# LFE - Code is Data is Code (de novo)

Bruno Dias

Elixir Curitiba 2025

# LFE - Code is Data is Code

**Agenda**

- ➡ Sobre lisp
- ➡ Linguagem
- ➡ Implementações atuais
- ➡ LFE - Lisp favored lisp
- ➡ Macros e meta-programming
- ➡ “Quero ver código!“
- ➡ Dicas e pitfalls
- ➡ Resources

# Sobre lisp

Lisp aparece nos anos 60’, desenvolvido por John McCarthy, Steve Russell e equipe.

A comunidade é vibrante.

Também, possui canais em todas as plataformas, mas principalmente no discord.

**Code is Data is Code** é um termo que se refere à homoiconicidade de uma linguagem - a própria linguagem pode ser manipulada como informação.

# Linguagem

**Symbols e keywords**

**Examplo:**

Symbol - :a
Keyword - 'a

**Function call**

(symbol &rest args)

(funcall #’symbol &rest args)

(apply #’symbol args-list)

**Examplo:**

(+ 1 2)

(funcall #’+ 1 2)

(apply #’+ (list 1 2))

# Implementações atuais

➡ **LFE**

Lisp favored erlang

➡ **Common Lisp**

General purpose language

➡ **Coalton**

Statically typed lisp

➡ **Carp**

Statically typed lisp for realtime applications

➡ **Clojure**

Lisp on the JVM

➡ **ClojureScript / Parenscript**

Lisp on the browser

➡ **ecl**

Embeddable common lisp

➡ **lispBM**

Lisp for microcontrollers

# Implementações atuais

**GNU Emacs**

Desenvolvido por **Robert Virding**, co-criador do erlang,
e **Duncan MacGreggor** como um dos principais
colaboradores atuais.
O projeto já possui 17 anos, e está na versão **2.2.0**.
A linguagem é pode ser utilizada tanto em Elixir,
como erlang (possivelmente qualquer linguagem
que tenha o tanger a BEAM).

# Macros e Meta-programação

O objetivo das **macros** são a redução de boilerplate de códigos que seguem a mesma estrutura sem variação. Pode expandir em compile-time ou run-time.

**Meta-programação** é a capacidade de um programa gerar outros programas.

# Macros e Meta-programação

Algumas **macros** são peças de código que expandem em um compiler-pass (passo do compilador) que converte tokens e aplicação de variáveis em uma espécie de template.

Tokens são meramente aplicados onde houver o placeholder, não sendo possível  a utilização de transformações mais trabalhadas.

**Exemplo de macros em C:**

```
#define fn(ty, size)
   struct ty##_list_t {
      ty value[size];
   };
```

# Macros e Meta-programação

Com **meta-programação**, temos mais controle sobre como esse código será gerado e quais transformações podemos aplicar.

Exemplo de meta-programação em Zig:

```
fn Vec2Of(comptime T: type) type {
    return struct{
        x: T,
        y: T,
        size: i64 = @sizeOf(T)
    };
}

const V2i64 = Vec2Of(i64);
const V2f64 = Vec2Of(f64);
```

"Quero ver código!”

# Dicas e pitfalls

➡ Evite gerar variáveis que serão externalizadas. Receba por parâmetro.

Difícil de saber quais variáveis estarão disponíveis.

➡ Evite flags e switches que mudam o comportamento.

Se precisar de troca de comportamento, extraia-o e passe como parâmetro.

➡ Não use para substituir patterns.

Gerar código e patterns são aproximadamente equivalentes em alguns casos. Prefira pattern.

# Obrigado!

Elixir Curitiba 2025

# Resources

**Emacs** https://www.gnu.org/software/emacs/
**Nord Theme** https://www.nordtheme.com/

**LFE Github** https://github.com/lfe/lfe/
**LFE Cookbook** https://cnbbooks.github.io/lfe-tutorial/

**Code** https://github.com/diasbruno/elixir-curitiba-2025

**Site** https://diasbruno.github.io
**Github** https://github.com/diasbruno/
**Substack** https://diasbruno.substack.com/