



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

Trabalho Prático I

1. Objetivo

O objetivo desse trabalho é desenvolver um interpretador para um subconjunto de uma linguagem de programação conhecida. Para isso foi criada *miniElixir*, uma linguagem de programação de brinquedo baseada em *Elixir* (<https://elixir-lang.org/>). Essa linguagem possui características funcionais (não possui estruturas de repetição), valores atômicos, suporte a funções anônimas com parâmetros, escopo dinâmico, compreensão de listas, entre outras.

A seguir é dado um exemplo de utilização desta. O código a seguir manipula frequências baixas e altas para demonstrar várias, mas não todas, das construções da linguagem.

```
# Read frequencies from the user.
input = fn ->
  khz = read("Entre com uma frequencia em kHz: ")
  cond do
    khz == "" -> []
    :true     -> [int(khz)] ++ input()
  end
end

# Generate stats for frequencies.
stats = fn freqs ->
  head = hd(freqs)
  if head == :error do
    [:nil, :nil]
  else
    [min, max] = stats(tl(freqs))
    [if (min == :nil) || (head < min) do head else min end,
     if (max == :nil) || (head > max) do head else max end]
  end
end

# Print frequencies.
show = fn msg, freqs ->
  unless length(freqs) == 0 do
    puts(msg)
    [min, max] = stats(freqs)
    puts("  Min: " <> str(min))
    puts("  Max: " <> str(max))
  end
end

freqs = input()
low_freqs = for f <- freqs, f < 300 do f end
show("Frequencias baixas: ", low_freqs)

high_freq = freqs -- low_freqs
show("Frequencias altas: ", high_freq)
```

frequencias.mexs



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

2. Instruções

Desenvolver um interpretador que funciona em dois modos. No modo prompt (sem argumentos), onde o interpretador executa expressões dadas pelo usuário em linha de comando até que seja interrompido com Ctrl+D (^D).

```
$ miex
> a = 3
3
> ^D
```

Ou no modo de arquivo, onde o interpretador recebe um programa-fonte na linguagem *miniElixir* como argumento e executa as expressões especificadas por esse programa. Por exemplo, para o programa *frecuencias.mexs* deve-se produzir as seguintes saídas:

```
$ miex frecuencias.mexs
Entre com uma frecuencia em kHz: 250
Entre com uma frecuencia em kHz: 100
Entre com uma frecuencia em kHz: 700
Entre com uma frecuencia em kHz: 450
Entre com uma frecuencia em kHz:
Frecuencias bajas:
  Min: 100
  Max: 250
Frecuencias altas:
  Min: 450
  Max: 700
```

O programa deverá abortar sua execução em caso de qualquer erro léxico, sintático ou semântico, indicando uma mensagem de erro. As mensagens são padronizadas indicando o número da linha com 2 dígitos onde ocorreram:

Tipo de Erro	Mensagem
Léxico	Lexema inválido [<i>lexema</i>]
	Fim de arquivo inesperado
Sintático	Lexema não esperado [<i>lexema</i>]
	Fim de arquivo inesperado
Semântico	Variável não declarada [<i>var</i>]
	Operação inválida

Exemplo de mensagem de erro:

```
$ miex
> puts(x)
01: Variável não declarada [x]
```

3. Características

A linguagem *miniElixir* possui somente expressões. Isso significa que qualquer construção da linguagem produz um valor. A seguir são dados alguns exemplos:

```
> a = 4
4
> if :false do 5 end
:nil
> unless :false do 7 end
```



Laboratório de Linguagens de Programação Prof. Andrei Rimsa Álvares

```
7  
> puts("oi")  
oi  
:ok
```

A linguagem possui os seguintes valores literais: atômicos (começam com dois pontos), inteiros, strings, funções (nativas, definidas pela própria linguagem como `int`, `str`, etc e padrões, definidas pelo usuário), listas (entre colchetes) e tuplas (entre chaves).

```
> :pessoa  
:pessoa  
> 3  
3  
> "abc"  
abc  
> str  
fn<str>  
> fn -> 1 end  
fn<std>  
> [1,"abc",:false]  
[1,abc,:false]  
> {"zero":0,"um":1}  
{zero:0,um:1}
```

Repare que, quando uma construção tem várias expressões associadas, apenas a última expressão é considerada na avaliação desta construção. Por exemplo, a avaliação da expressão `x + y` é usada como resultado da construção do `if`:

```
x = 3  
n = if :true do  
  y = 4  
  x + y  
end  
puts(n);
```

eval.mexs

```
$ miex eval.mexs  
7
```

Os operadores só funcionam com o mesmo tipo, exceto os operadores relacionais. Por exemplo, o operador `+` só funciona com dois inteiros, o operador `<>` só funciona com duas strings, o operador `++` só funciona com duas listas ou duas tuplas, e assim por diante. Se for preciso, pode-se usar conversores de tipos como `int`, `str`.

```
> a = 3 + 4  
7  
> msg = "oi " <> "mundo"  
oi mundo  
> [1,2,3] ++ [4,5,6]  
[1,2,3,4,5,6]  
> puts("a: " <> a)  
01: Operação inválida  
> puts("a: " <> str(a))  
a: 7  
:ok  
> a == msg  
:false
```



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

A linguagem suporta funções anônimas com parâmetros.

```
> soma = fn a, b -> a + b end
fn<std>
> soma(3,4)
7
```

Uma outra característica fundamental é que as construções da linguagem não possuem efeito colateral, ou seja, a linguagem possui transparência referencial conforme encontrado comumente em linguagens funcionais. Contudo, em miniElixir é possível reatribuir (*rebind*) um novo valor a uma variável em um novo escopo, como é o caso das expressões dentro do `if`.

```
a = 3
puts("fora do if] a (antes): " <> str(a))
if :true do
  puts("dentro do if] a (antes): " <> str(a))
  a = 5
  puts("dentro do if] a (depois): " <> str(a))
end
puts("fora do if] a (antes): " <> str(a))
```

retribuicao.mexs

```
$ miex retribuicao.mexs
fora do if] a (antes): 3
dentro do if] a (antes): 3
dentro do if] a (depois): 5
fora do if] a (antes): 3
```

Embora a linguagem Elixir use escopo léxico/estático, a linguagem miniElixir usa escopo dinâmico. O exemplo a seguir descreve esse comportamento.

```
f1 = fn ->
  puts("f1] x (antes): " <> str(x))
  x = 5
  puts("f1] x (depois): " <> str(x))
end

f2 = fn ->
  x = 3
  puts("f2] x (antes): " <> str(x))
  f1()
  puts("f2] x (depois): " <> str(x))
end

f2()
```

escopo.mexs

```
$ miex escopo.mexs
f2] x (antes): 3
f1] x (antes): 3
f1] x (depois): 5
f2] x (depois): 3
```

4. Detalhamento

A linguagem possui comentários em linha, onde são ignorados qualquer sequência de caracteres após o símbolo `#`. Um programa é formado por zero ou mais expressões em sequência, onde uma expressão pode ser:

1) inteiro: sequência de dígitos que formam números inteiros.

```
> 42
42
```

2) string: sequência de caracteres entre aspas duplas.

```
> "abc"
abc
```



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

```
> "abc"
01: Fim de arquivo inesperado
```

- 3) **atômico**: dois pontos, seguido de um caractere ou _ seguido de zero ou mais seqüências de caractere, dígito ou . A linguagem suporta os seguintes atômicos nativos: :nil, :false, :true, :ok, :error.

```
> :error
:error
> :test
:test
```

- 4) **lista**: seqüência de zero ou mais expressões separadas por vírgula entre colchetes. Repare que uma vez definida, essa lista é imutável.

```
> [2, "qwerty", :ok]
[2, "qwerty", :ok]
```

- 5) **tuplas**: seqüência de itens nomeados separados por vírgulas, onde cada item possui uma chave, dois pontos e um valor, entre chaves. As chaves e valores são expressões.

```
> {:nome : "Eu", :idade : 23}
{:nome:Eu, :idade:23}
```

- 6) **if**: executar uma seqüência de expressões (em um novo escopo) se a expressão condicional for verdadeira (ou seja, se diferente de :false e :nil) e executar opcionalmente outras seqüências de expressões (se houverem, também em um outro escopo) caso contrário.

```
> if :true do 5 end
5
> if :false do 5 end
:nil
> if :false do 5 else 6 end
6
> if :nil do 5 else 6 end
6
> if :error do 5 else 6 end
5
> if "" do 5 else 6 end
5
> if "abc" do 5 else 6 end
5
```

- 7) **unless**: executar uma seqüência de expressões (em um novo escopo) se a expressão condicional for falsa, caso contrário avaliar em :nil.

```
> unless :false do 5 end
5
> unless :true do 5 end
:nil
```

- 8) **cond**: seqüência de condicionais com sua expressão (única) relacionada; caso a condicional seja verdadeira, executa-se sua expressão:

```
> x = 7
> cond do
```



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

```
x < 5 -> "menor"  
x < 10 -> "meio"  
:true -> "maior"  
end  
meio
```

- 9) **for**: compreensão de listas, onde listas são geradas dinamicamente de acordo com filtros; o corpo é executado em um novo escopo.

```
> for x <- [1,2,3,4,5], rem(x, 2) == 1 do x*x end  
[1,9,25]
```

- 10) **Função** com suporte a parâmetros:

- a. **padrão**: funções anônimas definidas pelo usuário através da palavra reservada `fn`; a chamada (invocação) é feita passando os argumentos para a função entre parênteses.

```
> plus = fn x -> x + 1 end  
fn<std>  
> plus(2)  
3
```

- b. **nativa**: funções definidas pela própria linguagem.

- `puts`: imprimir expressão na tela com nova linha; retorna `:ok`.

```
> puts(5)  
5  
:ok
```

- `read`: imprimir mensagem na tela sem nova linha e ler uma string do teclado.

```
> read("Entre com seu nome: ")  
Entre com seu nome: eu  
eu
```

- `int`: converter qualquer expressão em inteiro, se não for possível converter para 0.

```
> int("123")  
123  
> int([1,2,3])  
0
```

- `str`: converter qualquer tipo para string.

```
> str("123")  
123  
> str([1,2,3])  
[1,2,3]
```

- `length`: obter o tamanho de listas ou tuplas; caso contrário gerar um erro semântico de operação inválida.

```
> length([1,2,3])  
3  
> length({0:"zero"})  
1
```

**Laboratório de Linguagens de Programação****Prof. Andrei Rimsa Álvares**

```
> length(2)
01: Operação inválida
```

- **hd**: obter o primeiro elemento de uma lista ou a primeiro elemento de uma tupla (chave e valor como lista); se estiverem vazias ou forem de outros tipos deve-se gerar operação inválida.

```
> hd([1,2,3])
1
> hd({0:"zero",1:"um",2:"dois"})
[0,zero]
> hd([])
01: Operação inválida
> hd("abc")
01: Operação inválida
```

- **tl**: obter o restante de uma lista ou uma tupla após o primeiro elemento; se estiverem vazias ou forem de outros tipos deve-se gerar operação inválida.

```
> tl([1,2,3])
[2,3]
> tl({0:"zero",1:"um",2:"dois"})
{1:um,2:dois}
> tl([])
01: Operação inválida
> tl(123)
01: Operação inválida
```

- **at**: obter o valor de uma lista pelo seu índice ou de uma tupla pela sua chave, se não existir ou se não for uma lista ou tupla, deve-se gerar operação inválida.

```
> at([4,5,6], 1)
5
> at([4,5,6], 3)
01: Operação inválida
> at([4,5,6], "abc")
01: Operação inválida
> at({"zero":0,"um":1},"um")
1
> at({"zero":0,"um":1},"dois")
01: Operação inválida
> at({"zero":0,"um":1},2)
01: Operação inválida
> at(123,0)
01: Operação inválida
```

- **rem**: obter o resto de uma divisão, onde tanto o divisor quanto o dividendo são números inteiros; deve-se gerar operação inválida se a divisão for por zero ou se os tipos forem inválidos.

```
> rem(8,3)
2
> rem(8,0)
01: Operação inválida
> rem("abc", 1)
01: Operação inválida
```



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

- 11) Variáveis:** começam com `_` ou letra, seguidas de `_`, letras ou dígitos; variáveis não atribuídas não podem ser lidas; variáveis que começam com `_` são anônimas, ou seja, podem ser atribuídas, mas não podem ser lidas. Além disso, é possível atribuir variáveis com sintaxe de listas:

```
> x
01: Variável não declarada [x]
> x = 3
3
> x
3
> _ = 5
5
> _
01: Operação inválida
> [a,b] = [4,5]
[4,5]
> a
4
> b
5
```

- 12) Operadores binários (dois operandos):**

- a. Operadores aritméticos de **adição (+)**, **subtração (-)**, **multiplicação (*)** e **divisão (/)** operam somente sob números inteiros.

```
> 2 + 3
5
> 4 - 1
3
> 5 * 6
30
> 9 / 2
4
> 2 + "abc"
01: Operação inválida
```

- b. Operadores de **concatenação de strings (<>)**, **concatenação de listas (++)** e **subtração de listas (--)**, para qualquer outro tipo deve-se gerar operação inválida.

```
> "oi " <> "mundo"
oi mundo
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> [1,2,3,2,1] -- [3,2]
[1,1]
> "n: " <> 10
01: Operação inválida
```

- c. Operadores relacionais devem retornar os atômicos `:true` e `:false` dependendo da avaliação
- **Operador de igualdade (==)** e **desigualdade (!=)** operam sob quaisquer tipos; se os tipos forem diferentes retorna-se `:false`.



Laboratório de Linguagens de Programação Prof. Andrei Rimsa Álvares

```
> "abc" == ("a" <> "bc")
:true
> 123 != 456
:true
> "abc" == 123
:false
```

- Operadores de **menor** (<), **maior** (>), **menor igual** (<=) e **maior igual** (>=) comparam apenas números inteiros; se os tipos forem diferentes retorna-se :false.

```
> 4 < 5
:true
> 4 <= "abc"
:false
```

- d. Operadores conectores de conjunção (&&) e disjunção (||) que usam curto circuito. Todos os valores são avaliados em verdadeiro, exceto as constantes :false, :nil e :error que são avaliados em falso. Deve-se usar entre parênteses para a semântica correta. Note que

```
> :nil && []
:nil
> :false && 5
:false
> :nil || []
[]
> :false || 5
5
> (3 < 5) && ("abc" == "abc")
:true
```

- 13)** Operadores unários de negação (!) que funcionam com todos os tipos e de inversão de sinal (-) que só funciona com números inteiros.

```
> !:nil
:true
> !:false
:true
> !5
:false
> !"abc"
:false
> -5
-5
> -"abc"
01: Operação inválida
```

5. Avaliação

O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais a critério do professor. A avaliação é feita exclusivamente executando casos de



Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

```
<code> ::= { <expr> }
<expr> ::= <logic> [ '=' <expr> ]
<logic> ::= <rel> [ ( '&&' | '||' ) <expr> ]
<rel> ::= <subtract> [ ( '<' | '>' | '<=' | '>=' | '==' | '!=' ) <expr> ]
<subtract> ::= <concat> [ '--' <expr> ]
<concat> ::= <arith> [ ( '++' | '<' ) <expr> ]
<arith> ::= <term> [ ( '+' | '-' ) <expr> ]
<term> ::= <prefix> [ ( '*' | '/' ) <expr> ]
<prefix> ::= [ '!' | '-' ] <factor>
<factor> ::= ( '(' <expr> ')' | <rvalue> ) <invoke>
<rvalue> ::= <const> | <list> | <tuple> | <if> | <unless> | <cond> | <for> | <fn> | <native> | <name>
<const> ::= <int> | <string> | <atom>
<list> ::= '[' [ <expr> { ',' <expr> } ] ']'
<tuple> ::= '{' [ <expr> ':' <expr> { ',' <expr> ':' <expr> } ] '}'
<if> ::= if <expr> do <code> [ else <code> ] end
<unless> ::= unless <expr> do <code> end
<cond> ::= cond do { <expr> '->' <expr> } end
<for> ::= for <name> '<' <expr> { ',' <expr> } do <code> end
<fn> ::= fn [ <name> { ',' <name> } ] '->' <code> end
<native> ::= puts | read | int | str | length | hd | tl | at | rem
<invoke> ::= [ '(' [ <expr> { ',' <expr> } ] ')' ]
```

7.3 Interpretador

O diagrama de classes para implementar o modelo do interpretador é dado a seguir.

