Cleuton Sampaio.

Caminho para o repo no GitHub

Domain Specific Language

Uma DSL (Domain-specific language) é uma linguagem de programação limitada a um domínio de problema, por exemplo, para parametrizar uma aplicação. DSLs tem sido utilizadas há muito tempo por muitas aplicações diferentes. Elas não servem para criar programas de uso geral, mas para parametrizar uma determinada aplicação, seja para modificar sua execução ou extrair dados.

Rust tem `crates` úteis para facilitar a análise léxica (lexer) e sintática (parser) de uma DSL, como o **logos** (analisador léxico) e o **nom** (analisador sintático).

Exemplo de DSL:

Vamos implementar uma DSL simples para expressões aritméticas como essa, analisando e retornando o resultado correto:

```
3 + 5 * (2 - 8)
```

Tokenização com logos

O que é um Token?

Um token é uma unidade básica de uma linguagem (números, operadores, parênteses, etc.). Por exemplo, na expressão `3 + 5`, os tokens são: `3`, `+`, `5`.

Como o logos ajuda?

A biblioteca `logos` é um lexer, ou seja, transforma texto bruto em tokens. Usamos `#[derive(Logos)]` para definir a lógica de conversão de texto para tokens:

```
#[regex(r"[0-9]+", |lex| lex.slice().parse())]
Numero(i32),
```

Essa declaração captura números inteiros (`[0-9]+`) e os converte em um token `Numero`.

Tokens como `+`, `-`, `*` e `/` são definidos com `#[token("<operador>")]`, associando diretamente o caractere ao tipo, e Espaços em branco são ignorados com:

```
#[regex(r"[ \t\n\f]+", logos::skip)]
```

Abstract Syntax Tree (AST)

A **AST** é uma estrutura de dados em árvore que representa a hierarquia da expressão matemática. Por exemplo, para `3 + 5 * 2`, a AST seria algo como:

```
Operacao(+)
    /   \
   3     Operacao(*)
            /    \
           5      2
```

Definição da estrutura da AST

```
enum Expr {
    Numero(i32),
    Operacao {
        op: Token,
        esquerda: Box<Expr>,
        direita: Box<Expr>,
```

```
    },
}
```

Cada nó da AST pode ser um número (`Numero`) ou uma operação (`Operacao`), com um operador (`op`) e dois operandos (`esquerda` e `direita`).

> Por que `Box<Expr>`? `Box` é usado porque `Expr` contém referências recursivas. Sem `Box`, o Rust não poderia determinar o tamanho de `Expr` em tempo de compilação.

Análise sintática (parsing)

O crate `nom` é uma biblioteca de parsing que ajuda a transformar tokens em estruturas mais complexas (como uma AST). Um parser analisa tokens passo a passo, verificando se eles correspondem a padrões esperados (números, operadores, etc.). Cada parser retorna:

- Uma fatia restante dos tokens (`&[Token]`).
- Um resultado correspondente ao que foi analisado.

Parsers principais no código:

1. **parse_numero:**
    - Identifica números na lista de tokens.

```
fn parse_numero(tokens: &[Token]) -> IResult<&[Token], Expr> {
    if let Some((Token::Numero(valor), rest)) = tokens.split_first() {
        Ok((rest, Expr::Numero(*valor)))
    } else {
        Err(nom::Err::Error(nom::error::Error::new(tokens,
nom::error::ErrorKind::Tag)))
    }
}
```

    - `split_first` separa o primeiro token (`Token::Numero`) do resto.
    - Se for um número, retorna a AST parcial `Expr::Numero`.

2. **parse_fator:**
    - Lida com fatores (números ou expressões entre parênteses):

```
alt((
    parse_numero,
    delimited(parse_abre_parenteses, parse_expr,
parse_fecha_parenteses),
))(tokens)
```

- `alt` tenta múltiplas opções. Primeiro, tenta `parse_numero`. Se falhar, tenta uma expressão entre parênteses (`delimited`).

3. **parse_termo:**
   - Processa multiplicação e divisão com **precedência**:

```
fold_many0(
    pair(parse_multiplicacao_ou_divisao, parse_fator),
    move || inicial.clone(),
    |esquerda, (op, direita)| Expr::Operacao {
        op,
        esquerda: Box::new(esquerda),
        direita: Box::new(direita),
    },
)
```

   - **pair**: Analisa um operador (`*` ou `/`) seguido de um fator.
   - **fold_many0**: Aplica múltiplas operações sequenciais, acumulando a AST.

4. **parse_expr:**
   - Processa soma e subtração com **precedência menor**:

```
fold_many0(
    pair(parse_mais_ou_menos, parse_termo),
    move || inicial.clone(),
    |esquerda, (op, direita)| Expr::Operacao {
        op,
        esquerda: Box::new(esquerda),
        direita: Box::new(direita),
    },
)
```

   - Semelhante a `parse_termo`, mas para `+` e `-`.

Avaliação (execução) da AST

O avaliador percorre a AST recursivamente, resolvendo e acumulando o resultado:

```
fn avaliar(expr: &Expr) -> i32 {
    match expr {
        Expr::Numero(valor) => *valor,
        Expr::Operacao { op, esquerda, direita } => {
            let esq = avaliar(esquerda);
            let dir = avaliar(direita);
            match op {
                Token::Mais => esq + dir,
```

```
                Token::Menos => esq - dir,
                Token::Multiplicacao => esq * dir,
                Token::Divisao => esq / dir,
                _ => panic!("Operador inválido"),
            }
        }
    }
}
```

> Por que **recursivo?** Cada nó da AST pode conter suboperações que precisam ser avaliadas antes.

Utilizando a DSL

Ao executar o programa (`cargo run "<expressão>"`) a entrada do usuário (uma string) é convertida em tokens com `logos`, que são processados pelo parser para construir a AST. Por fim, a AST é avaliada para produzir o resultado.

Coisas estranhas no código

1. **`Box` em AST recursiva:**
    - Necessário porque Rust exige saber o tamanho exato de tipos não recursivos em tempo de compilação.
2. **`fold_many0`:**
    - Cria operações encadeadas (ex.: `3 + 5 + 7`) sem precisar de loops explícitos.
3. **Erro com `nom`:**
    - O `nom::Err` é usado para lidar com falhas de parsing.
    - Exemplo:

```
Err(nom::Err::Error(nom::error::Error::new(tokens,
nom::error::ErrorKind::Tag)))
```