Erlang – núcleo funcional

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Núcleo funcional

- tipos de dados;
- variáveis;
- pattern matching;
- funções;
- guardas;
- sequências;
- condicionais;
- tail calls e máquinas de estados;
- módulos;
- funções de ordem superior;
- compreensões;
- binários e bitstreams;

Tipos de dados simples

- Números:
  - podem ser inteiros ou floating point;

```
123, 3.1415, -1.2e3
2#11101001, 16#10FE          % base 2 e 16; bases entre 2 e 16
$A, $\n                          % valor ascii de A e de \n
```

- Átomos:
  - constantes com nome;
  - começam por minúsculas; plicas para usar espaços;

```
true, false          % booleanos, com operacoes not, and, or, xor
red, green, blue, request, reply, 'hello world\n'
```

- Pids:
  - guardam identificadores de processos;
  - para usar como destino de mensagens;
- Referências:
  - para guardar referências únicas em todo o sistema;

Tipos de dados compostos

- Túplos: número fixo de itens;

```
{}
{add, 123}
```

- Listas: número variável de itens; polimórficas;

```
[]
[1, 2, 3]
[1, 1.1, abc]
[[1,2], abc, [1, a]]
[1 | [2, 3]]   % = [1,2,3]
"ABCD"         % = [65,66,67,68]; nao existe o tipo string
```

- Mapas: associações de chave para valor

```
#{}
#{name => "Alice", age => 25, friends => ["Bob", "Carol"]}
#{[2,4] => {circle, 3}, [9, 7] => {rectangle, 3, 5}}
```

Variáveis

- Começam por maiúsculas;

```
X = 2
Y = {circle, 3}
```

- Não são variáveis, mas associações de nomes com valores;
- Depois de ser criada a associação, não pode mudar (single assignment);

```
L1 = [1, 2, 3]
L2 = [0 | L1]
M1 = #{name => "Alice", age => 25, friends => ["Bob", "Carol"]}
M2 = M1#{age := 26}                 % update de chave presente
M3 = M2#{country => "Portugal"}     % update ou acrescenta nova chave
```

Pattern matching

- Usado para associar um valor a um padrão;
  - em atribuições a variáveis;
  - associação de argumentos com parâmetros de funções;
  - na expressão “case”;
  - na entrega de mensagens na mailbox;
- Exemplos:

```
A = 12
{X, Y} = {2, 3}
{C, D} = {[1,2,3], {hello, world}}      % C=[1,2,3]  D={hello, world}
[H|T] = [1,2,3]                         % H=1  T=[2,3]
{ok, Res} = {ok, 23}                    % Res=23
{_, Res} = {ok, 23}                     % Res=23  _ match e ignora
[A,A,C] = [1,1,3]                       % A=1  C=3
[A,A,C] = [1,2,3]                       % Falha
{ok, Res} = {red, 23}                   % Falha
[A,B,C,D] = [1,2,3]                     % Falha
Color = #{ r => 0.2, g => 0.5, b => 0.7}
#{r := Red, b := Blue} = Color          % Red = 0.2, Blue = 0.7
```

Funções

- Podem ser definidas por cláusulas, usando pattern matching e recursividade;

```
factorial(0) ->
  1;
factorial(N) ->
  N * factorial(N-1).
```

- Cada cláusula é separada por ';';
- É usada a primeira cláusula que faz matching;
- No exemplo acima a ordem é importante; no exemplo abaixo não.

```
len([_|T]) ->
  1 + len(T);
len([]) ->
  0.
```

Guardas

- Podem ser usadas em funções, condicionais, recepção de mensagens;

```
factorial(N) when N > 0 ->
  N * factorial(N-1);
factorial(N) when N == 0 ->
  1.
```

- Guarda é uma sequência de testes separados por vírgulas; têm que ser todos verdadeiros para a guarda ter sucesso;
- Uma sequência de guardas separadas por ";" tem sucesso se uma das guardas tiver sucesso;
- Testes podem conter constantes, expressões aritméticas, comparações, fazer uso de testes pré-definidos como:

```
is_atom/1, is_binary/1, is_constant/1, is_float/1, is_function/1,
is_function/2, is_integer/1, is_list/1, is_number/1, is_pid/1,
is_port/1, is_reference/1, is_tuple/1, is_record/2, is_record/3
```

  e funções pre-definidas como:

```
abs(Number), element(N, Tuple), float(Term), hd(List),
length(List), node(), node(Pid|Ref|Port), round(Number),
self(), size(Tuple|Binary), tl(List), trunc(Number)
```

Sequências

- O corpo de uma cláusula é uma sequência de expressões;
- Expressões são separadas por vírgula;
- Expressões são avaliadas sequencialmente
  - relevante se tiverem efeitos laterais como enviar mensagem;

```
factorial(N) when N > 0 ->
  N1 = N - 1,
  F1 = factorial(N1),
  N * F1;
factorial(0) ->
  1.
```

Expressões condicionais – if

```
if
  Guard1 -> Seq1;
  Guard2 -> Seq2;
  ...
end
```

- Se nenhuma guarda tiver sucesso o processo aborta com erro;
- Para 'else' pode ser usado o idioma:

```
if
  ...
  true -> ...
end
```

- Exemplo:

```
factorial(N) ->
  if
    N == 0 -> 1;
    N >  0 -> N * factorial(N-1)
  end.
```

- Este factorial não está definido para números negativos.

Expressões condicionais – case

```
case Expr of
  Pattern1 [when Guard1] -> Seq1;
  ...
  PatternN [when GuardN] -> SeqN
end
```

- Exemplo:

```
factorial(N) ->
  case N of
    0 -> 1;
    N when N > 0 -> N * factorial(N-1)
  end.
```

Tail recursion

- Em Erlang, ciclos são obtidos com recursividade;
- É importante o caso especial de tail-calls, em que a invocação recursiva aparece sempre no fim das expressões;
- Permite evitar o crescimento do stack e executar em memória constante.
- Importante para processos que correm indefinidamente.
- Exemplo; comparar:

```
len([_|T]) ->
  1 + len(T);
len([]) ->
  0.
```

- com (usando overloading do nome len para 1 e 2 parâmetros):

```
len(L) ->
  len(L,0).

len([_|T], N) ->
  len(T, 1 + N);
len([], N) ->
  N.
```

Máquinas de estados com tail calls

- O caso de tail recursion é um caso especial;
- Em geral, várias funções podem invocar-se mutuamente;
- Se usarem tail calls, executam em memória constante;
- Tail calls permitem codificar muito facilmente uma máquina de estados, com cada função a representar um estado.

```
acquired(X) ->
  ...
  released(Y).

released(X) ->
  ...
  acquiring(Y).

acquiring(X) ->
  receive
    ...
      acquiring(Y);
    ...
      acquired(Z)
  end.
```

Módulos

- Programa é organizado em módulos, que definem funções;
- Nomes dos módulos e funções são átomos;
- É necessário exportar as funções usadas fora do módulo;
- Exemplo:
  - módulo demo, exporta função factorial com 1 parâmetro;
  - no início do ficheiro de nome demo.erl:

```
-module(demo).
-export([factorial/1]).
```

  - uso de factorial noutro módulo:

```
demo:factorial(3)
```

  - ou com um import no início do ficheiro:

```
-import(demo, [factorial/1]).

factorial(3)
```

Funções de ordem superior

- É possível criar uma função anónima, com fun.
- Exemplo: adder devolve função que adiciona N ao argumento:

```
adder(N) ->
  fun (X) -> X + N end.
```

- A closure faz referência à variável N do contexto onde definida;
- A função devolvida pode ser invocada com a sintaxe usual:

```
F = adder(2),
F(3).              % = 3+2
```

- Uma função global pode ser passada a um contexto que requeira uma closure com:

```
fun name_func/arity
fun module:name_func/arity
```

- Exemplo:

```
lists:filter(fun par/1, lists:seq(1,10))
```

Listas e compreensões

- O módulo `lists` possui funções como: `map`, `foldl`, `zip`, `filter`, ...

```
L = lists:seq(1, 100),
Par = fun(X) ->
        if X rem 2 == 0 -> true;
           true -> false
        end
      end,
Pares = lists:filter(Par, L),
lists:map(fun(X) -> 3 * X end, Pares).
```

- Existem também sintaxe especial para listas por compreensão:

```
[ 3 * X || X <- lists:seq(1, 100), X rem 2 == 0]
```

Binários

- Bloco de memória não tipado (sequência de bytes);
- Usado para processar ficheiros, streams;
- Construído com bit-sintaxe;
- Processado com pattern-matching;
- Exemplo: função que recebe e faz parsing de um datagrama IPv4 via pattern-matching:

```
parse_IP_packet(Dgram) ->
  DgramSize = size(Dgram),
  case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32, DestIP:32, RestDgram/bytes>>
    when HLen >= 5, 4*HLen =< DgramSize ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/bytes,Data/bytes>> = RestDgram,
      ...
    end.
```

Bitstreams

- Bitstream: sequência de bits;
- Generalização: binários não múltiplos de 8 bits;
- Processamento eficiente de streams não orientados ao byte;
- UU-encoding numa linha; com compreensões sobre bitstreams:

```
uuencode(BitStr) -> << (X+32):8 || <<X:6>> <= BitStr >>.

uudecode(Text) -> << (X-32):6 || <<X:8>> <= Text >>.
```

- Existem compreensões sobre listas ou bitstreams, que criam listas ou bitstreams (todas as 4 combinações);