {
        return eval
    }

    // Check the type of the evaluated expression
    switch eval.Type() {
    case object.STRING_OBJ:
        // If it's a string, perform the string loop
        res := stringLoop(
          eval.(*object.String),
          idxIdent,
          varIdent,
          body,
          localEnv
        )

        if isError(res) {
            return res
        }

    case object.ARRAY_OBJ:
        // If it's an array, perform the array loop
        res := arrayLoop(
          eval.(*object.Array),
          idxIdent,
          varIdent,
          body,
          localEnv
        )

        if isError(res) {
            return res
        }

    default:
        // If it's not iterable, return an error
        return throwError(
          "Type %s is not iterable", eval.Type()
        )
    }

    return NULL
}
```

You can see here the only two things iterable in DE are Arrays and Strings and we handle each one of them separately in our implementation. Let's see how we deal when iterating over arrays/strings.

```go
// Function to handle array iteration in a for loop
func arrayLoop(
  array *object.Array,
  idxIdent string,
  varIdent string,
  body *ast.BlockStatement,
  env *object.Environment

) object.Object {

    for idx, val := range array.Elements {
        // Set the index identifier in the environment
        if idxIdent != "" {
            env.Set(idxIdent, &object.Integer{Value: int64(idx)}, false)
        }

        // Set the variable identifier in the environment
        env.Set(varIdent, val, false)

        // Evaluate the block statement in the loop body
        result := evalBlockStatement(body.Statements, env)

        if isError(result) {
            return result
        }

        // Check for break or continue statements
        if result != nil {
            if result.Type() == object.BREAK_OBJ {
                break
            }

            if result.Type() == object.SKIP_OBJ {
                continue
            }
        }
    }

    return NULL
}
```

```go
// Function to handle string iteration in a for loop
func stringLoop(
  str *object.String,
  idxIdent string,
  varIdent string,
  body *ast.BlockStatement,
  env *object.Environment

) object.Object {

    for idx, val := range str.Value {
        // Set the index identifier in the environment
        env.Set(idxIdent, &object.Integer{Value: int64(idx)}, false)

        // Set the variable identifier in the environment
        env.Set(varIdent, &object.String{Value: string(val)}, false)

        // Evaluate the block statement in the loop body
        result := evalBlockStatement(body.Statements, env)

        if isError(result) {
            return result
        }

        // Check for break or continue statements
        if result != nil {
            if result.Type() == object.BREAK_OBJ {
                break
            }

            if result.Type() == object.SKIP_OBJ {
                continue
            }
        }
    }

    return NULL
}
```

And that's it, I know that's a lot and you may not get the code 100%, but that's ok let's explain what the above image is all about.

evalForStatement Function

- This function is the entry point for evaluating a for loop.
- It creates a new local environment specific to the for loop, providing a scoped context for variable declarations.
- Handles index and variable identifiers, ensuring they are distinct.
- Retrieves the body of the for loop for further evaluation.
- Checks for the presence of an expression after the for statement, throwing an error if not found.
- Evaluates the expression, determining its type and proceeding accordingly (either array or string).
- Returns a NULL object after the loop completes.

arrayLoop & stringLoop Functions:

- Responsible for iterating through elements of an (array/string) in the for loop.
- Uses a for-range loop to traverse each element in the (array/string).
- Sets the index and variable identifiers in the local environment.
- Evaluates the block statement in the loop body.
- Checks for break or skip statements, acting accordingly.
- Returns a NULL object after completing the array iteration.

Error Handling:

- Throughout the evaluation process, error checks are in place to handle unexpected situations, providing meaningful error messages.

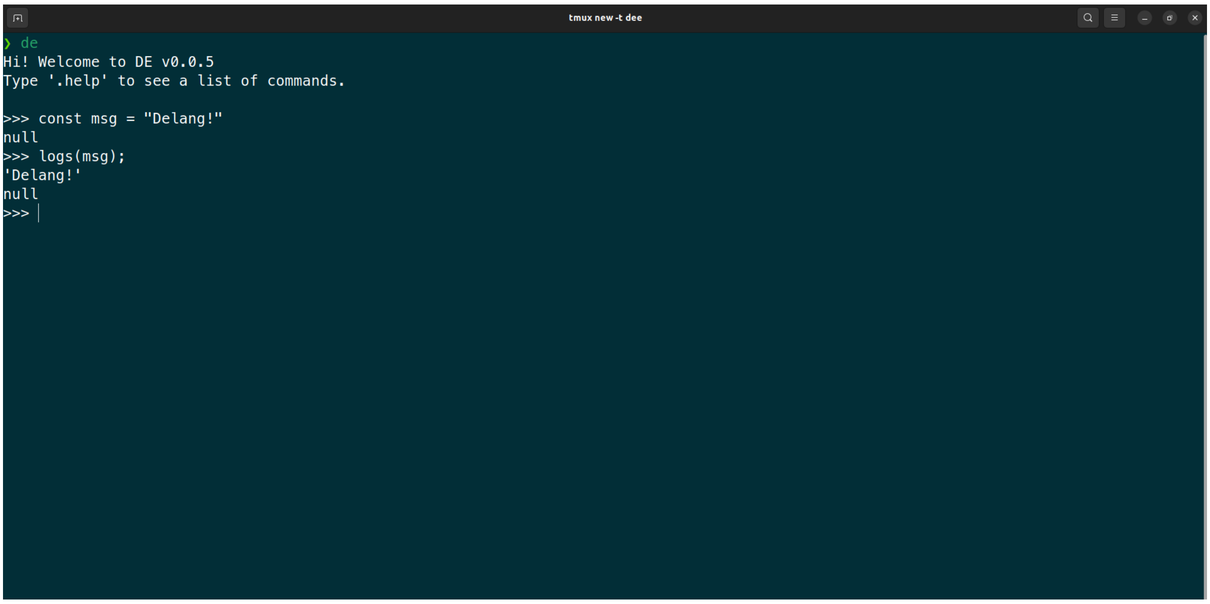
CHAPTER 8

The Interactive REPL Experience

In this chapter, we'll delve into the implementation of the Read-Eval-Print Loop (REPL) for our DE programming language. Our basic REPL was created also using Go language.

8.1 - Understanding the REPL

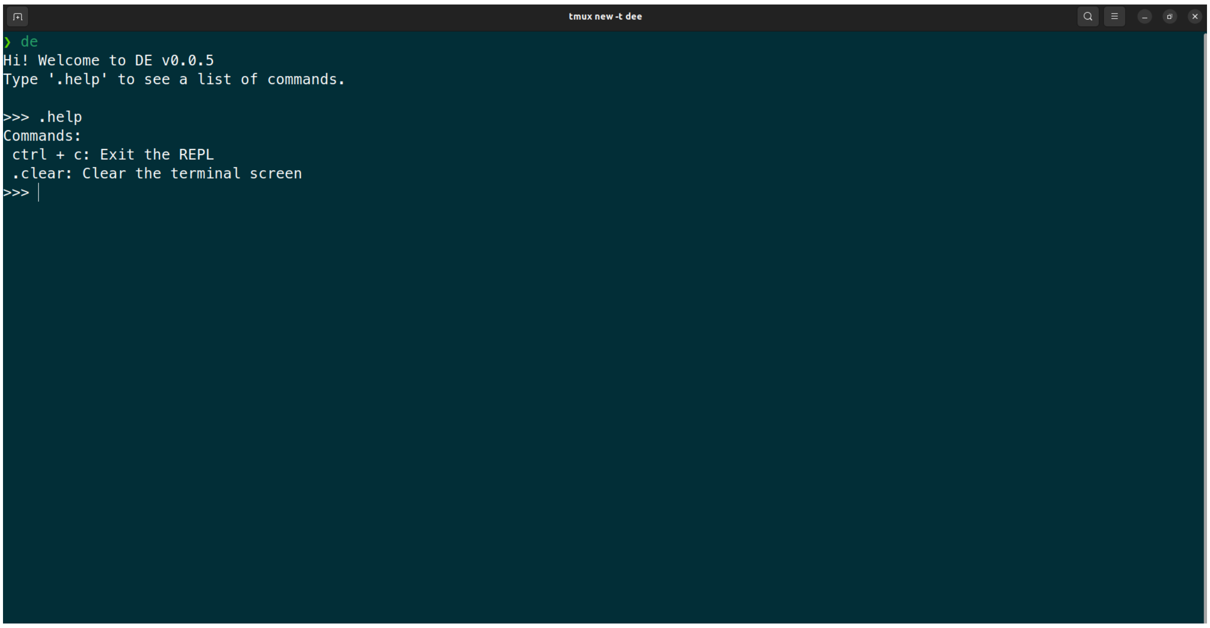
A REPL, which stands for Read-Eval-Print Loop, is an essential tool for many programming languages. It provides an interactive environment where users can enter code, the code is evaluated, and the result is displayed back to the user. This iterative process allows for quick experimentation, testing, and exploration of language features.



8.2 - Setting Up the REPL

Our REPL implementation is encapsulated within the repl package. The core function responsible for managing the REPL session is `StartSession()`. Let's break down the main components of our REPL implementation:

1. Initialization and Welcome Message: The REPL starts by initializing a few variables, including the input history, cursor position, and the current user input. I use the keyboard library to handle `"keyboard"` input. The user is greeted with a welcome message and is informed about a special command `".help"` that provides a list of available commands.



2. Main Loop: The REPL's main loop continually reads keyboard input. It handles various key presses and characters, responding to actions like navigation, deletion, adding spaces, and screen clearing. Depending on the key pressed, the loop either modifies the current input or triggers specific actions.

3. User Input Processing: When the user presses the Enter key, the current input is processed. If the input is a special command like `".clear"`, the screen is cleared. If it's `".help"`, a list of commands is displayed. Otherwise, the input is parsed and evaluated using the DE interpreter. The result of the evaluation is printed on the console.

8.3 - Interactive Experience

The interactive nature of a REPL makes it an invaluable tool for developers. The user can navigate through input history using the arrow up key, make corrections using backspace, and even move the cursor to different positions within the input line.

Let's take a look at the actual Go code that implements our REPL terminal. I've organized the code into functions that manage different aspects of the REPL, such as user input processing and evaluation. This separation of concerns ensures that our REPL remains modular and maintainable.

Here's a snippet of the Go code that represents the foundation of our REPL implementation:

```go
package repl // import statements and init

func StartSession() { // REPL init and welcome msg
    history := []string{}
    historyIndex := 0
    currentInput := ""
    cursorPosition := 0
    fmt.Print(PROMPT)

    for {
        // Reading keyboard input, handling actions
    }
}

func startExec(command string, env *object.Environment) {
    /*
        Parsing user input,
        handling special commands,
        And evaluation
    */
}
```

The choice was taken to handle the keystrokes and keyboard event is to use a keyboard package in Go called `"github.com/eiannone/keyboard"` It's a good package that gives you the ability to interact easily with keyboard in simple and straightforward way.

Let's explore how we used the package to implement our REPL. I will not put the actual implementation as I did in the evaluation chapter, but I will provide an overview code and explain what I did to handle the keystrokes.

```
func StartSession() {
    for {
        char, key, err := keyboard.GetKey()

        if key == keyboard.KeyCtrlC {
            fmt.Println("\nBye!")
            break

        } else if key == keyboard.KeyEnter {
                startExec(currentInput, env)
            }
        }

        if key == keyboard.KeyArrowUp {...}

        if key == keyboard.KeyArrowDown {...}

        if (
            key == keyboard.KeyBackspace ||
            key == keyboard.KeyBackspace2
        ) {...}

        if key == keyboard.KeyArrowLeft {...}

        if key == keyboard.KeyArrowRight {...}

        if key == keyboard.KeyCtrlL {...}

        if char != 0 {...}

        if key == keyboard.KeySpace {...}
    }
}
```

The StartSession function initiates a user session in DE, providing an interactive command-line interface. Here's a simplified breakdown of its functionality:

- **Environment Setup:** It establishes an initial environment for DE, initializes the prompt, and displays a welcome message.

- **Input Handling:** It enters a loop to continuously read user input character by character. The loop supports various functionalities:

  - **Ctrl+C Exit:** Detects if the user presses Ctrl+C to gracefully exit the session.

  - **Enter Key:** Processes the entered command when the user presses Enter, executing it through the startExec function.

  - **Arrow Keys (Up and Down):** Allows navigation through command history. The up arrow retrieves the previous command, and the down arrow retrieves the next one.

  - **Backspace:** Removes characters when the user presses Backspace.

  - **Arrow Keys (Left and Right):** Enables cursor movement left and right within the input.

  - **Ctrl+L:** Clears the screen.

  - **Space Key:** Inserts a space in the input.

To delve deeper into the implementation of the DE REPL, you can explore the source code via

```
https://github.com/Mostafa-DE/delang/tree/master/src/repl
```