PROGRAMAÇÃO FUNCIONAL USANDO ERLANG

Henrique E. Mostaert Rebêlo
Bolsista de Iniciação Científica do Núcleo de Pesquisa do CESBAM

Ricardo Massa F. Lima
Centro de Estudos Superiores Barros Melo - CESBAM

Resumo: Este trabalho representa o primeiro tutorial em língua portuguesa da linguagem funcional Erlang. Não é intenção do tutorial apresentar uma especificação formal da linguagem. Por esta razão, a linguagem é apresentada de forma prática, através de uma descrição textual dos seus construtores, juntamente com exemplos de programas e avaliação de funções na linguagem. O tutorial não aborda aspectos de implementação da linguagem. Na primeira parte do tutorial são discutidos aspectos da programação seqüencial em Erlang, contemplando a definição de módulos, o Erlang Shell, definição de variáveis, tipos de dados e definição de funções. Finalmente, aspectos da programação concorrente em Erlang são brevemente descritos.

Palavras-chave: Linguagens Funcionais, Sistemas Distribuídos e Concorrentes, Telecomunicações.

1. Introdução

Erlang é uma linguagem de programação funcional não-pura para programação de sistemas concorrentes e distribuídos. Erlang incorpora muita das recentes inovações de linguagem de programação, incluindo funções de alta ordem, avaliação estrita, gerenciamento automático de memória (garbage collection), pattern-matching, compreensão de listas, sistema de módulos, tipos de dados primitivos, incluindo inteiros, números reais, suporte para programação seqüencial e concorrente.

Erlang possui um modelo de concorrência baseado em processos. A concorrência é explicita, possibilitando que o usuário controle precisamente as computações realizadas seqüencialmente e aquelas executadas em paralelo. O único meio para troca de dados em

Erlang é a passagem de mensagens de forma assíncrona. O mecanismo para troca de mensagens em Erlang será abordado na seção 4.

Além desta introdução, a seção 2 deste tutorial descreve a programação seqüencial em Erlang. A seção 3 introduz os mecanismos para programação concorrente na linguagem. A seção 4 discute aspectos relativos à programação concorrente em Erlang.

2. Programação Seqüencial

Esta seção descreve a linguagem Erlang, sem considerar os construtores utilizados para expressar concorrência.

Como primeiro exemplo de Erlang, considere o programa a seguir:

```
-module( math1 ).               %(1)
-export( [multiplicação/2]).    %(2)
multiplicação( X, Y ) -> X * Y. %(3)
```

No passo (1) é declarado o módulo denominado math1. No passo (2) tem-se a exportação da função `multiplicação` através da primitiva `export`. Após o nome da função é informado o número de parâmetros que ela possui. No exemplo, a função deve receber 2 argumentos. No passo (3) é definida a cláusula da implementação da função `multiplicação`. A função recebe dois parâmetros e retorna a multiplicação destes valores como resultado.

2.1. Como Compilar um Programa Erlang ?

Esta seção descreve como compilar e executar um simples programa Erlang usando o Erlang Shell. Assumindo que o programa que multiplica dois números tenha sido desenvolvido em um editor de texto qualquer e salvo como arquivo chamado (`math1.erl`), usando o Erlang Shell, siga as instruções abaixo para compilar o programa.

```
Erlang (BEAM) emulator version 5..3  [threads:0]


Eshell V5.3  (abort with ^G)
1>  c(math1).                              %(1)
{ok,math1}                                 %(2)
2>  c(math23).                             %(3)
```

```
./math23.erl:none: no such file or directory %(4)
error                                        %(5)
3>  math1:multiplicação( 5 , 4 ).           %(6)
20                                           %(7)
```

A Tabela 1 apresenta o passo–a–passo das operações realizadas no Erlang Shell.

Tabela 1: Passos das operações realizadas do Erlang Shell

Passo (1)	Compilando o módulo math1 com o comando “c(math1)”
Passo (2)	Resultado bem sucedido da compilação do módulo math1
Passo (3)	Compilando o módulo math1 com o comando “c(math23)”
Passos (4) e (5)	Resultado sem sucesso da compilação do módulo math23: afirma que o arquivo não existe!!!
Passo (6)	Aplica função multiplicação aos parâmetros 5 e 4
Passo (7)	Resultado da avaliação da função multiplicação(5,4)

2.2 Funções Recursivas

A seguir tem-se uma definição recursiva da função fatorial. A definição apresenta duas cláusulas, sendo a primeira o caso base da recursão. As cláusulas são analisadas na ordem em que foram definidas. Portanto, o caso base da recursão deve sempre ser a primeira cláusula. Do contrário, a recursão jamais terminará.

```
-module ( math2 ).
-export ( [factorial/1] ).
factorial( 0 ) -> 1;
factorial( N ) -> N *  factorial( N – 1 ).
```

2.3 Variáveis

Variáveis são usadas para guardar termos em Erlang. Elas necessariamente devem começar com letras maiúsculas. Não é necessário declarar as variáveis. Elas só podem ser instanciadas uma única vez. Para evitar efeitos colaterais o valor de uma variável não pode mudar depois de ter sido atribuído.

Analise algumas operações com variáveis utilizando o Erlang Shell:

```
Erlang (BEAM) emulator version 5..3  [threads:0]


Eshell V5.3  (abort with ^G)
1>  A = 10. // Observe a variável com letra maiúscula
10                                                    2>  X = 53.
53
3>  X = 53. // Variável já foi atribuída, ERRO!!!
badmatch
4>  t = 123. // Variável com letra minúscula, não pode
badmatch
5>  c(math1).
{ok,math1}
6>  Operando1 = 1.
1
7>  Operando2 = 5.
5
8>  math1:multiplicação( Operando1, Operando2 ).
5
9>  c(math2).
{ok,math2}
10>  math2:factorial( Operando2 ).
120
```

2.4 Tipos de Dados

Nesta seção serão descritos os tipos de dados utilizados em Erlang.

Erlang não é uma linguagem tipada, ou seja, ela não suporta a definição de tipos. Apesar desta característica, a linguagem inclui valores com diferentes estruturas e formatos, possibilitando identificá-los como inteiros, reais, tuplas, etc.

2.5 Inteiros

A linguagem de programação Erlang aceita inteiros de até 24 bits de precisão. Isso significa que um número entre $2^{24} - 1$ e $-2^{24} - 1$ deve representar um inteiro. Existem muitas maneiras de escrever constantes inteiras. Alguns exemplos estão ilustrados abaixo.

```
 16777215 --------→ Decimal Positivo
 -16777215 -------→ Decimal Negativo
 2#101 -----------→ Binário
 16#1A -----------→ Hexadecimal
```

Considere a seguir um simples programa que faz uso de números inteiros.

```
-module(maior).
-export( [maxi/2]).
maxi(X, Y) ->
    if
       X > Y -> X;
       Y > X -> Y;
       X == Y -> 'IGUAIS'
    end.
```

O programa `maior` recebe dois números como parâmetro e retorna o maior deles. Caso ambos tenham o mesmo valor, retorna o átomo "iguais".

OBS: Embora esse exemplo de programa trabalhe com números inteiros; nada impede que seja passado números reais para a função `maxi`. Isso acontece porque a linguagem Erlang não é tipada.

A avaliação do programa `maxi` no Erlang Shell é mostrada a seguir:

```
1>  maior:maxi(12,34).
34
2>  maior:maxi(120,34).
120
3>  maior:maxi(120,120).
IGUAIS
```

2.6 Reais

Erlang usa a notação convencional para números. Observe os exemplos abaixo.

```
16.0

-16.22

1.0e3
```

Considere o programa `math3`, que calcula a área de um círculo. Para calcular a área a função recebe um número que representa o raio do círculo.

```
-module(math3).

-export( [area/1]).

area( {circle, Radius} ) ->

    math:pi() * Radius * Radius;
```

A avaliação do programa `math3` no Erlang Shell é mostrada a seguir:

```
1>  math3:area( {circle, 2.5} ).

19.6350

2>  math3:area( {circle, 2} ).

12.5664
```

2.7 Átomos

Um átomo é um nome constante. Dois átomos são equivalentes quando eles são, caractere por caractere, idênticos. Átomos começam por letra minúscula, podem conter espaços em branco, ou um átomo é colocado em aspas simples (‘’), quando contém várias palavras separadas por espaços em branco.

```
M1 = asdf1234

M2 = ‘podem conter espaços’

M3 =‘Ou qualquer caractere dentro de aspas simples maiúsculo ou
minúsculo’
```

Considere a seguir um simples programa que faz uso de átomos:

```
-module(dia).
-export( [classify_day/1]).
classify_day( sabado ) -> weekEnd;
classify_day( domingo ) -> weekEnd;
classify_day(_)-> weekDay.
```

O programa dia acima recebe um átomo como parâmetro e retorna a classificação do dia de acordo com o átomo de entrada.

A avaliação do programa dia no Erlang Shell é mostrada a seguir:

```
1>  dia:classify_day(sabado).
weekEnd
2>  dia:classify_day(sexta).
weekDay
```

2.8 Tuplas

Tupla é um tipo de dado particular em Erlang que é semelhante as “structures” em linguagem C, ou “record” em Pascal.

Uma tupla pode conter valores de tipos distintos, como: lista, átomos, outra tupla, números reais ou inteiros e etc. Alguns exemplos de tuplas são apresentados a seguir:

```
{1,2,3} -------> números inteiros
{fred,20.0,3} -------> átomo, número real, e número inteiro
{15,’fifteen’} -------> número inteiro e um átomo
{3,{a,ab,c}} ---------> número inteiro e outra tupla
{3,[a,b,c]} ---------> número inteiro e uma lista
```

Considere a seguir o programa `calculadora` que faz uso de tuplas:

```
-module(calculadora).
-export([result/1]).
result({soma, A, B}) -> {'resultado soma',A + B};           %(1)
result({subtracao, A, B}) -> {'resultado subtracao',A - B}; %(2)
```

```
result({multiplicacao, A, B}) ->                          %(3)

  {'resultado multiplicacao',A * B};

result({divisao,A, B}) -> {'resultado divisao',A / B};    %(4)

result({_, A, B}) -> {'Operacao invalida!!'}.             %(5)
```

As cláusula (1), (2), (3), (4), são similares, ou seja, a função `result` recebe o nome da operação em forma de átomo, recebe duas variáveis *A* (primeiro operando) e *B* (segundo operando) e, por fim, retorna uma tupla com o nome e o resultado da operação. Na cláusula (5), se a função `result` receber uma operação diferente das especificadas nas cláusulas, ela retorna a tupla com a mensagem de "Operação inválida!!".

A avaliação do programa calculadora no Erlang Shell é mostrada a seguir:

```
1>  calculadora:result({soma,20,10}).
{'resulado soma',30}
2>  calculadora:result( {elevado,2,34}).
{'Operação inválida'}
```

2.9 Listas

Listas em Erlang representam uma coleção de elementos de um tipo particular, por exemplo: podem ser listas, tuplas, inteiros, átomos e etc. Esta seção descreve funções de processamento de listas mais comuns, entre elas: `membro`, `concat` e `deleta_todos`.

2.9.1 Função membro

A função `membro` retorna verdadeiro se o elemento procurado for encontrado na lista, e falso caso contrário.

Considere o seguinte programa da função `membro`:

```
-module(lista_membro).
-export( [membro/2] ).
membro( X, [X|_])-> true;       %(1)
membro(X,[_|T]) -> membro(X,T); %(2)
membro(X,[]) -> false.          %(3)
```

A cláusula (1) verifica se X é o elemento na cabeça da lista. A cláusula (2) verifica se o elemento está na cauda da lista. A cláusula (3) verifica se a lista é vazia e, conseqüentemente, retorna falso para membro.

A avaliação do programa membro no Erlang Shell é mostrada a seguir:

```
1>  lista_membro:membro(x,[s,d,f,g,h,x]).
true
2>  lista_membro:membro(x,[]).
false
```

2.9.2 Função concat

A função concat recebe duas listas e retorna uma lista.

Considere o seguinte programa da função concat:

```
-module(lista_concat).
-export( [concat/2] ).
concat([H|L1],L2) ->[H|concat( L1, L2)]; %(1)
concat([],L) ->L. %(2)
```

A cláusula (1) concatena L2 na cauda da lista. A cláusula (2) concatena uma lista L a uma lista vazia, resultando na própria lista L.

A avaliação da função concat no Erlang Shell é mostrada a seguir:

```
1>  lista_concat:concat([a],[e,r,t,2,3]).
[a,e,r,t,2,3]
2>  lista_concat:concat([x],[]).
[x]
```

2.9.3 Função deleta_todos

A função `deleta_todos`, deleta todas as ocorrências de um elemento particular na lista.

Considere o seguinte programa da função `deleta_todos`:

```
-module(lista_deleta).
-export( [deleta_todos/2] ).
deleta_todos(X, [X|T]) -> deleta_todos(X,T);     %(1)
deleta_todos(X, [Y|T]) -> [Y|deleta_todos(X,T)]; %(2)
deleta_todos(_,[]) -> [].                        %(3)
```

A cláusula (1) representa a remoção do elemento procurado (X) da cabeça da lista e a chamada recursiva da função aplicada aos elementos restantes na cauda da lista. Na cláusula (2) oelemento na cabeça da lista não é o procurado, sendo, portanto, mantido na lista, enquanto a função é aplicada recursivamente aos elementos restantes na cauda da lista para continuar a busca por elementos a serem removidos. A cláusula (3) representa o fim da busca, pois a lista está vazia.

A avaliação do programa lista_deleta no Erlang Shell é mostrada a seguir:

```
2>  lista_deleta:deleta_todos(a,[1,2,a,3,a,5]).
[1,2,3,5]
3>  lista_deleta:deleta_todos(x,[a,d,e,3,4,5,6]).
[a,d,e,3,4,5,6]
```

3 Programação Concorrente

Um programa concorrente consiste em uma coleção de processos e objetos compartilhados. Cada processo é definido por um programa seqüencial.

Um processo cooperativo é aquele que pode afetar ou ser afetado por outros processos no sistema. Para o paralelismo temos dois tipos, a saber: o paralelismo real (um processador físico para cada processo); e o paralelismo aparente (um processador físico alternando entre os processos).

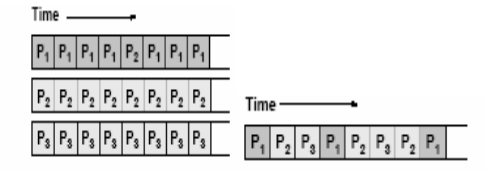

Figura 1: (a) Exemplo de paralelismo real. (b) Exemplo de paralelismo aparente

Programas que modelam ou interagem com o mundo real precisam refletir a concorrência que é observada neste mundo. Atividades reais são concorrentes. Portanto, escrever programas para interagir com o mundo real pode ser um simples problema de identificar a concorrência no problema.

Infelizmente, programação concorrente tem adquirido uma reputação de ser “difícil” e algo a ser evitado. Os autores da linguagem Erlang acreditam que esse efeito colateral é derivado do uso de linguagens convencionais para programação concorrente, como C, Java e C++.

4 Programação Concorrente em Erlang

A criação de processos, bem como a comunicação entre eles, são operações explicitamente definidas pelo programador Erlang.

4.1 Criação de Processos

Um processo é uma unidade separada de computação que existe concorrentemente com outros processos no sistema. Processos em Erlang são extremamente leves. Criar um processo em Erlang é mais rápido do que em Java ou C++. Isto acontece porque a concorrência é definida na linguagem e não tem nenhuma relação com o sistema operacional.

A função `spawn/3` cria e inicia a execução de um novo processo.

Ao invés de avaliar a função e retornar o resultado da avaliação, como faz a função `apply`, `spawn/3` cria um novo processo concorrente para avaliar a função e retorna o `Pid` (identificador de processo) do processo criado. `Pids` são usados para todas as formas de

comunicações entre os processos. A chamada para `spawn/3` retorna imediatamente após a criação do processo, sem esperar pela avaliação da função. Uma declaração para criação de processo deve respeitar o seguinte formato:

```
Pid = spawn(Modulo, Nome da Função, Lista de Argumentos)
```

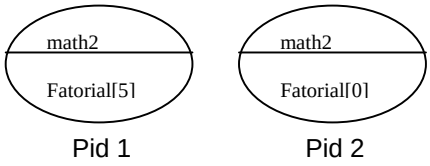

Figura 1: Exemplos da criação de dois processos

Observe como são criados os dois processos apresentados na Figura 3 usando o Erlang Shell:

```
1> Pid1 = spawn( math2, factorial, [5] ).  %(1)
 <0.46.0>
2> Pid2 = spawn( math2, factorial, [0] ).  %(2)
 <0.51.0>
3> b().
 Pid1 = <0.46.0>
 Pid2 = <0.51.0>
 ok
```

No passo (1) um processo é instanciado na variável *Pid1* usando a função `spawn`. Essa função recebe como parâmetro o módulo `math2`, a função `factorial` e a lista `[5]`. Um procedimento similar ocorre no passo (2). As variáveis *Pid1* e *Pid2* recebem o identificador dos processos criados. Para visualizar os processos criados no Erlang basta utilizar a função `b( )`.

4.2 Comunicação entre processos

Em Erlang, a única forma de comunicação entre os processos é por passagem de mensagem. A mensagem é enviada para outro processo pela primitiva "`!`" (send).

```
PID ! Mensagem
```

`Pid` é o identificador de processo na qual a mensagem é enviada. A mensagem pode ser qualquer termo válido em Erlang.

Em Erlang, a passagem de mensagens é gerenciada por uma máquina virtual, sem a interrupção do sistema operacional.

4.3 Enviando mensagens para um processo específico

Para enviar uma mensagem para um processo específico deve-se primeiramente conhecer o `Pid`, ou seja, o identificador do processo ao qual a mensagem será enviada.

```
Pid ! { Pid2, 'hello!'}
```

Considere o seguinte exemplo no Erlang Shell:

```
1> Pid1 = spawn( math2, factorial, [5] ).  %(1)
 <0.46.0>
2> Pid2 = spawn( math2, factorial, [0] ).  %(2)
 <0.51.0>
3> Pid1 ! {Pid2,'Recebe oi de Pid1'}.
{<0.51.0>,'Recebe oi de Pid1'}
4> Pid2 ! {Pid1,'Recebe oi tambem'}.
{<0.46.0>,'Recebe oi tambem'}
```

Em Erlang os processos se comunicam pelo mecanismo de passagem de mensagem assíncrona, ou seja, o processo envia a mensagem e não espera que o outro processo a receba. Caso o destinatário da mensagem não esteja mais ativo, nenhuma mensagem de erro será apresentada.

5. Conclusão

Este tutorial apresentou os principais aspectos práticos de programação seqüencial e concorrente usando a linguagem Erlang. Os construtores da linguagem foram descritos de maneira informal. As descrições textuais dos construtores foram acompanhadas de exemplos de programas, bem como de avaliação de funções utilizando o Erlang Shell.

O tutorial foi dividido em duas partes. A primeira parte detalhou a programação seqüencial em Erlang. Após uma breve descrição sobre as principais características da linguagem e do Erlang shell, foram discutidos os conceitos de módulos, funções e variáveis em Erlang, seguido de um estudo sobre os tipos de dados da linguagem.

A segunda parte do tutorial aborda a programação concorrente em Erlang. Além de apresentar o modelo de comunicação da linguagem, foram descritos os construtores para criação e comunicação entre processos.

O texto apresentado deve ser entendido como uma introdução à linguagem. As referências [2] e [5] são leituras obrigatórias para interessados em aprofundar seus conhecimentos sobre a linguagem de programação funcional Erlang.

Referências Bibliográficas

[1] R. W. Sebesta. Conceitos de Linguagens de Programação. Porto Alegre -4 e.d BooKman, 2000.

[2] J. Armstrong, R. Virding, C. Wkstrom, M. Williams. Concurrent Programming in Erlang, Prentince-Hall, Englewood Cliffsm N.J., 1996.

[3] M. Castro. Erlang in Real Time. Department of Computer Science RMIT Austrália, 1998-2001.

[4] Philip Wadler. Lazy Vs Strict, University of Glasgow, Scotland.

[5] M. B. Feldman. Software and Data Structures. Addilson-Wesley, chapter 15, Introduction to Concurrent Programming, 1996.