Shunting Yard em Rust

Cleuton Sampaio

Um dos grandes desafios em algoritmos é a construção de interpretadores e compiladores. E, dentro desse desafio, analisar expressões é sempre vista como uma tarefa complexa.

Felizmente, temos algoritmos clássicos para isso, como o **Shunting yard**:

> É um método para analisar expressões aritméticas ou lógicas, ou uma combinação de ambas, especificadas em notação infixa. Ele pode produzir uma string de notação pós-fixada, também conhecida como notação polonesa reversa (RPN), ou uma árvore de sintaxe abstrata (AST).O algoritmo foi inventado pelo Professor Doutor **Edsger Dijkstra** e denominado algoritmo de "pátio de manobras" porque sua operação se assemelha à de um pátio de manobras ferroviárias. Dijkstra descreveu pela primeira vez o algoritmo do pátio de manobras no relatório Mathematisch Centrum.

Uma expressão infixa é uma expressão aritmética onde os operadores são colocados entre os operandos. Esse é o formato comum que usamos na matemática e nas calculadoras básicas. Por exemplo, a expressão

aritmética `(3 + 4) * 5` é uma expressão infixa.

Aqui estão alguns exemplos para esclarecer melhor:

Exemplos de Expressões Infixas

1. **Simples**:
   - `3 + 4`
   - `5 - 2`
2. **Com Parênteses**:
   - `(3 + 4) * 5`
   - `7 / (2 + 3)`
3. **Com Vários Operadores**:
   - `3 + 4 * 2`
   - `8 / 2 - 3`
4. **Com Funções**:
   - `SIN(30) + COS(60)`
   - `EXP(2) * 5`

Características das Expressões Infixas

1. **Ordem dos Operadores**: Os operadores são colocados entre os operandos.
   - Exemplo: `A + B`, onde `+` é o operador e `A` e `B` são os operandos.
2. **Uso de Parênteses**: Parênteses são usados para alterar a ordem natural das operações e garantir que certas operações sejam realizadas primeiro.
   - Exemplo: `(A + B) * C` garante que `A + B` é calculado antes de multiplicar por `C`.
3. **Precedência de Operadores**: Diferentes operadores têm diferentes níveis de precedência. Multiplicação e divisão têm precedência mais alta que adição e subtração.
   - Exemplo: Na expressão `A + B * C`, a multiplicação `B * C` é realizada antes da adição `A +`.
4. **Associação**: Define a ordem em que operadores do mesmo nível de precedência são avaliados. A maioria dos operadores aritméticos são associativos à esquerda, o que significa que a avaliação é feita da esquerda para a direita.
   - Exemplo: Na expressão `A - B - C`, a avaliação é feita como `(A - B) - C`.

Comparação com Outras Notações

1. **Notação Prefixa (Notação Polonesa)**:
   - Os operadores precedem os operandos.
   - Exemplo: `+ A B` em vez de `A + B`.

2. **Notação Posfixa (Notação Polonesa Reversa)**:
    - Os operadores seguem os operandos.
    - Exemplo: `A B +` em vez de `A + B`.

Exemplo de Conversão de Infixa para Posfixa

Considere a expressão infixa: `(3 + 4) * 5`

- **Passo 1**: Avalie o conteúdo dos parênteses:
    - `3 + 4` resulta em `7`.
- **Passo 2**: Substitua a expressão avaliada no lugar dos parênteses:
    - A expressão se torna `7 * 5`.
- **Passo 3**: Em notação posfixa, o operador `*` vem após os operandos:
    - A expressão `7 * 5` se torna `7 5 *`.

Portanto, a expressão infixa `(3 + 4) * 5` em notação posfixa é `3 4 + 5 *`.

Benefícios da Notação Posfixa

- **Eliminação de Parênteses**: Parênteses não são necessários, pois a ordem das operações é clara.
- **Facilidade de Avaliação**: As expressões posfixas podem ser avaliadas de maneira simples usando uma pilha.

Implementar a conversão de infixa para posfixa pode ser particularmente útil em contextos onde a expressão precisa ser avaliada programaticamente de maneira eficiente, como em compiladores e interpretadores de linguagens de programação.

Implementação em Rust

Eu implementei esse algoritmo em C++ desta maneira:

```rust
/*
Implementação do algoritmo Shunting Yard © 2024 por Cleuton Sampaio
licenciado sob CC BY-SA 4.0. Para ver uma cópia desta licença,
visite https://creativecommons.org/licenses/by-sa/4.0/
*/

use std::f64::consts::PI;

// Verifica se um caractere é operador
fn e_operador(c: char) -> bool {
    c == '+' || c == '-' || c == '*' || c == '/' || c == '^'
}

// Retorna a precedência do operador
fn precedencia(op: char) -> i32 {
```

```
    match op {
        '+' | '-' => 1,
        '*' | '/' => 2,
        '^' => 3,
        _ => 0,
    }
}

// Verifica se uma string é uma função válida
fn e_funcao(token: &str) -> bool {
    token == "EXP" || token == "SQR" || token == "SIN" || token == "COS"
}

// Analisa a expressão e divide em tokens
fn tokenizar(infixa: &str) -> Vec<String> {
    let mut tokens = Vec::new();
    let mut token = String::new();
    let mut esperando_operando = true; // Indica se estamos esperando um
operando (usado para operadores unários)

    let chars: Vec<char> = infixa.chars().collect();
    let mut i = 0;
    while i < chars.len() {
        if chars[i].is_whitespace() {
            i += 1;
            continue;
        }
        if chars[i].is_digit(10) || chars[i] == '.' {
            token.push(chars[i]);
            while i + 1 < chars.len() && (chars[i + 1].is_digit(10) ||
chars[i + 1] == '.') {
                i += 1;
                token.push(chars[i]);
            }
            tokens.push(token.clone());
            token.clear();
            esperando_operando = false;
        } else if chars[i].is_alphabetic() {
            token.push(chars[i]);
            while i + 1 < chars.len() && chars[i + 1].is_alphabetic() {
                i += 1;
                token.push(chars[i]);
            }
            tokens.push(token.clone());
            token.clear();
            esperando_operando = false;
        } else if e_operador(chars[i]) {
            if esperando_operando && chars[i] == '-' {
                // Trata o operador unário
                token.push('-');
                if i + 1 < chars.len() && (chars[i + 1].is_digit(10) ||
chars[i + 1] == '.') {
                    i += 1;
                    token.push(chars[i]);
```

```
                    while i + 1 < chars.len() && (chars[i +
1].is_digit(10) || chars[i + 1] == '.') {
                        i += 1;
                        token.push(chars[i]);
                    }
                }
                tokens.push(token.clone());
                token.clear();
                esperando_operando = false;
            } else {
                tokens.push(chars[i].to_string());
                esperando_operando = true;
            }
        } else if chars[i] == '(' || chars[i] == ')' {
            tokens.push(chars[i].to_string());
            esperando_operando = chars[i] == '(';
        } else {
            // Caractere inválido
            tokens.push(chars[i].to_string());
            esperando_operando = true;
        }
        i += 1;
    }
    tokens
}

// Verifica se uma expressão infixa é válida
fn validar_expressao(tokens: &[String]) -> &str {
    let mut balanceamento_parenteses = 0;
    for i in 0..tokens.len() {
        let token = &tokens[i];
        if token == "(" {
            balanceamento_parenteses += 1;
        } else if token == ")" {
            balanceamento_parenteses -= 1;
            if balanceamento_parenteses < 0 {
                return "Parenteses desbalanceados";
            }
        } else if e_operador(token.chars().next().unwrap()) && token.len()
== 1 {
            if i == 0 || i == tokens.len() - 1 {
                return "Expressao invalida"; // Operador no início ou no
fim
            }
            if i + 1 < tokens.len()
                && e_operador(tokens[i + 1].chars().next().unwrap())
                && tokens[i + 1].len() == 1
            {
                return "Expressao invalida"; // Operadores duplos
            }
        } else if e_funcao(token) {
            if i + 1 >= tokens.len() || tokens[i + 1] != "(" {
                return "Expressao invalida"; // Função deve ser seguida
por '('
```

```
            }
        } else if token.chars().next().unwrap().is_digit(10)
            || (token.len() > 1 &&
token.chars().nth(1).unwrap().is_digit(10))
        {
            // Número é considerado válido
        } else {
            return "Expressao invalida";
        }
    }
    if balanceamento_parenteses == 0 {
        "OK"
    } else {
        "Parenteses desbalanceados"
    }
}

// Converte uma expressão infixa para posfixa usando o algoritmo de
Shunting Yard
fn infixa_para_posfixa(infixa: &str) -> String {
    let mut operadores: Vec<String> = Vec::new();
    let mut saida = String::new();
    let tokens = tokenizar(infixa);

    let resultado_validacao = validar_expressao(&tokens);
    if resultado_validacao != "OK" {
        return resultado_validacao.to_string();
    }

    for token in tokens {
        if token.chars().next().unwrap().is_digit(10)
            || (token.len() > 1 &&
token.chars().nth(1).unwrap().is_digit(10))
        {
            // Token é um operando (número)
            saida.push_str(&token);
            saida.push(' ');
        } else if e_funcao(&token) {
            // Token é uma função
            operadores.push(token);
        } else if token == "(" {
            // Token é um parêntese de abertura
            operadores.push(token);
        } else if token == ")" {
            // Token é um parêntese de fechamento
            while !operadores.is_empty() && operadores.last().unwrap() !=
"(" {
                saida.push_str(&operadores.pop().unwrap());
                saida.push(' ');
            }
            if !operadores.is_empty() && operadores.last().unwrap() == "("
{
                operadores.pop(); // Remove o '('
            } else {
```

```
                return "Parenteses desbalanceados".to_string();
            }
            if !operadores.is_empty() &&
e_funcao(operadores.last().unwrap()) {
                saida.push_str(&operadores.pop().unwrap());
                saida.push(' ');
            }
        } else if e_operador(token.chars().next().unwrap()) {
            while !operadores.is_empty()
                && ((token.chars().next().unwrap() != '^'
                    &&
precedencia(operadores.last().unwrap().chars().next().unwrap())
                        >= precedencia(token.chars().next().unwrap()))
                    || (token.chars().next().unwrap() == '^'
                        &&
precedencia(operadores.last().unwrap().chars().next().unwrap())
                            > precedencia(token.chars().next().unwrap())))
            {
                saida.push_str(&operadores.pop().unwrap());
                saida.push(' ');
            }
            operadores.push(token);
        }
    }

    // Esvazia a pilha de operadores
    while !operadores.is_empty() {
        if operadores.last().unwrap() == "(" {
            return "Parenteses desbalanceados".to_string();
        }
        saida.push_str(&operadores.pop().unwrap());
        saida.push(' ');
    }

    saida
}

// Calcula o seno de um ângulo em graus
fn seno_graus(valor: f64) -> f64 {
    (valor * PI / 180.0).sin()
}

// Calcula o cosseno de um ângulo em graus
fn cosseno_graus(valor: f64) -> f64 {
    (valor * PI / 180.0).cos()
}

// Avalia uma expressão em notação polonesa reversa (RPN)
fn avaliar_rpn(rpn: &str) -> Result<f64, String> {
    let mut pilha: Vec<f64> = Vec::new();
    let tokens = rpn.split_whitespace();

    for token in tokens {
        if token.chars().next().unwrap().is_digit(10)
```

```
            || (token.len() > 1 &&
token.chars().nth(1).unwrap().is_digit(10))
        {
            pilha.push(token.parse::<f64>().map_err(|_| "Número
inválido")?);
        } else if e_operador(token.chars().next().unwrap()) {
            if pilha.len() < 2 {
                return Err("Expressão RPN inválida".to_string());
            }
            let b = pilha.pop().unwrap();
            let a = pilha.pop().unwrap();
            match token.chars().next().unwrap() {
                '+' => pilha.push(a + b),
                '-' => pilha.push(a - b),
                '*' => pilha.push(a * b),
                '/' => pilha.push(a / b),
                '^' => pilha.push(a.powf(b)),
                _ => return Err("Operador desconhecido".to_string()),
            }
        } else if e_funcao(token) {
            if pilha.is_empty() {
                return Err("Expressão RPN inválida".to_string());
            }
            let a = pilha.pop().unwrap();
            if token == "SIN" {
                pilha.push(seno_graus(a));
            } else if token == "COS" {
                pilha.push(cosseno_graus(a));
            } else if token == "EXP" {
                pilha.push(a.exp());
            } else if token == "SQR" {
                pilha.push(a.sqrt());
            } else {
                return Err("Função desconhecida".to_string());
            }
        } else {
            return Err("Token desconhecido".to_string());
        }
    }

    if pilha.len() != 1 {
        return Err("Expressão RPN inválida".to_string());
    }
    Ok(pilha.pop().unwrap())
}

// Função de teste
fn executar_testes() {
    struct TestCase {
        infixa: &'static str,
        rpn_esperada: &'static str,
        valor_esperado: f64,
    }
```

```
    let casos_de_teste = vec![
        TestCase {
            infixa: "3+4*2/(1-5)^2^3",
            rpn_esperada: "3 4 2 * 1 5 - 2 3 ^ ^ / + ",
            valor_esperado: 3.0001220703125,
        },
        TestCase {
            infixa: "SIN(3+4)*COS(2-1)",
            rpn_esperada: "3 4 + SIN 2 1 - COS * ",
            valor_esperado: 0.121851,
        },
        TestCase {
            infixa: "-3+4*-2/(1--5)^2^3",
            rpn_esperada: "-3 4 -2 * 1 -5 - 2 3 ^ ^ / + ",
            valor_esperado: -3.0001220703125,
        },
        TestCase {
            infixa: "3++4",
            rpn_esperada: "Expressao invalida",
            valor_esperado: 0.0,
        },
        TestCase {
            infixa: "SIN(3+4)*INVALID(2-1)",
            rpn_esperada: "Expressao invalida",
            valor_esperado: 0.0,
        },
        TestCase {
            infixa: "3+4**2",
            rpn_esperada: "Expressao invalida",
            valor_esperado: 0.0,
        },
        TestCase {
            infixa: "(3+4",
            rpn_esperada: "Parenteses desbalanceados",
            valor_esperado: 0.0,
        },
    ];

    for teste in casos_de_teste {
        let rpn_resultante = infixa_para_posfixa(teste.infixa);
        println!("Infixa: {}", teste.infixa);
        println!("RPN esperada: {}", teste.rpn_esperada);
        println!("RPN retornada: {}", rpn_resultante);

        if rpn_resultante == teste.rpn_esperada {
            println!("Conversao em RPN: OK");
        } else {
            println!("Conversao em RPN: FALHA");
        }

        if rpn_resultante == teste.rpn_esperada
            && rpn_resultante != "Expressao invalida"
            && rpn_resultante != "Parenteses desbalanceados"
        {
```

```
            match avaliar_rpn(&rpn_resultante) {
                Ok(valor_resultante) => {
                    println!("Valor esperado: {}", teste.valor_esperado);
                    println!("Valor calculado: {:.6}", valor_resultante);
                    if (valor_resultante - teste.valor_esperado).abs() <
1e-6 {
                        println!("Avaliacao da RPN: OK");
                    } else {
                        println!("Avaliacao da RPN: FALHA");
                    }
                }
                Err(e) => {
                    println!("Erro na avaliação da RPN: {}", e);
                }
            }
        }
        println!("------------------------");
    }
}

fn main() {
    executar_testes();
}
```

Em vez de escrever uma função **tokenizadora** eu poderia utilizar algo do ecossistema **Rust**, como o **logos**, mas quis migrar diretamente do **C++** para Rust de modo que vocês possam ver as diferenças.

Há alguns erros de arredondamento, coisa boba, que vou resolver em breve.

Como gerar o projeto

Se ainda não tem **Rust** instale o **rustup**.

Crie um projeto com o `cargo`:

```
cargo new shunting_yard --bin
```

Troque o arquivo `src/main.rs` pela listagem acima. Execute o programa com:

```
cargo run
```

Confira minha **página de cursos! Me siga aqui!**