Recursividade em Erlang

Paulo Ferreira
paf@dei.isep.ipp.pt

Fevereiro de 2006

Guards

Guards 1

Considerando:

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

E factorial (-1) ?

Podemos usar um *guard*:

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

Guards 2

- A palavra reservada `when` introduz um *guard*
- Todas as variáveis usadas no *guard* devem estar instanciadas
- As cláusulas protegidas com *guards* podem ser reordenadas:

```
% versão 1
factorial(1)->1;
factorial(N) when N>1 -> N*factorial(N-1).
% versão 2
factorial(N) when N>1 -> N*factorial(N-1);
factorial(1)->1.
```

Guards 3

Se não usarmos *Guards* a definição fica errada se trocarmos a ordem das cláusulas:

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

Porque é que isto não funciona ?

Funções de guard

`number(X)`	– X é um número
`integer(X)`	– X é um inteiro
`float(X)`	– X é um float
`atom(X)`	– X é um átomo
`tuple(X)`	– X é um tuplo
`list(X)`	– X é uma lista

Guards aritméticos

`length(X)==3` – X é uma lista de comprimento 3
`size(X)==2` – X é um tuplo de tamanho 2
`X > Y + Z` – X é maior do que Y + Z
`X == Y` – X é igual a Y
`X =:= Y` – X é exactamente igual a Y
(`1==1.0` é verdadeiro, mas `1=:=1.0` é falso)

Cuidado: nem todas as funções podem ser usadas em *guards*, ler o manual para mais esclarecimentos! Funções escritas pelo utilizador não podem nunca ser usadas em *guards*.

Ordenação

Lista >
Tuplo >
Pid >
Port >
Reference >
Àtomo >
Número

- Os tuplos são ordenados primeiro pelo seu tamanho e depois pelos seus elementos.
- As listas são ordenadas pelas cabeças primeiro.

Recursividade

Razões a favor

- Simples
- Elegante
- Fácil de provar a correcção matematicamente (indução)
- Programas pequenos
- Código mais simples

Razões contra

- Não se usa porque dá *stack overflow*!
- Conforme a linguagem
- Conforme o compilador
- Conforme as opções do compilador
- Conforme a memória disponível

Iterações

Se o valor de uma variável não pode mudar depois de atribuído, como podemos fazer iterações ?

```
For i= 1 to N do ...
```

Resposta: usando recursividade

Ciclos

- Ciclos de N até 1:

```
for(0)-> done;
for(N)-> <fazer uma passagem>,
                  for(N-1).
```

Ciclos

- Ciclos de N até 1:

```
for(0)-> done;
for(N)-> <fazer uma passagem>,
                  for(N-1).
```

- Ciclos de 1 até N:

```
for(N)-> for(1,N).

for(N,N)->   <fazer uma passagem>;
for(I,N)->   <fazer uma passagem>,
                    for(I+1,N).
```

- O padrão da recursividade é o mesmo nos dois casos

Mais usos 1

Recursividade em listas

- A recursividade em listas é muito comum:

```
average(X)-> sum(X) / len(X).


sum([H|T])-> H + sum(T);
sum([]) ->0.


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

Mais usos 2

Outros dois padrões comuns de recursividade:

```
double([H|T])-> [2*H|double(T)];
double([])->[].


member(H,[H|_])-> true;
member(H,[_|T])-> member(H,T);
member(_,[])-> false.
```

Acumuladores

Recursividade usando acumuladores:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

- Esta definição é executada num espaço constante e atravessa a lista apenas uma vez
- A variável `Sum` desempenha o papel de *acumulador*

Explicação

Mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

- Temos duas funções diferentes!
- A primeira apenas chama a segunda com o acumulador inicializado a zero.
- A segunda *faz o trabalho todo*.
- Termina quando a lista estiver vazia.

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

`sum([1,2,3])` passa a `sum([1,2,3],0)`[1]

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

`sum([1,2,3])`	passa a `sum([1,2,3],0)`[1]
`sum([1,2,3],0)`	passa a `sum([2,3],1)`

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

```
sum([1,2,3])     passa a sum([1,2,3],0)[1]
sum([1,2,3],0)   passa a sum([2,3],1)
sum([2,3],1)     passa a sum([3],3)
```

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

`sum([1,2,3])`	passa a `sum([1,2,3],0)`[1]
`sum([1,2,3],0)`	passa a `sum([2,3],1)`
`sum([2,3],1)`	passa a `sum([3],3)`
`sum([3],3)`	passa a `sum([],6)`

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

`sum([1,2,3])`	passa a `sum([1,2,3],0)`[1]
`sum([1,2,3],0)`	passa a `sum([2,3],1)`
`sum([2,3],1)`	passa a `sum([3],3)`
`sum([3],3)`	passa a `sum([],6)`
`sum([],6)`	passa a 6

Devagarinho

Ainda mais uma vez:

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

sum([H|T],Sum)-> sum(T,Sum+H);
sum([],Sum)-> Sum.
```

`sum([1,2,3])`	passa a `sum([1,2,3],0)`[1]
`sum([1,2,3],0)`	passa a `sum([2,3],1)`
`sum([2,3],1)`	passa a `sum([3],3)`
`sum([3],3)`	passa a `sum([],6)`
`sum([],6)`	passa a 6

[1] Mudança de função!

Versão Normal

Normalmente:

```
sum([])->0;
sum([H|T])->H+sum([T]).
```

`sum([1,2,3])` dá `1+sum([2,3])`

Versão Normal

Normalmente:

```
sum([])->0;
sum([H|T])->H+sum([T]).
```

```
sum([1,2,3])    dá 1+sum([2,3])
1+sum([2,3])    dá 1+2+sum([3])
```

Versão Normal

Normalmente:

```
sum([])->0;
sum([H|T])->H+sum([T]).
```

```
sum([1,2,3])    dá 1+sum([2,3])
1+sum([2,3])    dá 1+2+sum([3])
1+2+sum([3])    dá 1+2+3+sum([])
```

Versão Normal

Normalmente:

```
sum([])->0;
sum([H|T])->H+sum([T]).
```

```
sum([1,2,3])    dá 1+sum([2,3])
1+sum([2,3])    dá 1+2+sum([3])
1+2+sum([3])    dá 1+2+3+sum([])
1+2+3+sum([])   dá 1+2+3+0
```

Exemplos

```
length(L)-> length(L,0).
length([H|T],L)-> length(T,L+1);
length([],L)->L.

average(X)-> average(X,0,0).
average([H|T],Length,Sum)->
              average(T,Length+1,Sum+H);
average([],Length,Sum) ->Sum/Length.
```

`length([1,2,3])` dá `lenght([1,2,3],0)`

Exemplos

```
length(L)-> length(L,0).
length([H|T],L)-> length(T,L+1);
length([],L)->L.

average(X)-> average(X,0,0).
average([H|T],Length,Sum)->
              average(T,Length+1,Sum+H);
average([],Length,Sum) ->Sum/Length.
```

```
length([1,2,3])     dá lenght([1,2,3],0)
length([1,2,3],0)   dá length([2,3],1)
```

Exemplos

```
length(L)-> length(L,0).
length([H|T],L)-> length(T,L+1);
length([],L)->L.

average(X)-> average(X,0,0).
average([H|T],Length,Sum)->
            average(T,Length+1,Sum+H);
average([],Length,Sum) ->Sum/Length.
```

```
length([1,2,3])     dá lenght([1,2,3],0)
length([1,2,3],0)   dá length([2,3],1)
length([2,3],1)     dá length([3],2)
```

Exemplos

```
length(L)-> length(L,0).
length([H|T],L)-> length(T,L+1);
length([],L)->L.

average(X)-> average(X,0,0).
average([H|T],Length,Sum)->
            average(T,Length+1,Sum+H);
average([],Length,Sum) ->Sum/Length.
```

```
length([1,2,3])     dá lenght([1,2,3],0)
length([1,2,3],0)   dá length([2,3],1)
length([2,3],1)     dá length([3],2)
length([3],2)       dá length([],3)
```

Exemplos

```
length(L)-> length(L,0).
length([H|T],L)-> length(T,L+1);
length([],L)->L.

average(X)-> average(X,0,0).
average([H|T],Length,Sum)->
              average(T,Length+1,Sum+H);
average([],Length,Sum) ->Sum/Length.
```

```
length([1,2,3])     dá lenght([1,2,3],0)
length([1,2,3],0)   dá length([2,3],1)
length([2,3],1)     dá length([3],2)
length([3],2)       dá length([],3)
lenght([],3)        dá 3
```

Funções Tail recursive

```
inverter(L)-> inverter(L,[]).
inverter([],X)-> X;
inverter([H|T],X)-> inverter(T,[H|X]).
```

- Esta definição termina com uma chamada a si própria sem cálculos pendentes

```
adicionar([],X)-> X;
adicionar([H|T],X)->
                    [H|adicionar(T,X)].
```

- A última expressão avaliada não é uma chamada à função nem uma constante!

Importância

- Importantes porque permitem ter funções recursivas que correm num espaço de memória constante
- Importante em termos de memória e velocidade do programa
- Permitem fazer *Last Call Optimization*
- Se usarmos acumuladores temos normalmente tail recursive functions
- Os servidores devem ser escritos como funções recursivas (ver matéria mais à frente)

Outros

Case

- Sintaxe:

```
case f(X) of
            {ok, Result}-> g( Result);
            {error, Reason} ->
          io:format(" Error in f/1: ~p~n", [Reason]),
                    h(Reason);
            _Other -> ignore
 end
```

- Um dos ramos do case deve ser seguido, senão vai ocorrer um erro de *run-time*.
- Um programa bem escrito não necessita de case

If

- Sintaxe:

```
if          X> 5 -> f( X);
              X< 15 -> g( X);
              true ->
        io:format("X outside interval [6..14]: ~p~n", [X])
end
```

- Podem-se usar as mesmas expressões que nos *guards* Um dos ramos do `if` deve ser seguido, senão vai ocorrer um erro de *run-time*!
- Um programa bem escrito não necessita de `if`

Interacção

- Como usar o Erlang
- `erl` – em unix

```
1>ls().
2>pwd().
3>cd('D:/erlang/work').
4>c(demo).
5>demo:dobro(2).
6>halt().
```

- O ponto no final de um comando é importante
- A barra de directório no Windows, escreve-se como em Unix