REGEX em Rust

Cleuton Sampaio

Link para o Repo

O que é REGEX?

Regex (abreviação de *Regular Expressions*, ou expressões regulares) é uma linguagem para definir padrões usados para encontrar e manipular texto. Ela é amplamente utilizada em programação para:

- Validar entradas, como emails, senhas ou números de telefone.
- Buscar e substituir partes de texto em documentos ou arquivos.
- Analisar e processar strings em dados estruturados.

Uma regex é composta por símbolos e operadores que descrevem padrões. Alguns exemplos:

- `\d`: Corresponde a um dígito (0–9).

- `[a-zA-Z]`: Corresponde a qualquer letra maiúscula ou minúscula.
- `+`: Um ou mais caracteres.
- `^`: Indica o início de uma string.
- `$`: Indica o final de uma string.

Por exemplo, a regex `^\d+$` valida strings compostas apenas por dígitos (`123`, `4567`, etc.).

Ge rador de REGEX em Rust

Fiz um código bem simples, que pode ler um texto e gerar a regex correspontente, por exemplo:

```
"início da string seguido de um nome de usuário válido seguido de um
símbolo arroba seguido de um domínio válido seguido de uma extensão de
domínio seguido de fim da string"
```

Explicando o Código Rust

Eis o codigo:

```rust
use std::collections::HashMap;
use regex::Regex;

fn gerar_regex(comando: &str) -> Result<String, String> {
    let mut mapeamento = HashMap::new();
    mapeamento.insert("uma letra", "[a-zA-Z]");
    mapeamento.insert("um dígito", "\\d");
    mapeamento.insert("qualquer caractere", ".");
    mapeamento.insert("opcional", "?");
    mapeamento.insert("início da string", "^");
    mapeamento.insert("fim da string", "$");
    mapeamento.insert("uma ou mais vezes", "+");
    mapeamento.insert("nenhuma ou mais vezes", "*");
    mapeamento.insert("um nome de usuário válido", "[a-zA-Z0-9._%+-]+");
    mapeamento.insert("um símbolo arroba", "@");
    mapeamento.insert("um domínio válido", "[a-zA-Z0-9.-]+");
    mapeamento.insert("uma extensão de domínio", "\\.[a-zA-Z]{2,}");

    let mut regex = String::new();
    let partes: Vec<&str> = comando.split("seguido de").map(|s|
s.trim()).collect();

    for parte in partes {
        let mut encontrado = false;
        for (chave, valor) in &mapeamento {
            if parte.contains(chave) {
                regex.push_str(valor);
                encontrado = true;

                // Verifica por repetições
                if parte.contains("repetido de") {
```

```
                    if let Some(intervalo) = parte.split("repetido
de").nth(1) {
                        let intervalo = intervalo.trim();
                        regex.push_str(&format!("{{{}}}", intervalo));
                    }
                }
                break;
            }
        }
        if !encontrado {
            return Err(format!("Comando não reconhecido: '{}'", parte));
        }
    }

    Ok(regex)
}

fn main() {
    // Comando para gerar a regex para validar emails
    let comando = "início da string seguido de um nome de usuário válido
seguido de um símbolo arroba seguido de um domínio válido seguido de uma
extensão de domínio seguido de fim da string";

    match gerar_regex(comando) {
        Ok(regex_str) => {
            println!("Regex gerada: {}", regex_str);

            // Compile a regex gerada
            match Regex::new(&regex_str) {
                Ok(regex) => {
                    // Testar a regex com emails de exemplo
                    let emails = vec![
                        "usuario@exemplo.com",
                        "teste.email@dominio.org",
                        "email_invalido@dominio",
                        "@semnome.com",
                        "nome@.com",
                        "email@dominio.com.br",
                    ];

                    for email in emails {
                        if regex.is_match(email) {
                            println!("'{}' é um email válido.", email);
                        } else {
                            println!("'{}' NÃO é um email válido.",
email);
                        }
                    }
                }
                Err(err) => eprintln!("Erro ao compilar a regex: {}",
err),
            }
        }
        Err(err) => eprintln!("Erro: {}", err),
```

```
    }
}
```

O código Rust que criei faz duas coisas principais:

1. **Gera uma regex automaticamente a partir de uma linguagem simples e legível.**
2. **Valida entradas (como emails) usando a regex gerada.**

Função `gerar_regex`

Essa função é responsável por converter uma descrição em linguagem simples para uma regex. Aqui está o passo a passo:

1. **Mapeamento de padrões**: Usamos um `HashMap` para associar palavras-chave (como `"uma letra"` ou `"um dígito"`) a trechos de regex correspondentes:

   ```
   mapeamento.insert("uma letra", "[a-zA-Z]");
   mapeamento.insert("um dígito", "\\d");
   mapeamento.insert("um nome de usuário válido", "[a-zA-Z0-9._%+-]+");
   ```

   Isso permite traduzir descrições como `"um nome de usuário válido"` para `[a-zA-Z0-9._%+-]+`.

2. **Divisão do comando**: O comando fornecido é dividido em partes com base na palavra `"seguido de"`. Isso ajuda a processar cada componente da descrição separadamente.

3. **Construção da regex**: Cada parte é comparada com o mapeamento. Se houver correspondência, o trecho da regex é adicionado à string final.

4. **Erro para comandos não reconhecidos**: Se o comando contiver algo fora do mapeamento, um erro é retornado:

   ```
   return Err(format!("Comando não reconhecido: '{}'", parte));
   ```

Regex Gerada

Quando o comando para validar email é fornecido:

```
início da string seguido de um nome de usuário válido seguido de um
símbolo arroba seguido de um domínio válido seguido de uma extensão de
domínio seguido de fim da string
```

A regex gerada é:

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
```

Essa regex segue as regras gerais de validação de emails:

- `[a-zA-Z0-9._%+-]+`: Nome do usuário (letras, números e caracteres como `_`, `.`, `%`, `+`, `-`).
- `@`: O símbolo arroba obrigatório.
- `[a-zA-Z0-9.-]+`: Domínio válido (letras, números, `-` e `.`).
- `\\.[a-zA-Z]{2,}`: Extensão do domínio, começando com um ponto (`.`) e tendo pelo menos dois caracteres alfabéticos.

Validação de Emails

No `main`, testamos a regex gerada em uma lista de emails:

1. **Compilar a regex**:

   ```
   match Regex::new(&regex_str) {
   ```

   Usamos a crate `regex` para compilar a string de regex gerada.

2. **Testar strings**: Para cada email na lista, usamos `is_match` para verificar se ele corresponde à regex:

   ```
   if regex.is_match(email) {
       println!("'{}' é um email válido.", email);
   } else {
       println!("'{}' NÃO é um email válido.", email);
   }
   ```

3. **Exemplos de saída**:

   - `"usuario@exemplo.com"` → válido.
   - `"email_invalido@dominio"` → inválido (faltando extensão).
   - `"nome@.com"` → inválido (domínio incompleto).