Personal Research

Paper Title:

Regular Expression for Pattern Recognition Computation in Golang.

WRITTEN BY:-

Mr. Jaywant Sandeep Adhau

(Undergraduate B.Tech Computer Science)

ABSTRACT

Regular expressions are a powerful tool for pattern recognition, text manipulation, and data validation in various programming languages. This paper explores the implementation and usage of regular expressions in Golang (Go), a statically typed, compiled language developed by Google. We delve into the nuances of both literal characters and metacharacters, explaining their formation, usage, and mathematical representation. Through practical examples and detailed documentation, we demonstrate the efficiency and robustness of Go in handling complex regular expression patterns for various applications, ranging from simple string matching to complex text processing tasks.

Golang, known for its simplicity, concurrency support, and high performance, is an ideal candidate for My research. The language's clean syntax and powerful standard library, particularly the regexp package, make it straightforward to implement and execute regular expressions efficiently. My research provides a comprehensive overview of how to leverage Go's capabilities for pattern recognition tasks, highlighting its advantages over other programming languages.

We start by providing a foundational understanding of literal characters in regular expressions, which are the simplest form of regex patterns. These characters represent themselves and can include alphabets, digits, and special symbols. We present a mathematical framework to describe literal characters, followed by a Go implementation to test various literal character patterns.

Next, we explore metacharacters, the backbone of regular expressions that enable advanced pattern matching. Metacharacters include symbols such as . (dot), * (asterisk), + (plus), ? (question mark), ^ (caret), $ (dollar sign), and more. Each metacharacter has a specific function, allowing for the construction of highly flexible and dynamic patterns. We provide a mathematical representation of each metacharacter and a corresponding Go program to illustrate their usage in real-world scenarios.

In addition to individual categories, we investigate the combination of literal characters and metacharacters to create more sophisticated regex patterns. These combined patterns are essential for tasks like email validation, URL parsing, and complex search operations. We provide examples and detailed explanations to showcase how these combinations can be effectively used in Go.

My research emphasises the importance of documentation and industry-grade commenting for code clarity and maintainability. Each function and code snippet is thoroughly documented to ensure that readers can easily understand and reproduce the examples. This aspect is crucial for educational purposes and for developers looking to implement regular expressions in their own Go projects.

We conclude by discussing the implications of My findings and potential areas for future research. This includes exploring advanced regex techniques, optimization strategies for performance improvement, and real-world applications in various domains such as data science, web development, and cybersecurity.

Through this paper, we aim to provide a valuable resource for developers and researchers interested in utilising regular expressions in Golang. By combining theoretical insights with practical examples, we hope to enhance the understanding and application of regex patterns, ultimately contributing to the broader field of pattern recognition and text processing.

Introduction:-

What is Golang?

Golang, also known as Go, is an open-source programming language developed by Google. Designed for simplicity, efficiency, and reliability, Go is known for its powerful concurrency mechanisms, garbage collection, and static typing. It is widely used for developing scalable and high-performance applications, making it an excellent choice for My research on regular expressions.

Why Choose Golang for Regular Expressions?

Performance: Go is compiled to machine code, offering performance comparable to languages like C and C++. This ensures that regular expression operations are executed swiftly, which is crucial for applications requiring real-time pattern recognition.

2.Simplicity and Readability: Go's syntax is clean and straightforward, making it easier to write, read, and maintain regular expression code.

3.Concurrency: Go's native support for concurrency allows regular expression operations to be performed concurrently, enhancing the efficiency of pattern recognition tasks in multi-threaded applications.
4.Standard Library: Go's standard library includes the regexp package, which provides a rich set of functions for compiling and executing regular expressions. This built-in support simplifies the development process and ensures consistency across different projects.

Regular Expressions in Golang

Regular expressions (regex) are sequences of characters that define search patterns. They are used for string matching, searching, and manipulation tasks. In Go, the regexp package provides the necessary functions to work with regular expressions.

1. Understanding Basic Concepts

- **Literal Characters**: These are the simplest type of regex, matching the exact characters.
    - Example: abc matches "abc".
- **Metacharacters**: Special characters with specific meanings in regex.
    - Common metacharacters: . ^ $ * + ? [] {} \ | ().

2. Basic Regex Patterns

- **Anchors**:
    - ^: Start of the string.
    - $: End of the string.
    - Example: ^abc$ matches "abc" but not "abcde" or "deabc".
- **Character Classes**:
    - [abc]: Matches any one of the characters inside the brackets.
    - [^abc]: Matches any character not inside the brackets.
    - [a-z]: Matches any character from a to z.
    - \d: Matches any digit, equivalent to [0-9].
    - \w: Matches any word character, equivalent to [a-zA-Z0-9_].
    - \s: Matches any whitespace character.
    - Example: [A-Za-z] matches any uppercase or lowercase letter.
- **Quantifiers**:
    - *: Matches 0 or more of the preceding element.

    - +: Matches 1 or more of the preceding element.
    - ?: Matches 0 or 1 of the preceding element.
    - {n}: Matches exactly n occurrences of the preceding element.
    - {n,}: Matches n or more occurrences of the preceding element.
    - {n,m}: Matches between n and m occurrences of the preceding element.
    - Example: a{2,4} matches "aa", "aaa", and "aaaa".
- **Groups and Alternations**:
    - (): Groups expressions.
    - |: Alternation (logical OR).
    - Example: a(b|c) matches "ab" and "ac".

Go provides a regexp package for working with regular expressions. Here are some common functions:

```
package main

import (
	"fmt"
	"regexp"
)

func main() {
	// Compile a regular expression
	re := regexp.MustCompile(`\w+@\w+\.\w+`)

	// MatchString: Test if a string matches the pattern
	fmt.Println(re.MatchString("test@example.com")) // true

	// FindString: Find the first match in the string
	fmt.Println(re.FindString("test@example.com")) // "test@example.com"

	// FindAllString: Find all matches in the string
	fmt.Println(re.FindAllString("test1@example.com test2@example.com",
-1)) // ["test1@example.com", "test2@example.com"]

	// ReplaceAllString: Replace all matches with a replacement string
	fmt.Println(re.ReplaceAllString("test@example.com", "REPLACED")) //
"REPLACED"

	// Split: Split the string by matches of the pattern
	fmt.Println(re.Split("test1@example.com test2@example.com", -1)) //
["", " ", ""]
}
```

---

4. Practical Use Cases

- **Validation**: Email, phone numbers, postal codes.
- **Search and Replace**: Find and replace patterns in text.
- **Data Extraction**: Extract specific data from strings (e.g., log files, web scraping).

Understanding Literal Characters and Metacharacters in Regular Expressions

Literal Characters

Literal characters are the simplest elements in a regular expression. They match exactly what you type. If you use a literal character in a regex, it will match that specific character in the input string.

Example:

- The regex a matches the character a.
- The regex cat matches the string cat.

Test Cases for Literal Characters:

1. **Regex:** a
    - **String:** "apple"
    - **Match:** Yes (matches the first 'a' in "apple")
2. **Regex:** cat
    - **String:** "I have a cat."
    - **Match:** Yes (matches "cat" in the string)
3. **Regex:** dog
    - **String:** "I have a cat."
    - **Match:** No (there is no "dog" in the string)

Metacharacters

Metacharacters are special characters in regular expressions that have specific meanings. They help in creating complex patterns for matching various text sequences.

Here's a list of some common metacharacters and their meanings:

- . : Matches any single character except a newline.
- ^ : Matches the beginning of a string.
- $: Matches the end of a string.
- * : Matches 0 or more occurrences of the preceding element.
- + : Matches 1 or more occurrences of the preceding element.
- ? : Matches 0 or 1 occurrence of the preceding element.
- [] : Matches any single character within the brackets.
- [^] : Matches any single character not in the brackets.
- \ : Escapes a metacharacter to treat it as a literal character.
- | : Alternation, matches either the expression before or the expression after the |.
- () : Groups expressions together.

Detailed Explanation and Test Cases:

1. **Dot (.)**
   Explanation: Matches any single character except a newline.
   Regex: c.t
   - **String:** "cat"
   - **Match:** Yes (matches "cat")
   - **String:** "cot"
   - **Match:** Yes (matches "cot")
   - **String:** "cut"
   - **Match:** Yes (matches "cut")
2. **Caret (^)**
   Explanation: Matches the beginning of a string.
   Regex: ^a
   - **String:** "apple"
   - **Match:** Yes (matches 'a' at the beginning)
   - **String:** "banana"
   - **Match:** No (does not start with 'a')
3. **Dollar ($)**
   Explanation: Matches the end of a string.
   Regex: t$

- **String:** "cat"
- **Match:** Yes (matches 't' at the end)
- **String:** "catfish"
- **Match:** No (does not end with 't')

4. **Asterisk (*)**
   Explanation: Matches 0 or more occurrences of the preceding element.
   Regex: ca*t
   - **String:** "ct"
   - **Match:** Yes (0 'a's followed by 't')
   - **String:** "cat"
   - **Match:** Yes (1 'a' followed by 't')
   - **String:** "caaaat"
   - **Match:** Yes (multiple 'a's followed by 't')
5. **Plus (+)**
   Explanation: Matches 1 or more occurrences of the preceding element.
   Regex: ca+t
   - **String:** "ct"
   - **Match:** No (0 'a's, needs at least one 'a')
   - **String:** "cat"
   - **Match:** Yes (1 'a' followed by 't')
   - **String:** "caaaat"
   - **Match:** Yes (multiple 'a's followed by 't')
6. **Question Mark (?)**
   Explanation: Matches 0 or 1 occurrence of the preceding element.
   Regex: ca?t
   - **String:** "ct"
   - **Match:** Yes (0 'a's followed by 't')
   - **String:** "cat"
   - **Match:** Yes (1 'a' followed by 't')
   - **String:** "caaat"
   - **Match:** No (more than 1 'a')
7. **Square Brackets ([])**
   Explanation: Matches any single character within the brackets.
   Regex: [cb]at
   - **String:** "cat"
   - **Match:** Yes (matches 'c' in "cat")
   - **String:** "bat"
   - **Match:** Yes (matches 'b' in "bat")
   - **String:** "rat"

- **Match:** No (does not match 'r')

8. **Negation ([^])**
   Explanation: Matches any single character not in the brackets.
   Regex: [^b]at
   - **String:** "cat"
   - **Match:** Yes (does not match 'b')
   - **String:** "bat"
   - **Match:** No (matches 'b')
   - **String:** "rat"
   - **Match:** Yes (does not match 'b')
9. **Backslash (\)**
   Explanation: Escapes a metacharacter to treat it as a literal character.
   Regex: \^abc
   - **String:** "^abc"
   - **Match:** Yes (matches '^abc')
   - **String:** "abc"
   - **Match:** No (does not match 'abc')
10. **Pipe (|)**
    Explanation: Alternation, matches either the expression before or the expression after the |.
    Regex: cat|dog
    - **String:** "cat"
    - **Match:** Yes (matches 'cat')
    - **String:** "dog"
    - **Match:** Yes (matches 'dog')
    - **String:** "bird"
    - **Match:** No (does not match 'bird')
11. **Parentheses (())**
    Explanation: Groups expressions together.
    Regex: a(bc)+
    - **String:** "abc"
    - **Match:** Yes (matches 'abc')
    - **String:** "abcbc"
    - **Match:** Yes (matches 'abcbc')
    - **String:** "ac"
    - **Match:** No (does not match 'ac')

Mathematical Expression for Literal Characters in Regular Expressions

Definitions and Notations

- **Literal Characters (LC)**: Characters that match themselves in a regular expression.
- **Alphabet Set**: Denoted as **Σ**, which includes:
  - **Alphabetic Characters: (LC_{char}): {a,b,c,...,z,A,B,C,...,Z}**
  - **Numeric Characters: (LC_{int}):{0,1,2,3,...,9}**

Union of Character Sets

The set of all literal characters (LC) can be expressed as the union of alphabetic characters and numeric characters:

$$LC = LC_{char} \cup LC_{int}$$

Regular Expression Representation

The regular expression for matching any single literal character in the combined set LC can be represented as:

$$RegExLC = [a-zA-Z0-9]$$

This regular expression $RegEx_{LC}$ matches any single character from the combined set of lowercase alphabetic, uppercase alphabetic, and numeric characters.

Formal Definition

Formally, the expression for literal characters in regular expressions is:

$$\mathbf{RegEx_{LC} = \{LC_{char} \cup LC_{int}\}}$$

Here, $\mathbf{\{LC_{int} \cup LC_{int}\}}$ denotes the set of all characters included in the regular expression.

Boundary Conditions

To denote the start and end boundaries of the string (if necessary), the regular expression can be wrapped with boundary markers:

$$\mathbf{RegEx = \wedge\{RegEx_{LC}\}\$}$$

Where:

- ^ asserts the position at the start of the line.
- $ asserts the position at the end of the line.

Explanation of Components

1. **Literal Characters (LC)**: Characters that are used as-is in regular expressions without any special meaning.

2. **Character Sets**:
    - **LC_{char}**: Set of alphabetic characters from `a` to `z` and `A` to `Z`.
    - **LC_{int}:** Set of numeric characters from `0` to `9`.
3. **Union of Sets ($\cup$)**: Combines the two sets LC_{char} and LC_{int} into a single set **LC**.
4. **Regular Expression ($RegEx_{LC}$)**: The pattern that matches any character from the combined set **LC**.

Example Regular Expressions

- To match a single alphabetic character (either lowercase or uppercase): `[a-zA-Z]`
- To match a single numeric character: `[0-9]`
- To match any single alphabetic or numeric character: `[a-zA-Z0-9]`

We will express this equation into a code in Golang which demonstrates matching numerical values using regular expressions and the proves it's mathematical existense:

```
package main


import (
	"fmt"
	"regexp"
)


func main() {
	examples := []string{
		"12345", "6789", "01234", "12345678", "7890",
"123456789", "34567", "a1b", "a2b", "123-45-6789", "123456789",
	}


	patterns := map[string]string{
		"literal digit '5'":                "5",
		"literal number '123'":             "123",
		"digit in any position '8'":        "8",
		"literal number '90'":              "90",
		"literal number '456'":             "456",
```

```
            "literal digit '2'":                  "2",
            "combining characters 'a1b'":         "a1b",
            "SSN pattern '\\d{3}-\\d{2}-\\d{4}'":
`\d{3}-\d{2}-\d{4}`,
      }

      for desc, pattern := range patterns {
            fmt.Printf("Pattern: %s (%s)\n", pattern, desc)
            re := regexp.MustCompile(pattern)
            for _, example := range examples {
                  if re.MatchString(example) {
                        fmt.Printf("  %s -> MATCH\n", example)
                  } else {
                        fmt.Printf("  %s -> NO MATCH\n", example)
                  }
            }
            fmt.Println()
      }
}
```

Here, all the test-cases are tested against the set of rules or simply to be said Regular Expression.

The output is returned as traversal of the whole array of test-cases.

OUTPUT:-

```
Start time:  2024-07-26 22:29:30.365111682 +0530 IST m=+0.000087449
Pattern: 2 (literal digit '2')
  12345 -> MATCH
  6789 -> NO MATCH
  01234 -> MATCH
  12345678 -> MATCH
  7890 -> NO MATCH
  123456789 -> MATCH
  34567 -> NO MATCH
  a1b -> NO MATCH
  a2b -> MATCH
  123-45-6789 -> MATCH
  123456789 -> MATCH

Pattern: a1b (combining characters 'a1b')
  12345 -> NO MATCH
  6789 -> NO MATCH
  01234 -> NO MATCH
  12345678 -> NO MATCH
  7890 -> NO MATCH
  123456789 -> NO MATCH
  34567 -> NO MATCH
  a1b -> MATCH
  a2b -> NO MATCH
```

```
  123-45-6789 -> NO MATCH
  123456789 -> NO MATCH

Pattern: \d{3}-\d{2}-\d{4} (SSN pattern '\d{3}-\d{2}-\d{4}')
  12345 -> NO MATCH
  6789 -> NO MATCH
  01234 -> NO MATCH
  12345678 -> NO MATCH
  7890 -> NO MATCH
  123456789 -> NO MATCH
  34567 -> NO MATCH
  a1b -> NO MATCH
  a2b -> NO MATCH
  123-45-6789 -> MATCH
  123456789 -> NO MATCH

Pattern: 5 (literal digit '5')
  12345 -> MATCH
  6789 -> NO MATCH
  01234 -> NO MATCH
  12345678 -> MATCH
  7890 -> NO MATCH
  123456789 -> MATCH
  34567 -> MATCH
```

```
  a1b -> NO MATCH
  a2b -> NO MATCH
  123-45-6789 -> MATCH
  123456789 -> MATCH

Pattern: 123 (literal number '123')
  12345 -> MATCH
  6789 -> NO MATCH
  01234 -> MATCH
  12345678 -> MATCH
  7890 -> NO MATCH
  123456789 -> MATCH
  34567 -> NO MATCH
  a1b -> NO MATCH
  a2b -> NO MATCH
  123-45-6789 -> MATCH
  123456789 -> MATCH

Pattern: 8 (digit in any position '8')
  12345 -> NO MATCH
  6789 -> MATCH
  01234 -> NO MATCH
  12345678 -> MATCH
  7890 -> MATCH
```

```
  123456789 -> MATCH
  34567 -> NO MATCH
  a1b -> NO MATCH
  a2b -> NO MATCH
  123-45-6789 -> MATCH
  123456789 -> MATCH

Pattern: 90 (literal number '90')
  12345 -> NO MATCH
  6789 -> NO MATCH
  01234 -> NO MATCH
  12345678 -> NO MATCH
  7890 -> MATCH
  123456789 -> NO MATCH
  34567 -> NO MATCH
  a1b -> NO MATCH
  a2b -> NO MATCH
  123-45-6789 -> NO MATCH
  123456789 -> NO MATCH

Pattern: 456 (literal number '456')
  12345 -> NO MATCH
  6789 -> NO MATCH
  01234 -> NO MATCH
```

```
  12345678 -> MATCH

  7890 -> NO MATCH

  123456789 -> MATCH

  34567 -> MATCH

  a1b -> NO MATCH

  a2b -> NO MATCH

  123-45-6789 -> NO MATCH

  123456789 -> MATCH


Total execution time: 495.049µs

------------------------------------------------------------------
```

Mathematical Expression for Metacharacters in Regular Expressions

Definitions and Notations

- **Metacharacters (MC)**: Special characters in regular expressions that have specific meanings and are not treated as literal characters.
- **Common Metacharacters**:
  - `.` : Matches any single character except newline.
  - `^` : Matches the start of a line.
  - `$` : Matches the end of a line.
  - `*` : Matches 0 or more occurrences of the preceding element.
  - `+` : Matches 1 or more occurrences of the preceding element.

- ? : Matches 0 or 1 occurrence of the preceding element.
- [] : Matches any one of the characters enclosed within the square brackets.
- \ : Escapes a metacharacter, treating it as a literal character.
- | : Alternation operator, matches the expression before or after the |.
- () : Groups expressions.

Mathematical Representation of Metacharacters

1. **Dot Metacharacter (.)**:
   - Matches any single character except newline.

     $$\mathbf{RegEx_{MC} = .}$$

2. **Start of Line (^)**:
   - Asserts the position at the start of the line

     $$\mathbf{RegEx_{MC} = \wedge}$$

3. **End of Line ($)**:
   - Asserts the position at the end of the line.

     $$\mathbf{RegEx_{MC} = \$}$$

4. **Asterisk (*)**:
   - Matches 0 or more occurrences of the preceding element E.

     $$\mathbf{RegEx_{MC} = E*}$$

5. **Plus (+)**:
   - Matches 1 or more occurrences of the preceding element E.

     $$\mathbf{RegEx_{MC} = E+}$$

6. **Question Mark (?)**:
   - Matches 0 or 1 occurrence of the preceding element E.

     $$\mathbf{RegEx_{MC} = E?}$$

7. **Character Class ([])**:
   - Matches any one of the characters enclosed within the square brackets.

     $$\mathbf{RegEx_{MC} = [C_1,C_2,\ldots,C_n]}$$

8. **Escape Character ()**:
    - Escapes a metacharacter, treating it as a literal character.

      $$\mathbf{RegEx_{MC} = M}$$

9. **Alternation (|)**:
    - Matches the expression before or after the **|**.

      $$\mathbf{RegEx_{MC} = E_1 \mid E_2}$$

10. **Grouping (())**:
    - Group expressions to apply operators to the grouped subexpression.

      $$\mathbf{(E)RegEx_{MC} = (E)}$$

Combining Metacharacters

The combination of metacharacters can be represented mathematically by applying the rules of each metacharacter to form more complex expressions.

These Expressions can be proved with set of codes in Golang:

```
package main

import (

	"fmt"

	"regexp"

)


func main() {

	examples := []string{

		"apple", "banana", "cat", "cot", "cut", "ct", "cat",
"caaaat",

		"cat", "bat", "rat", "^abc", "abc", "dog", "cat", "bird",
"abc", "abcbc", "ac",
```

```
}


patterns := map[string]string{
      "dot (.)":                              "c.t",
      "caret (^) at start":                   "^a",
      "dollar ($) at end":                    "t$",
      "asterisk (*) 0 or more 'a's":          "ca*t",
      "plus (+) 1 or more 'a's":              "ca+t",
      "question mark (?) 0 or 1 'a'":         "ca?t",
      "square brackets [cb]":                 "[cb]at",
      "negation [^b]":                        "[^b]at",
      "escaped caret (\\^)":                  `\^abc`,
      "alternation (|)":                      "cat|dog",
      "grouping (a(bc)+)":                    "a(bc)+",
}


for desc, pattern := range patterns {
      fmt.Printf("Pattern: %s (%s)\n", pattern, desc)
      re := regexp.MustCompile(pattern)
      for _, example := range examples {
            if re.MatchString(example) {
                  fmt.Printf("  %s -> MATCH\n", example)
            } else {
                  fmt.Printf("  %s -> NO MATCH\n", example)
```

```
                }
            }
            fmt.Println()
        }
}
```

One of the Test Case output would be:

```
Pattern: ca?t (question mark (?) 0 or 1 'a')
  apple -> NO MATCH
  banana -> NO MATCH
  cat -> MATCH
  cot -> NO MATCH
  cut -> NO MATCH
  ct -> MATCH
  cat -> MATCH
  caaaat -> NO MATCH
  cat -> MATCH
  bat -> NO MATCH
```

```
rat -> NO MATCH

^abc -> NO MATCH

abc -> NO MATCH

dog -> NO MATCH

cat -> MATCH

bird -> NO MATCH

abc -> NO MATCH

abcbc -> NO MATCH

ac -> NO MATCH
```

An Application With Regular Expression

Creating an email recognition and verification software in Go involves several steps, including email syntax validation, domain validation, and possibly sending verification emails.

Where we use regular expressions to validate the basic structure of an email address.

CODE:

```
package main


import (

	"fmt"

	"log"

	"net"
```

```
	"net/smtp"
	"regexp"
	"strings"
)

func isValidEmail(email string) bool {
	const emailRegex = `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
	re := regexp.MustCompile(emailRegex)
	return re.MatchString(email)
}

func isDomainValid(email string) bool {
	parts := strings.Split(email, "@")
	if len(parts) != 2 {
		return false
	}
	domain := parts[1]
	_, err := net.LookupMX(domain)
	return err == nil
}

func sendVerificationEmail(email, token string) error {
	from := "your-email@example.com"
	password := "your-email-password"
	smtpHost := "smtp.example.com"
	smtpPort := "587"
```

```
	to := []string{email}

	message := []byte(fmt.Sprintf("Subject: Email Verification\n\nPlease verify
your email by clicking the following link: http://yourdomain.com/verify?token=%s",
token))



	auth := smtp.PlainAuth("", from, password, smtpHost)

	err := smtp.SendMail(smtpHost+":"+smtpPort, auth, from, to, message)

	if err != nil {

		return err

	}

	return nil

}



func main() {

	email := "test@example.com"

	if isValidEmail(email) && isDomainValid(email) {

		token := "unique-verification-token"

		err := sendVerificationEmail(email, token)

		if err != nil {

			log.Fatalf("Failed to send verification email: %v", err)

		} else {

			fmt.Println("Verification email sent successfully!")

		}

	} else {

		fmt.Println("Invalid email address.")

	}

}
```

Features of this application:-

- Validate email format using regular expression.
- Verify the domain of the e-mail address.
- Send verification email with a unique token.

How Regular Expression Helps?

- `^[a-zA-Z0-9._%+-]+`**:** Ensures the local part of the email (before the `@`) contains one or more valid characters.
- `@[a-zA-Z0-9.-]+`**:** Ensures there is a single `@` character followed by one or more valid characters for the domain part.
- `\.[a-zA-Z]{2,}$`**:** Ensures the domain part contains a period followed by at least two valid characters for the TLD, and the pattern matches up to the end of the string.

Example Matches

- `test@example.com`**: Valid email address**
- `user.name+tag@sub.domain.co`**: Valid email address**
- `first.last@domain`**: Invalid (missing TLD)**
- `@domain.com`**: Invalid (missing local part)**
- `user@.com`**: Invalid (missing domain part before the period)**

Conclusion

This paper presents a comprehensive study of regular expressions in Golang, including both literal characters and metacharacters. By providing detailed explanations, mathematical representations, and practical implementations, we aim to facilitate the understanding and application of regular expressions in Go for various pattern recognition tasks.

Future Work

Further research can explore advanced regular expression techniques and optimization strategies in Golang, including performance benchmarking and real-world applications.

References

- **GO DOCUMENTATION:** https://go.dev/doc/
- **REGEX Package Documentation: https://pkg.go.dev/regexp**
- **Regular-expression website: https://www.regular-expressions.info/**
- **BOOKS:**
    - "Mastering Regular Expressions" by Jeffrey E. F. Friedl.
    - "Regular Expressions Cookbook" by Jan Goyvaerts and Steven Levithan.

GIT-HUB Links:

- **Email Verification Application: https://github.com/Jaywantadh/Email-verification**
- **All Regular Expression Test Codes:- https://github.com/Jaywantadh/RegEx**