Introdução ao Erlang

Paulo Ferreira
paf@dei.isep.ipp.pt

Fevereiro de 2006

Apresentação

História - Ericsson

- 1982-1986 Investigação de linguagens para telecomunicações
  - Resultado: Fazer uma.
- 1987 Primeiras Experiências com Erlang
- 1991 Uma implementação rápida
- 1993 Distribuição. Fundação de Erlang Systems.
- 1998 Erlang em «*Open Source*»

Aplicações

- Distribuídas
- Concorrentes
- Tolerantes a falhas
- *Non-Stop*
- «*Soft Real-Time*»
- Actualização do código em «*Run-Time*»
- Gestão de memória automática
- Plataformas Win32 + Unix + VxWorks (comercial)

Onde obter

- `http://www.erlang.se` - Versão comercial
- `http://www.erlang.org` - Versão *Open Source*

Usos

- Centrais Telefónicas
- Serviços Móveis
- Produtos de banda larga e móveis
- `http://www.bluetail.com` – Mail Robustifier, Web Prioritizer Bluetail (25 pessoas) Jan1999, vendida em Ago 2000 à AlteonWebSystems por $152.000.000 US, vendida à Nortel
- `http://yaws.hyber.org` – «*Yet Another Web Server*»
- `http://www.wings3d.com/` – Modelador Geométrico
- `http://wagerlabs.com/` – Servidor de Poker em Erlang
- `http://eddie.sourceforge.net/` – Servidor web robusto e escalável
- `http://www.erlang.se/euc/` – Erlang User Conference
- `http://ll2.ai.mit.edu/talks/armstrong.pdf` – Lightweight Languages

OTP

Além do Erlang a distribuição inclui o OTP – *Open Telecom Platform* – um conjunto de módulos que ajudam na implementação de soluções típicas

- Bibliotecas e interfaces existentes (exemplos)
- Compilador, kernel e biblioteca standard
- Handler para eventos e alarmes, SNMP, medidor de avaliação.
- ASN1, interfaces de baixo nível para C e Java, Servidor Web e cliente Ftp.
- Chamada a objectos COM em windows, SSL e criptografia.
- Base de dados Mnesia (tempo real) e interface ODBC.
- Serviços CORBA e compilador IDL.
- Ferramentas de debugging e monitorização.

Prática

- Grande número de processos por nó – (4000 no AXD301).
- *Soft RealTime* – Custo das chamadas.
- Sistemas distribuídos – A versão de 40 Gbits do AXD301 tem 72 processadores.
- Comunicação com hardware – Interfaces standard para C e device drivers,
- Sistemas grandes: AXD301 Rel3.2 tem 1M linhas de Erlang.
- Funcionalidade complexa: AXD301 protoc. ITU + ATM Forum.
- Operação contínua: Mobility Server (1994) com 400 produtos.
- Manutenção do software: Upgrades sem parar o sistema.
- Requisitos de qualidade e fiabilidade: GPRS 99.995% de disponibilidade.
- Tolerância a falhas de hardware e software: falhando um dos processadores do GPRS apenas baixa a capacidade do sistema.

Bibliografia

Livros

- *Concurrent programming in ERLANG (2nd ed.)* – Robert Virding; Claes Wikström; Mike Williams e Joe Armstrong
  - Prentice Hall International, 1996, 351 páginas, ISBN:0-13-508301-X
  - O Erlang já evoluiu desde então, e o livro está desactualizado
  - A primeira parte está em: `http://www.erlang.org/download/erlang-book-part1.pdf`

Livros

- *Concurrent programming in ERLANG (2nd ed.)* – Robert Virding; Claes Wikström; Mike Williams e Joe Armstrong
  - Prentice Hall International, 1996, 351 páginas, ISBN:0-13-508301-X
  - O Erlang já evoluiu desde então, e o livro está desactualizado
  - A primeira parte está em: `http://www.erlang.org/download/erlang-book-part1.pdf`
- *Erlang programmation* – Mikaël Rémond, Eyrolles
  - Eyrolles, 2003, 360 páginas, ISBN 2-212-11079-0
  - Em francês...

Outros

- *Making reliable distributed systems in the presence of software errors* – Joe Armstrong
  - Tese de Doutoramento
  - Legível e acessível mas certas coisas ainda não funcionam...
  - `http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf`

Outros

- *Making reliable distributed systems in the presence of software errors* – Joe Armstrong
  - Tese de Doutoramento
  - Legível e acessível mas certas coisas ainda não funcionam...
  - `http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf`
- Documentação da disciplina
  - ≪Fortemente inspirada≫ na documentação oficial, mas em português
  - Ver `http://www.dei.isep.ipp.pt/~paf`
  - Uma introdução ao Erlang mais slides
- Aulas práticas: `http://www.dei.isep.ipp.pt/~luis/orgc/index.html`

Mais coisas

Documentação oficial – Fevereiro de 2006

- http://www.erlang.org/doc.html
- Getting Started – http://www.erlang.org/doc/doc-5.4.12/pdf/getting_started-5.4.1
- Programming Examples – http://www.erlang.org/doc/doc-5.4.12/pdf/programming_examples-
- Reference Manual – http://www.erlang.org/doc/doc-5.4.12/pdf/reference_manual-5.4.
- Standard Library – http://www.erlang.org/doc/doc-5.4.12/pdf/stdlib-1.13.11.pdf
- Programming Rules – http://www.erlang.se/doc/programming_rules.pdf

Linguagens Funcionais

Linguagem Funcional

- Erlang é uma linguagem de programação funcional
- Os programas são escritos à custa de funções
- Noutras linguagens são escritos em termos de comandos

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

- Alto nível de abstração
- Programas mais compactos

Armadilhas

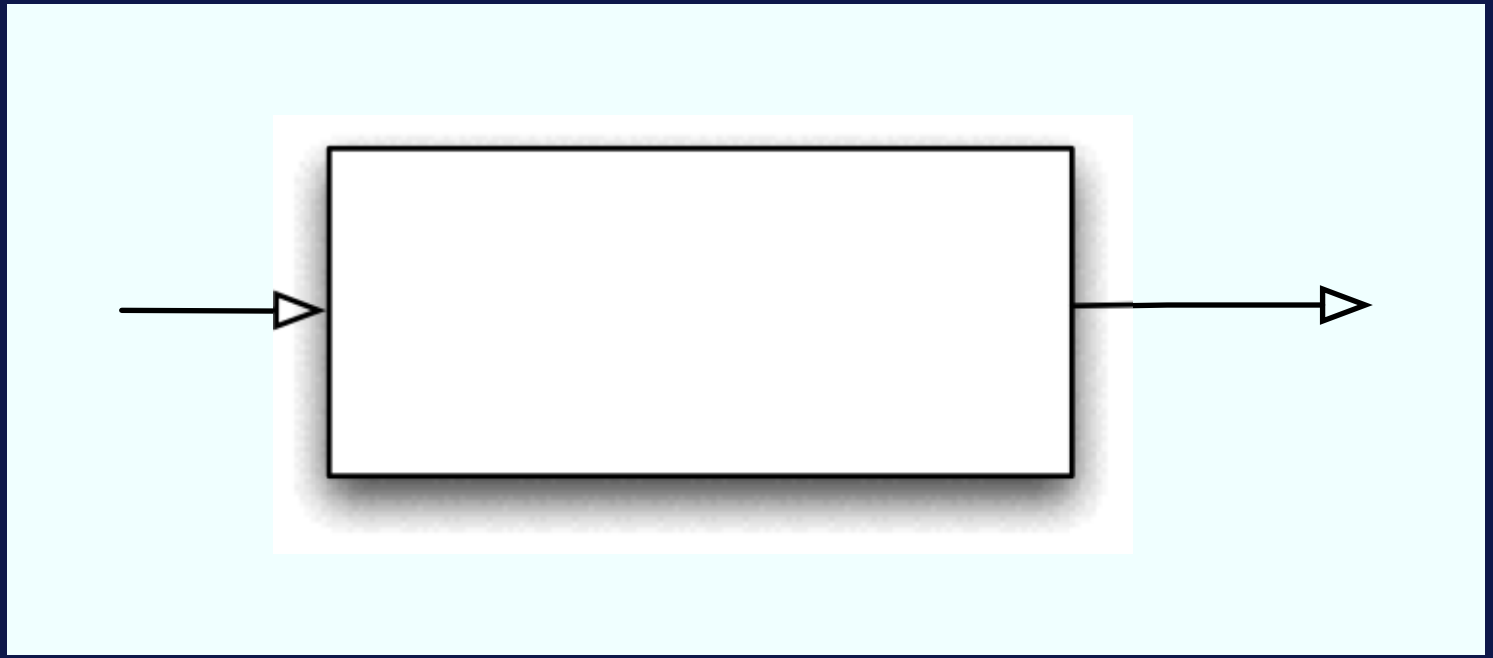
- O Erlang não é Prolog!
- Não há *backtracking*!
- Todas as variáveis passadas a uma função devem estar instanciadas.
- Uma função devolve sempre um valor.
- Uma variável não muda de valor depois de instanciada!

Linguagem funcional?

- Não se mudam os valores das variáveis.
- As funções não possuem «efeitos laterais».
- Exemplo de efeito lateral: variáveis globais.
- Sem efeitos laterais: puramente funcional.
- Sem efeitos laterais: É mais fácil analizar um programa do ponto de vista matemático.
- Sem efeitos laterais: Cada função pode ser analizada independentemente do ambiente.
- O Erlang não é uma linguagem puramente funcional (I/O e comunicação).

Caixas negras

- Para um determinado valor de entrada, a saída é sempre a mesma.
- Podemos *esquecer* o seu interior.
- Temos os *software ICs*, componentes modulares que são independentes entre si.



O que é programar?

- Resolver problemas decompondo-os em problemas mais simples.
- Os paradigmas da programação fornecem maneiras de estruturar a «fragmentação» dos problemas.
- As linguagens são uma forma de «explicitar» os conceitos que possuímos.
- Se o conceito não existir na linguagem teremos problemas em o expressar (no mínimo).

Vantagens das Linguagens Funcionais

- Se as funções não possuem efeitos laterais então podem ser analisadas de uma forma independente do contexto.
- Torna-se mais fácil a decomposição do problema/programa em funções.
- Torna-se mais fácil a reutilização de código.
- Torna-se mais fácil (ou possível) a análise formal do programa.
- Torna-se mais fácil a utilização de funções como parâmetros de outras funções.

E a sua utilização?

- A indústria não tem tempo para ser criativa, e a academia não é flexivel devido a restrições várias.
- Trauma das teorias matemáticas normalmente associadas às linguagens funcionais.
- Carga semântica extremamente forte de algumas linguagens funcionais.
- Vistas mais como assunto de pesquisa do que como ferramentas práticas.
- Falta de ≪ferramentas CASE≫.
- O debate sobre linguagens de programação assume muitas vezes contornos de ≪fanatismo religioso≫ onde se defende o que se conhece, contra a incerteza do que não se conhece.

Características

- Puras/Impuras: Depende da quantidade de efeitos laterais
- *Lazy* ou não: Avaliam sempre completamente os parâmetros ou apenas quando é necessário
- Com ou sem tipos: Verificação dos parâmetros
- Funções de ordem elevada: Aceitam funções como parâmetros

Outras linguagens funcionais

Scheme	http://www.drscheme.org/
Lisp	http://www.lisp.org
Dylan	http://www.gwydiondylan.org/
SML	http://www.smlnj.org/
(O)Caml	http://caml.inria.fr/
Haskell	http://www.haskell.org/
Clean	http://www.cs.ru.nl/~clean/
F#	http://research.microsoft.com/
J	http://www.jsoftware.com/
Mozart/Oz	http://www.mozart-oz.org/
Mercury	http://www.cs.mu.oz.au/research/mercury/
Pizza	http://pizzacompiler.sourceforge.net/
Qi	http://www.lambdassociates.org/

Tipos de Dados

Tipos de Dados em Erlang

Existem os tipos de dados seguintes:

- Números: Inteiros e Floats
- Àtomos
- Tuplos
- Listas
- Pids (*Process Ids*)
- Ports
- Referências
- Binários

Inteiros

```
10
234
16#ab10f
2#11010101011
$a
```

- `Base#Valor` representa um Valor numa certa base
- `$Char` representa o valor Ascii do Char
  (Exemplo: `$A` em vez de 65)

Números em vírgula flutuante

```
17.368
56.656
12.34E-10
```

Àtomos

Os àtomos são constantes literais:

```
abcef23
comecam_com_uma_letra_minuscula
'podem ter espaços'
'Ou qualquer char dentro de pelicas \n\012'
```

- O valor de um átomo é o próprio àtomo
- Começam com uma letra minúscula
- Podem conter qualquer código de caracter
- Fora de pelicas usar apenas letras normais+dígitos+underscore

≪Plicas≫

- Na realidade – apóstrofo!
- Uma coisa é um acento agudo (´), outra um apóstrofo ('), em inglês ≪*quote*≫.
- Estão em sítios diferentes nos teclados:
- Teclado português – Plica encontra-se ao lado direito do zero.
- OK?

Caracteres Especiais

Se o átomo tiver plicas:

\b	Backspace
\d	Delete
\e	Escape
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab
\v	Vertical Tab
\\	Backslash
\^A	Control-A (idem até Z) Ascii 0 até Ascii 26
\'	Pelica
\"	Aspas
\000	O caracter com o código octal 000

Tuplos

- São usados para guardar um número fixo de items
- Podemos ter tuplos de qualquer tamanho
- Podemos ter tuplos de qualquer nível
- O nível dos tuplos não é fixo

```
{123,bcd}
{123,def,abc}
{pessoa,'Joe','Armstrong'}
{abc,{def,123},jkl}
{}
```

Funções Auxiliares

- `element(N,T)` – devolve elemento na posição `N`
  - `element(2,{a,b,c})` $\Rightarrow$ `b`
- `setelement(N,T,V)` atribui ao elemento na posição `N` o valor `V`
  - `setelement(2,{a,b,c},x)` $\Rightarrow$ `{a,x,c}`

Listas

- Uma lista é a lista vazia: `[ ]`
- Ou um elemento seguido por uma lista: `[Elem|Lista]`

```
[ ]
[3|[ ]]              <--> [3]
[2|[3|[ ]]]          <--> [2,3]
[1|[2|[3|[ ]]]]      <--> [1,2,3]
```

- São usadas para representar um número variável de itens
- As strings são representadas à custa de listas

`"abcdef"`$\Leftrightarrow$ `[97,98,99,100,101,102]`

Estruturas de dados

- Podemos criar estruturas de dados complexas

```
[{{pessoa,'Joe','Armstrong'},
{n_telefone,[3,5,9,7]},
{sapatos,42},
{bichos,[{gato,tubby},{gato,tiger}]},
{filhos,[{thomas,5},{claire,1}]}},
{{pessoa,....
.....}]
```

- São criadas quando as escrevemos,sem termos de gerir a memória
- As estruturas de dados podem conter variáveis já instanciadas

Variáveis

- São usadas para guardar termos Erlang

```
ABC
Uma_variavel_com_o_nome_grande
ComONomeGrandeOrientadoAoObjecto
```

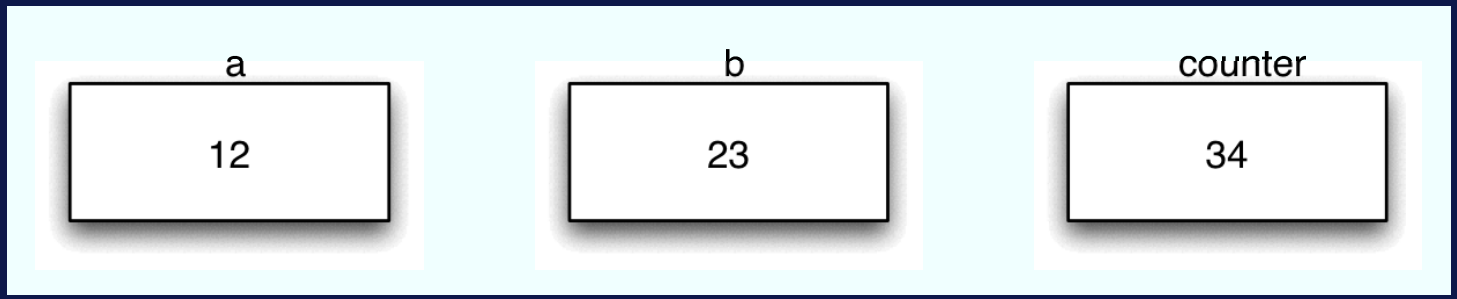
- Começam com uma letra maiúscula
- Evitar caracteres estranhos/esquisitos fora do ASCII
- Não é necessário declarar variáveis
- As variáveis só podem ser instanciadas uma vez. O valor de uma variável não pode mudar depois de ter sido atribuído
- Variável Anónima _ (underscore)
  - Podemos fazer a instanciação repetida do _ com qualquer valor, qualquer número de vezes
  - Extremamente útil quando a sintaxe exige uma variável mas não nos interessa o seu valor

Variáveis imperativas

- Exemplo de variáveis em C:

```
int myfunc(int a,b)
{
 counter=counter+1;
 return(a+b+counter)
 }
```

- Ilustração gráfica:

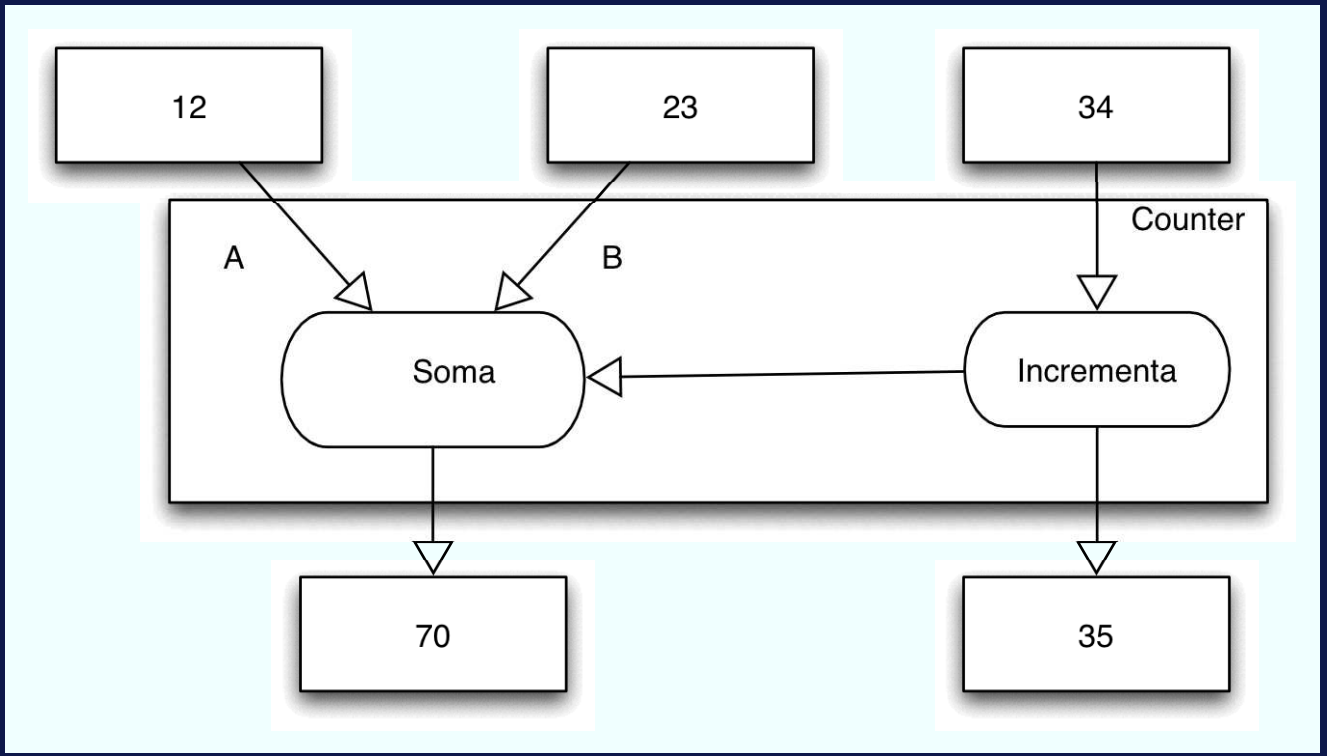


Variáveis funcionais

- Exemplo de variáveis em Erlang:

```
myfunc(A,B,Counter)->{A+B+Counter+1,Counter+1}.
```

- Ilustração gráfica:



Polémica

Há três slides atrás: ≪Não é necessário declarar variáveis≫

- Isto quer dizer que não necessitamos de declarar nem que variáveis vamos usar, nem o seu tipo...

Polémica

Há três slides atrás: ≪Não é necessário declarar variáveis≫

- Isto quer dizer que não necessitamos de declarar nem que variáveis vamos usar, nem o seu tipo...
- Com tipos não se apanham mais erros?

Polémica

Há três slides atrás: ≪Não é necessário declarar variáveis≫

- Isto quer dizer que não necessitamos de declarar nem que variáveis vamos usar, nem o seu tipo...
- Com tipos não se apanham mais erros?
- Sim, não e talvez...;-)

Polémica

Há três slides atrás: ≪Não é necessário declarar variáveis≫

- Isto quer dizer que não necessitamos de declarar nem que variáveis vamos usar, nem o seu tipo...
- Com tipos não se apanham mais erros?
- Sim, não e talvez...;-)
- Ganha-se em robustez mas perde-se em agilidade
- Pode-se analisar a correcção de um programa escrito em Erlang mesmo sem tipos[1]

[1] `http://www.it.uu.se/research/group/hipe/dialyzer/`

Pattern Matching

É usado para três coisas:

Atribuição	`A=2`
Extração de dados	{Nome,Conteudo}={minhalista,[1,2,3]}
Teste	`Valor=1, {Valor,Valor}=UmTermoQualquer`

Exemplos:

`A=10`	Sucesso, A é instanciada com o valor 10
`{B,C,D}={10,pim,pam}`	Sucesso, B ⇐ 10, C ⇐ pim, D ⇐ pam
`{A,A,B}={abc,abc,pim}`	Sucesso, A ⇐ abc e B ⇐ pim
`{A,A,B}={abc,def,123}`	Falha

Programação simples

Funções 1

- Função simples:

```
soma(Padrao1,Padrao2) ->
      Padrao1+Padrao2.
```

- Tem uma cabeça, com o nome da função e os parâmetros, o corpo da função e o terminador (ponto final)
- Os parâmetros podem ser qualquer padrão
- Uma função pode não ter argumentos
- As funções devolvem sempre um valor determinado pela última acção feita no corpo

Funções 2

Uma função é usada quando é chamada pelo seu nome com o número de parâmetros correcto

```
vezes(X,N)->X*N.
dobro(X)->vezes(X,2).
```

Uma função com o mesmo nome mas um número de parâmetros diferente é uma função diferente

Funções 3

De uma forma mais geral uma função parece-se com:

```
funcao(Padrao1,Padrao2,..)->
  <instruçao>,
  ....
  <instruçao>;
  ......
funcao(PadraoM,PadraoN,..)->
  <instruçao>,
  ....
 <instruçao>.
```

Regras de avaliação

- As cláusulas são pesquisadas sequencialmente até se encontrar um padrão correcto
- Quando isso acontece todas as variáveis que ocorrem na cabeça da função ficam instanciadas
- As variáveis são locais para cada cláusula, a sua alocação é automática
- O corpo é avaliado sequencialmente

Factorial

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

```
factorial(3)
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
==3*factorial(2)
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
==3*factorial(2)
==3*2*factorial(2-1)
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
==3*factorial(2)
==3*2*factorial(2-1)
==3*2*factorial(1)
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
==3*factorial(2)
==3*2*factorial(2-1)
==3*2*factorial(1)
==3*2*1
```

Factorial

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

```
factorial(3)
==3*factorial(3-1)
==3*factorial(2)
==3*2*factorial(2-1)
==3*2*factorial(1)
==3*2*1
==6
```

Variáveis

Cuidados com as variáveis:

- As variáveis são locais a cada cláusula e não a cada função
- A memória correspondente às variáveis é alocada e libertada automaticamente
- Todos os parâmetros devem ser conhecidos quando a função é chamada

Módulos 1

- As funções devem ser definidas num ficheiro que constitui um módulo:

```
-module(demo).
-export([dobro/1]).
    vezes(X,N)->
    X*N.
    dobro(X)->vezes(X,2).
```

- O módulo `demo` deve ser gravado num ficheiro chamado `demo.erl`
- A função `dobro/1` pode ser chamada de fora do módulo enquanto que a função `vezes/2` é local ao módulo

Módulos 2

- As funções são identificadas de um forma única através do módulo, nome da função e aridade.
- Uma função deve ser exportada para ser visível fora do módulo em que está inserida
- Uma função no mesmo módulo é chamada usando:
  `funcao(Arg1,...,ArgN)`
- Uma função noutro módulo é chamada usando:
  `modulo:funcao(Arg1,...,ArgN)`
- Exemplo:
  `1>N=demo:dobro(2).`

Módulos 3

- Dentro do mesmo módulo pode ser omitido o nome do módulo na chamada de uma função
-  Se quisermos substituição do código em *run-time* então é obrigatório usar o nome do módulo em todas as chamadas a funções

Módulos 4

```
-module(mathstuff).
-export([area/1]).
%% calcular a àrea de vários objectos
area({quadrado,Lado})-> Lado*Lado;
area({circulo,Raio})->
   math:pi()*Raio*Raio;
area({triangulo,A,B,C})->
     S=(A+B+C)/2,math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Outro)->{objecto_invalido,Outro}.
```

Built-in Functions

Existe um certo número de funções incluídas na linguagem, localizadas no módulo erlang, que fazem coisas impossíveis (ou difíceis em Erlang).
Exemplos:

```
date()                  ⇒ {1997,8,7}
time()                  ⇒ {10,46,5}
length([1,2,3,4,5])     ⇒ 5
size({a,b,c})           ⇒ 3
list_to_tuple([1,2,3,4]) ⇒ {1,2,3,4}
integer_to_list(2234)   ⇒ "2234"
tuple_to_list({})       ⇒ []
```

Lista de BIFs

Esta lista não é exaustiva

`is_atom/1`	`is_binary/1`	`is_list/1`
`is_float/1`	`is_integer/1`	`is_number/1`
`is_pid/1`	`is_reference/1`	`is_record/1`
`length(List)`	`size(Tuple)`	`element(N,Tuple)`
`trunc(Float)`	`round(Float)`	`float(Int)`

Prioridades 1

- Expressões Unárias (prioridade 1):

```
+X
-X
bnot X
```

- Expressões Binárias (prioridade 2)

```
X*Y
X/Y
X div Y
X rem Y
X band Y
X+Y
```

Prioridades 2

- Expressões binárias (prioridade 3):

```
X-Y
X bor Y
X bxor Y
X bsl N
X bsr N
```