Uma introdução ao Erlang

Paulo Alexandre Duarte Ferreira
paf@dei.isep.ipp.pt
DEI - ISEP
Instituto Politécnico do Porto

Versão α 0.1
Fevereiro de 2006

Apresentação

O Erlang é uma linguagem funcional, vocacionada para a construção de aplicações concorrentes, distribuídas, tolerantes a falhas, com características de «*soft real-time*», de alta disponibilidade. Foi desenvolvida pela Ericsson para aplicações na área das telecomunicações existindo versões comerciais com suporte[1] para vários sistemas operativos e versões «*Open Source*» para os sistemas operativos mais comuns[2].

Este documento pretende apenas fazer uma apresentação suave da linguagem Erlang e das suas potencialidades, não procurando ser exaustivo nem substituir a documentação existente.

[1] `http://www.erlang.se`
[2] `http://www.erlang.org`

Ficha técnica

Este texto foi produzido usando o $\LaTeX$ com o TeXShop como «frontend», no OS-X. O tipo de fonte utilizado é o Palatino.

Muitos exemplos, exercícios e algum texto foram retirados da documentação original do Erlang. Recomenda-se a leitura dessa documentação em caso de dúvidas (ou gralhas) neste documento.

O autor deste documento agradece a comunicação de erros, gralhas e sugestões sobre o conteúdo deste documento por correio electrónico.

Índice

1 Razões para gostar de Erlang 1
- 1.1 Razões funcionais 1
- 1.2 Memórias e esquecimento 6
- 1.3 Processos, processos e mais processos 6
- 1.4 O que o Erlang não é 8

2 Erlang básico 11
- 2.1 Tipos de Dados em Erlang 11
  - 2.1.1 Inteiros 11
  - 2.1.2 Números em vírgula flutuante 11
  - 2.1.3 Àtomos 12
  - 2.1.4 Tuplos 13
  - 2.1.5 Listas 13
  - 2.1.6 Estruturas de dados complexas 14
  - 2.1.7 Outros tipos 14
  - 2.1.8 Variáveis 15
  - 2.1.9 Comparação de padrões 16
- 2.2 Funções 16
  - 2.2.1 Funções simples 16
  - 2.2.2 Módulos 19
  - 2.2.3 Funções incorporadas 20

3 Programação sequencial 23
- 3.1 «Guards» 23
  - 3.1.1 Ordenação 24
- 3.2 Recursividade 25
  - 3.2.1 Introdução 25
  - 3.2.2 Ciclos 25

3.2.3 Exemplos de recursividade 27
3.2.4 Controle de fluxo 30

4 Processos 33
4.1 Introdução 33
4.2 Mensagens 34
4.2.1 Envio de mensagens 34
4.2.2 Recepção de mensagens 35
4.2.3 Processos registados 39
4.2.4 Prazos de recepção 39

5 Erros 41
5.1 Filosofia dos erros em Erlang 41
5.2 *Links* 41
5.2.1 Introdução 41
5.2.2 Sinais de fim 42
5.2.3 A função *exit* 44
5.3 Detecção de erros 45
5.3.1 Funcionamento 45
5.3.2 Um servidor robusto 47

6 Extensões 53
6.1 Records 53
6.1.1 Representação dos «Records» 55
6.2 Operações com Listas 57
6.3 Compreensão de Listas 57
6.4 Macros 61
6.5 Binários 63
6.6 Objectos Funcionais 67

1 Razões para gostar de Erlang

1.1 Razões funcionais

O Erlang é uma linguagem de programação funcional. Isto é uma definição correcta, mas como todas as definições é inútil para se perceber aquilo de que se está a falar, se não percebermos antecipadamente o assunto.

As linguagens de programação mais comuns, como o BASIC, C ou Java são chamadas linguagens imperativas. Neste tipo de linguagens dá-se ordens ao computador e ele executa o que está escrito no programa. Vamos ver um exemplo escrito em C[1] :

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

Temos uma função chamada `myfunc` mas uma função em C é apenas um grupo de instruções para o computador [2]. Em termos matemáticos uma função é algo cujo valor de saída apenas depende dos valores de entrada, e não de outros valores.

Portanto, a nossa *função* em C não é uma *verdadeira* função porque também depende do valor da variável `counter`. O que temos dentro da definição da função é uma série de comandos, que devem ser executados na ordem correcta para

[1] É mau C, mas serve para o que queremos.

[2] Só porque podemos definir *funções* numa linguagem de computador, não podemos dizer que essa linguagem seja uma linguagem *functional*.

termos o resultado que queremos. Vamos traduzir a expressão «`counter=counter+1`»:

- Ler o valor da variável `counter`
- Somar um ao valor lido
- Guardar o resultado na variável `counter`

Aquilo a que chamamos normalmente uma expressão, não é uma expressão, mas uma série de ordens para o computador, e o nosso modelo mental do computador possui uma série de caixas (as variáveis) que podemos usar para colocar valores lá dentro.

Este modelo é o modelo da programação em linguagem «*assembly*», em que em vez de diferentes números para diferentes endereços de memória, possuímos a facilidade de usar nomes para o programa ficar mais claro.

As variáveis no modelo de programação imperativo são assim apenas «caixas» onde podemos colocar «coisas».

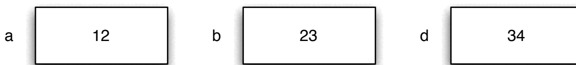

Figura 1.1: Variáveis numa linguagem imperativa

Mas numa linguagem de programação imperativa, nós não tratamos as variáveis da mesma forma que as tratamos numa expressão matemática. Numa expressão matemática x não pode ser igual a $x+1$. Portanto se quisermos uma linguagem de programação funcional, as variáveis só podem receber um valor uma vez. Sim, depois de darmos uma valor a uma variável não o podemos mudar[3].

[3] Por favor não entre em pânico! Isto fará sentido mais tarde.

O que é que ganhamos com isto? Bem, desta forma o programa pode ser analisado matematicamente de uma forma mais fácil, e poderemos demonstrar o seu comportamento. As variáveis são usadas não para armazenar valores, mas para expressar as dependências que existem entre elas e o fluxo de dados. Por exemplo a seguinte função em Erlang devolve a soma das duas variáveis de entrada.

```
  %  comentário
-module(mymodule).
  % nome deste módulo (=ficheiro)
-export([myfunc/2]).
  % declaração das funções "exportadas"
  % que se podem usar de fora deste módulo
myfunc(A,B)->A+B.
```

Voltaremos à sintaxe do Erlang mais tarde, mas o importante por agora é que o valor de `myfunc` vem da soma dos dois valores de entrada. Usamos os nomes das variáveis para especificar diferentes «conexões» dentro da função. Nós não temos variáveis globais numa linguagem funcional, por isso para converter o programa anterior em C para um equivalente em Erlang teremos um «trabalho» adicional. A nossa função receberá um argumento adicional (o contador) e em vez de devolver apenas a soma, devolverá dois valores, a soma e o contador.

```
-module(mymodule).
-export([myfunc/3]).

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

Porque é que havemos de ter este trabalho todo? Porque desta maneira podemos estar seguros do comportamento da função. Se uma função depende de uma variável global ou muda uma variável global, nós não podemos dizer o que acontece quando uma função é chamada, apenas olhando para os argumentos.

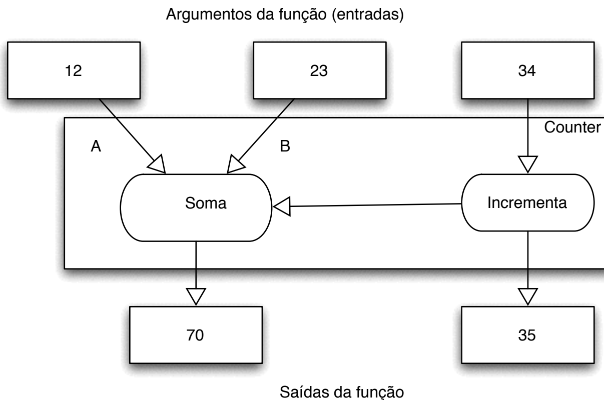

Figura 1.2: Variáveis numa linguagem funcional

Da mesma forma uma função matemática «normal» não tem *efeitos laterais*[4]. Se olharmos para a função matemática $f(x) = 2x^2$ esta dá sempre o mesmo resultado para valores iguais de x. Por outro lado uma função em C, chamada com os mesmos argumentos pode dar diferentes valores.

Os efeitos laterais são maus, mas necessitamos normalmente de um compromisso quando necessitamos de entrada e saída de dados num programa. Por exemplo, podemos dizer que um ficheiro é uma variável imperativa *muito grande* e necessitamos de poder mudar o conteúdo dos ficheiros para fazermos programa úteis. Não podemos dizer que o conteúdo dos ficheiros só pode mudar uma vez!

Este problema pode ser resolvido de duas maneiras. Uma delas é dizer que a linguagem é funcional mas não é 100% funcional quando tem coisas como instruções de entrada e saída

[4]«*Side effects*» em inglês.

«clássicas». Desta forma dizemos que se trata de uma linguagem funcional «impura», porque tem alguma «contaminação» das linguagens imperativas. Esta é a aproximação usada pelo Erlang e é um compromisso razoável na minha opinião.

Outra aproximação é fazer uma linguagem de programação 100% funcional. Neste caso podemos fazer uma análise formal mais profunda dos nossos programas, mas para operações de entrada e saída necessitamos de operadores especiais como os *«monads»* na linguagem Haskell ou *«Uniqueness type systems»* como nas linguagens Clean[5] e Mercury[6].

A atribuição única de valores às variáveis é um choque terrível para um programador que apenas tenha visto linguagens imperativas, e um programador demora algum tempo a habituar-se ao conceito. No entanto, simplifica o pensar o programa, e simplifica a análise do programa. Existem muitas linguagens funcionais, incluindo por exemplo uma chamada «*Single Assignment C*[7]». Mas a atribuição única é apenas uma parte da diferença entre as linguagens imperativas e as linguagens funcionais.

O nome de uma variável numa linguagem funcional é usado para etiquetar as caixas onde podemos colocar valores, enquanto que numa linguagem funcional o nome das variáveis é usado para etiquetar as conexões dentro de uma função. Logo, as variáveis são coisas diferentes numa linguagem funcional e numa linguagem imperativa[8].

[5] `http://www.cs.ru.nl/~clean/`

[6] `http://www.cs.mu.oz.au/research/mercury/`

[7] `http://www.sac-home.org`

[8] A pergunta – *«As variáveis são passadas por referência ou por valor?»* não faz sentido numa linguagem funcional. Numa linguagem funcional apenas se passam valores.

1.2 Memórias e esquecimento

Um dos problemas da programação é a «gestão de memória» que o programador tem de fazer no seu cérebro. Ele tem de estar consciente de um enorme conjunto de detalhes, e tem de se esquecer de alguns deles enquanto codifica uma parte de um programa; meia-hora depois necessita de esquecer esses detalhes enquanto focaliza a sua atenção noutros.

Uma boa linguagem de programação é uma linguagem na qual podemos esquecer (de uma forma segura) montes de detalhes, e uma forma de fazer isso é minimizar as interacções entre diferentes partes de um programa. Com uma linguagem funcional podemos fazer isso, porque os únicos pontos possíveis de interacção com uma função são os seus argumentos e o seu resultado. Quando estamos a codificar uma função podemos ignorar tudo o resto, que não seja os seus argumentos e o seu resultado. Quando usamos uma função, não necessitamos de saber como ela faz o que faz, apenas necessitamos de saber o que a função faz e quais os seus argumentos, e o seu resultado. Uma linguagem funcional ajuda-nos a ter a mente livre de detalhes desnecessários.

1.3 Processos, processos e mais processos

Uma das maneiras de dividir um programa em «partes» mais fáceis de gerir, é dividir-lo em processos diferentes. Em vez de «partir» um programa em procedimentos, ou classes, ou objectos, este é estruturado em termos de processos.

Desta forma temos concorrência que nos pode ajudar a ter uma maior performance, uma maior fiabilidade, um programa mais claro, ou tudo isto ao mesmo tempo.

Numa linguagem de programação «normal» ter centenas de processos pode ser um pesadelo, mas em Erlang é fácil ter milhares de processos. Em linguagens normais, o paralelismo é

conseguido à custa de «*locks*», semáforos e variáveis partilhadas. Em Erlang a comunicação entre processos é feita através da passagem de mensagens, sem que seja necessário um mecanismo de «*locking*». Esta aproximação conduz a que os processos sejam codificados de uma forma elegante e simples como veremos mais à frente.

A visão académica do paralelismo tem sido centrada na performance até há pouco tempo. As máquinas paralelas e os programas paralelos têm sido vistos apenas como uma forma de obter uma maior performance e as análises do paralelismo têm sofrido deste erro de visão. É claro que podemos ter um programa mais rápido numa máquina paralela, mas também podemos usar o paralelismo para ter uma maior fiabilidade e/ou uma melhor aproximação ao problemas que estamos a tentar resolver.

Um bom exemplo disto é o caso de um servidor Web. Se tivermos um processo (ou mais) por cada «cliente» então a programação será mais fácil. Se cada processo que «trabalha» tiver um «supervisor» adicional a vigiar o seu estado então isso conduzirá a uma maior fiabilidade do sistema. A fiabilidade é conseguida assim à custa da estruturação em «camadas» dos processos e da filosofia do «estourar» o mais cedo possível.

A um nível mais baixo existem os «trabalhadores» que apenas fazem o «seu trabalho», e quando não o conseguem fazer, devem «estourar» para que o erro seja detectado o mais cedo possível. Acima destes existem os «supervisores» que tomam conta dos trabalhadores tendo a responsabilidade de os «recriar» quando estes «estouram», tal como se mostra na figura 1.3.

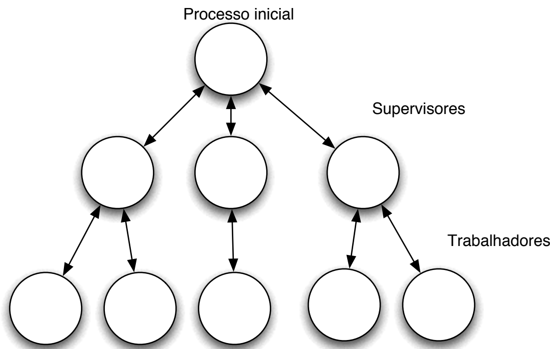

Figura 1.3: Uma possível hierarquia de processos

1.4 O que o Erlang não é

Com a programação funcional podemos ter uma maior reutilização de código[9]. A aproximação funcional também reduz o número de linhas necessário para uma dada tarefa.

O Erlang não é uma linguagem onde tenhamos de declarar antecipadamente os tipos das variáveis que queremos. Este tema assume por vezes os contornos de um debate religioso do tipo fundamentalista. Alguns dizem que as linguagens com sistemas de tipos fortes são mais robustas, e um sistema de tipos forte ajuda a ter um programa mais correcto. Outros dizem que um sistema de tipos é como uma armadura medieval, que nos dá protecção, mas nos tira agilidade e rapidez. A visão dos criadores do Erlang é a segunda[10].

[9] A reutilização de código não é fazer «copy-paste» de um ficheiro para outro! É reutilizar o código de uma forma fácil, sem ler outra vez todo o código fonte ou ter de verificar todas as possíveis interacções.

[10] Nas conferências de utilizadores de Erlang existem comunicações sobre

O Erlang não é uma linguagem orientada aos objectos. Isto pode parece estranho dada a força corrente do paradigma de programação orientado aos objectos, mas existem várias boas razões para que isto assim seja.

Uma das razões é que uma linguagem funcional não necessita dos objectos para que haja uma grande reutilização de código. Outra das razões é que os objectos normalmente apenas transferem a complexidade do código para a hierarquia de classes. O código fica simples apenas se a hierarquia de classes for complexa, sendo a hierarquia de classes mais uma coisa a ocupar a cabeça do programador. Outra razão é que os objectos e a concorrência não se misturam bem. Os objectos (ou métodos) remotos são desajeitados e complexos quando comparados com a visão da concorrência fornecida pelo Erlang.

Se estas opiniões são correctas, então porque é que (quase) ninguém usa uma linguagem funcional ou o Erlang? Antes de mais, medir a qualidade de algo pelo seu sucesso ou popularidade é um mau critério. Existe ainda um sério divórcio entre algumas visões académicas sobre a programação e o que a indústria quer que a programação seja.

As linguagens funcionais são normalmente vistas pela indústria como sendo algo eminentemente teórico, muito difícil de usar, e uma péssima escolha para programas «úteis». Alguns académicos estão interessados apenas em provar que as coisas funcionam de um ponto de vista matemático; se se provar que funciona então não é necessário construir nada para se demonstrar que funciona.

Dizendo a mesma coisa de outra maneira, a indústria é conservadora porque essa é uma atitude mais segura do ponto de vista empresarial. O objectivo da indústria é construir sistemas que funcionam, enquanto o objectivo dos (de alguns) académicos é provar que as suas ideias talvez funcionem. A construção de sistemas reais é vista por alguns académicos como

---

como adicionar tipos de dados ao Erlang.

uma opção a evitar a todo o custo.

2 Erlang básico

2.1 Tipos de Dados em Erlang

Em Erlang existem os seguintes tipos de dados:

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

2.1.1 Inteiros

Um número inteiro pode ser representado em Erlang de várias formas, além da numeração decimal, podemos usar a notação `Base#Valor` para representar um valor numa certa base, ou `$Char` para representar o valor ASCII do caracter em questão.

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

2.1.2 Números em vírgula flutuante

Um número para ser de vírgula flutuante tem de ter um ponto a separar a parte inteira da decimal, podendo ter um expoente adicional.

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

2.1.3 Àtomos

Os àtomos são constantes literais, que servem de etiquetas e permitem ter um programa mais legível. Para distinguir de uma forma fácil os átomos das variáveis, a convenção usada é começar o nome dos átomos por uma letra minúscula, e usar apenas letras, dígitos e o «*underscore*» no nome de um átomo. Se quisermos ser mais flexíveis no nome dos átomos, podemos usar (quase) qualquer caracter desde que o nome do átomo fique dentro de plicas[1].

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

Se o átomo tiver plicas podemos usar as seguintes abreviaturas:

[1]Uma das maiores confusões que pode acontecer a um novato em Erlang é confundir um acento agudo «´», com uma plica ou apóstrofo «'» («*quote*» em inglês). Estes caracteres estão em sítios diferentes nos teclados, sendo que no arranjo normal de um teclado português a plica encontra-se ao lado direito do zero.

`\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
`\'`	Plica
`\"`	Aspas
`\000`	O caracter com o código octal 000

2.1.4 Tuplos

Para associar um número fixo de itens entre si podemos usar os tuplos que são apenas a agregação de vários valores. Isto permite por exemplo, que uma função tenha como saída vários valores, associados num tuplo. Podemos ter tuplos de qualquer tamanho e de qualquer nível, o que pode servir para implementar estruturas de dados como árvores.

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

2.1.5 Listas

As listas são estruturas de dados com um número de elementos variável. As listas são úteis porque a sua manipulação é fácil de fazer se usarmos algoritmos recursivos.

Assim podemos definir uma lista de uma forma recursiva como sendo a lista vazia `[ ]` ou um elemento seguido por

uma lista: `[Elem|Lista]`

```
[ ]                  []
[3|[ ]]              [3]
[2|[3|[ ]]]          [2,3]
[1|[2|[3|[ ]]]]      [1,2,3]
```

Em Erlang as strings são representadas à custa de listas: `"abcdef"` é o mesmo que `[97,98,99,100,101,102]`

2.1.6 Estruturas de dados complexas

Podemos criar estruturas de dados complexas, colocando tuplos dentro de listas, listas dentro de tuplos, listas dentro de listas e tuplos dentro de tuplos até termos a estrutura desejada.

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

Uma das vantagens do Erlang é que a gestão de memória é automática. Assim as estruturas de dados são criadas quando as escrevemos, sem termos de gerir a memória e podem conter variáveis já instanciadas.

2.1.7 Outros tipos

Os outros tipos de dados serão descritos em pormenor mais à frente. Os Pids (ou «*process ids*») funcionam como identificadores dos diferentes processos, servindo entre outras coisas para a comunicação entre processos. Os Ports são canais de comunicação com outros programas, que podem estar escritos

noutra linguagem que não o Erlang. As referências funcionam como identificadores garantidamente únicos. Isto é, cada referência é diferente de todas as outras. Os binários são conjuntos de bytes (ou bits) que servem para para que se possa processar de uma forma mais fácil, estruturas como pacotes TCP/IP por exemplo.

2.1.8 Variáveis

As variáveis são usadas para guardar termos Erlang de qualquer complexidade. O seu nome começa com uma letra maiúscula, sendo de evitar caracteres fora dos normais, como os acentuados. As variáveis não necessitam de ser declaradas e só podem ser instanciadas uma vez. Isto é, depois de uma variável receber um valor, este não pode ser alterado.

```
ABC
Uma_variavel_com_o_nome_grande
ComONomeGrandeOrientadoAoObjecto
```

Nos casos em que a sintaxe do Erlang nos exige uma variável, mas não nos interessa o seu valor podemos usar a variável anónima _ ou «*underscore*». Neste caso podemos fazer a instanciação repetida de _ com qualquer valor, qualquer número de vezes. A variável anónima é apenas o _ ou qualquer variável cujo nome comece por _.

É uma boa prática de programação dar nomes longos ás variáveis anónimas, para uma melhor compreensão do programa, e para prever o caso de as variáveis anónimas deixarem de o ser. Por exemplo entre `_Nome` e _ é preferível o primeiro caso. Uma das tarefas mais importantes do programador (e muitas vezes descuidada) é a escolha de nomes correctos para as variáveis, sejam estas anónimas ou não.

2.1.9 Comparação de padrões

O operador = pode ser usado para três coisas diferentes:

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

Eis aqui alguns exemplos do seu uso e o seu resultado prático:

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

2.2 Funções

2.2.1 Funções simples

Uma função simples tem uma cabeça, com o nome da função e os parâmetros, o corpo da função e o terminador (ponto final). Exemplo:

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

Os parâmetros podem ser qualquer padrão, uma função pode não ter argumentos e as funções devolvem sempre um valor determinado pela última acção feita no corpo. Uma função é usada quando é chamada pelo seu nome com o número de parâmetros correcto. Exemplos:

```
quatro()->4.
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! Assim temos de ver não só o nome da função mas o seu número de parâmetros para distinguir entre duas funções diferentes.

Isto é importante porque como vamos ver mais à frente, é normal para codificarmos uma função recorrermos a outra função auxiliar com um número de parâmetros diferente.

De uma forma mais geral a declaração de uma função é separada por várias cláusulas assumindo a seguinte forma:

```
função(Padrão1,Padrão2,..)->
    <instrução>,
    ....
    <instrução>;
    ....
função(PadrãoM,PadrãoN,..)->
    <instrução>,
    ....
    <instrução>.
```

Podemos usar qualquer número (razoável) de cláusulas na definição de uma função sendo que a última cláusula de todas termina com ponto final, enquanto todas as outras terminam com ponto e vírgula.

Para a avaliação de uma função as cláusulas são pesquisadas sequencialmente até se encontrar um padrão correcto. Quando isso acontece todas as variáveis que ocorrem na cabeça da função ficam (têm de ficar) instanciadas[2].

As variáveis são locais para cada cláusula, sendo a sua alocação automática, e o corpo das cláusulas é executado. sequencialmente.

[2]O Erlang não é Prolog embora alguma da sintaxe seja parecida! Quando se «entra» na execução de uma cláusula todas as variáveis estão obrigatoriamente instanciadas, e não existe de forma alguma «*backtracking*» sendo o teste das cláusulas e a execução das instruções puramente sequencial.

Exemplo de uma função simples:

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

A sua avaliação (por exemplo) se chamarmos a função com o argumento 3, será feita pelo Erlang da seguinte forma:

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

Além de ser óbvio que se trata de uma função recursiva, note-se que para `factorial(3)` e `factorial(2)` a primeira cláusula não é válida sendo aplicada a segunda.

Para `factorial(1)` a primeira cláusula é válida, e como esta é válida é esta a ser a única a ser executada. Se uma determinada chamada a uma função resultar na validade de mais do que uma cláusula, então só a primeira das cláusulas é que será executada.

Como exemplo, nesta definição de função, o corpo da segunda cláusula nunca será executado:

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

Outro cuidado que temos de ter com as variáveis é perceber que estas «existem apenas» dentro de cada cláusula e não são partilhadas pela função toda. Isto é, as variáveis servem para «etiquetar os valores» apenas dentro de cada cláusula.

Como exemplo, se tivermos a seguinte definição:

```
vezes(1,X)->X;
vezes(2,Y)->2*Y;
vezes(W,Z)->W*Z.
```

é para o Erlang a mesma coisa que:

```
vezes(1,X)->X;
vezes(2,X)->2*X;
vezes(Y,X)->Y*X.
```

2.2.2 Módulos

Como forma de organizar o código fonte, as funções devem ser definidas num ficheiro que constitui um módulo do ponto de vista do Erlang:

```
-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`, neste caso a função `dobro/1` pode ser chamada de fora do módulo enquanto que a função `vezes/2` é local ao módulo. As funções são identificadas de um forma única através do módulo, nome da função e da sua aridade. Chama-se aridade da função ao seu número de argumentos, sendo a função designada muitas vezes pela forma `nome/aridade` como se pode ver acima.

Uma função deve ser exportada para ser visível fora do módulo em que está inserida, sendo que só podemos usar fora de um módulo, funções que tenham sido exportadas. Isto é, as funções que não sejam exportadas, são obrigatoriamente para uso «interno» no módulo.

Uma função no mesmo módulo é chamada usando:

```
funcao(Arg1,...,ArgN)
```

Uma função noutro módulo é chamada usando:

```
modulo:funcao(Arg1,...,ArgN)
```

Exemplo de um módulo:

```
-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}.
```

2.2.3 Funções incorporadas

Existe um certo número de funções incorporadas na linguagem («BIFs ou *Built-in Functions*»), localizadas no módulo `erlang`, que fazem coisas impossíveis (ou difíceis de fazer) 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({})         []
```

A prioridade das expressões matemáticas é a seguinte em Erlang, sendo a prioridade mais baixa a mais elevada.

Expressões Unárias (prioridade 1):

`+X`	+ unário
`-X`	- unário
`bnot X`	operador não binário

Expressões Binárias (prioridade 2):

`X*Y`	multiplicação
`X/Y`	divisão
`X div Y`	divisão inteira
`X rem Y`	resto da divisão
`X band Y`	e binário
`X+Y`	soma

Expressões binárias (prioridade 3):

`X-Y`	subtracção
`X bor Y`	ou binário
`X bxor Y`	ou exclusivo binário
`X bsl N`	deslocamento binário para a esquerda
`X bsr N`	deslocamento binário para a direita

3 Programação sequencial

3.1 «Guards»

Considerando o exemplo anterior da nossa definição da função factorial, podemos ver que esta não está protegida contra o uso de argumentos negativos. Olhando para o código podemos ver que (por exemplo) o cálculo de factorial de -1 vai dar problemas.

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

Para resolver o problema podemos usar aquilo a que em Erlang se chama um «guard» da seguinte forma:

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

A palavra reservada `when` introduz um «guard», e note-se mais uma vez que todas as variáveis usadas no «guard» (e na cabeça da cláusula) devem estar instanciadas. Desta forma as cláusulas protegidas com guards podem ser reordenadas, sem que hajam problemas.

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

Se não usarmos «guards» a definição fica errada se trocarmos a ordem das cláusulas:

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

Eis algumas das funções que podem ser usadas em guards:

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

O «igual» e o «exactamente igual» merecem uma explicação, que pode ser dada através do seguinte exemplo: `1 == 1.0` é verdadeiro, mas `1=:= 1.0` é falso.

Uma das características do Erlang que pode confundir os principiantes é o facto de nem todas as funções poderem ser usadas em «guards». Em especial, nenhuma função definida pelo programador pode ser utilizada em «guards». Isto por razões que têm a ver com a performance do sistema. A validação da cabeça das cláusulas deve ser feita de uma forma rápida e sem efeitos laterais, o que invalida o uso de funções definidas pelo programador.

3.1.1 Ordenação

A ordenação das variáveis necessita de regras de comportamento. Os tuplos são ordenados primeiro pelo seu tamanho e depois pelos seus elementos. As listas são ordenadas pelas cabeças primeiro. Quando se comparam variáveis de tipos diferentes o comportamento do normal Erlang passa pela noção de variáveis de um determinado tipo são «maiores» do que variáveis de outro tipo, seguindo a seguinte ordem:

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

3.2 Recursividade

3.2.1 Introdução

A recursividade nos programas de computador é uma das coisas mais desconcertantes que um aprendiz de programador tem de enfrentar.

Por um lado nas aulas teóricas dizem-lhe que os programas recursivos são simples, elegantes, curtos e fáceis de analisar matematicamente. Por outro lado, nas aulas práticas o uso de programas recursivos é fortemente desaconselhado, uma vez que o seu uso nas linguagens normalmente impostas aos principiantes, conduz a sérios problemas de memória.

Aparecem normalmente erros de «*stack overflow*» que variam conforme o grau de recursividade, a linguagem utilizada, o compilador (ou as suas opções), o sistema operativo e a memória disponível.

Vamos ver que no Erlang esses problemas podem ser evitados através de técnicas simples de programação. Mas primeiro vamos aprender a fazer ciclos em Erlang.

3.2.2 Ciclos

Uma das perguntas mais incisivas do leitor pode ser a seguinte:

> *–Se o valor de uma variável não pode mudar depois de atribuído, como podemos fazer iterações do estilo:*
> `for I=1 to N do myfunc(I)`*?*

A resposta a esta pergunta pode ser dada usando a recursividade, do seguinte modo:

```
for(0) -> done;
for(N) -> myfunc(N),
          for(N-1).
```

Note-se que o código anterior executa a função N vezes mas pela ordem contrária do desejado, executa de N até 1. Nalguns casos a ordem de execução não é importante, apenas nos interessa o número de vezes que uma função é chamada.

Se a ordem de execução for importante então podemos usar a variante seguinte:

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

for(N,N) -> myfunc(N);
for(I,N) -> myfunc(I),
            for(I+1,N).
```

O padrão da recursividade é o mesmo nos dois casos. O segundo caso é mais pedagógico do que o primeiro porque temos uma estratégia interessante para resolver o problema. Temos duas funções `for` diferentes, porque têm um número diferentes de argumentos. A primeira função (`for/1`) que é a que queremos usar normalmente, apenas funciona como interface para a segunda, chamando-a com os parâmetros correctos.

A segunda função (`for/2`) funciona do seguinte modo:

- Os seus dois argumentos funcionam como o contador do ciclo e o limite do ciclo.
- Quando esses dois valores são iguais, a função `myfunc/1` é chamada e a função `for/2` termina.
- Se esses valores forem diferentes, a função `myfunc/1` é chamada e a seguir é chamada a função `for/2` com

o primeiro argumento (o contador) incrementado, passando para a iteração seguinte dessa forma.

Daqui a pouco vamos explicar como podemos usar a recursividade sem problemas, para já podemos dizer que este tipo de chamada recursiva pode ser feito em Erlang sem problemas.

3.2.3 Exemplos de recursividade

A recursividade em listas é muito comum, e é fácil encontrar exemplos da simplicidade do código que é proporcionada pelo encontro entre os algoritmos recursivos e as listas como estrutura de dados.

Como exemplos temos funções que calculam a média dos elementos de uma lista, a sua soma, o comprimento de uma lista, duplicam todos os elementos de uma lista, ou ainda numa função que vê se um valor faz ou não parte de uma lista:

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

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

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

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

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

O uso da recursividade traz consigo os problemas de «*stack overflow*» que podem ser evitado usando acumuladores. Os acumuladores são argumentos adicionais para as funções que

queremos, que servem para «ir acumulando» os valores que pretendemos obter no final, para evitar os problemas da recursividade.

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

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

Esta definição é ligeiramente mais complexa do que a anterior mas atravessa a lista apenas uma vez.

Note-se que temos duas funções diferentes! O segundo argumento da função `sum/2` desempenha o papel de acumulador. A função `sum/1` ao chamar a função `sum/2` inicializa o segundo argumento desta com o valor 0 (zero).

A cada chamada, a função `sum/2` adiciona ao seu segundo argumento, o primeiro elemento da lista, chamando-se a si própria para processar o resto da lista. Quando a lista está vazia, isso quer dizer que já somamos todos os elementos da lista, e o valor do segundo argumento já é a soma de todos os elementos da lista.

Exemplificando com `sum([1,2,3])`:

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

Note-se que através do uso de acumuladores a recursividade ficou mais «simples» uma vez que em cada chamada recursiva não existem cálculos pendentes. Uma vez que não existem cálculos pendentes, o Erlang pode utilizar uma técnica denominada «*tail call optimization*» para executar as sucessivas chamadas recursivas à mesma função num espaço de memória

constante. Além disso a lista é percorrida apenas uma vez pela função não havendo lugar ao «desfazer» da recursividade.

Como contraponto podemos ver a definição clássica:

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

Neste caso as sucessivas iterações serão as seguintes:

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

É fácil de ver que ficamos com cálculos pendentes em cada iteração, que poderão provocar um uso desenfreado da memória.

Temos de seguida mais alguns exemplos de como usar acumuladores:

```
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.



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

As chamadas «*tail recursive functions*» são importantes porque permitem ter funções recursivas que correm num espaço de memória constante, o que é importante não só em termos de memória mas também para a velocidade do programa.

Se usarmos acumuladores temos normalmente «*tail recursive functions*» que permitirão ao Erlang fazer «*tail call optimization*». Mais à frente iremos ver que os servidores em Erlang são escritos como funções recursivas, que devem ser obrigatoriamente «*tail recursive functions*».

3.2.4 Controle de fluxo

O Erlang possui as instruções `case` e `if` para o controle de fluxo dos programas. No entanto, o uso dessas instruções é desaconselhado uma vez que qualquer programa pode ser escrito sem elas através de cláusulas ou funções adicionais.

Eis um exemplo da sintaxe e uso da instrução do `case`:

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

Tem de ser garantir que um dos ramos do case deve ser seguido, senão vai ocorrer um erro de «*run-time*».

A sintaxe do `if` é parecida com a sintaxe do `case` sendo que num `if` podemos usar as mesmas expressões que nos «guards». Tal como no caso anterior um dos ramos tem de ser obrigatoriamente seguido senão vai ocorrer um erro de «*run-time*».

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

4 Processos

4.1 Introdução

A filosofia do Erlang baseia-se na existência de processos independentes que comunicam entre si através do envio de mensagens. Não existe partilha de dados, tudo o que for partilhado tem de ser passado como mensagem. Isto porque a partilha de dados é ineficiente (não pode ser feita em paralelo) e complicada, necessitando de «*locks*» para ser feita ser feita em segurança.

Cada processo tem um identificador (PID) que não se pode falsificar. Sabendo o identificador do processo pode-se mandar uma mensagem. Não existem garantias de entrega da mensagem, sendo que cada processo possui uma «caixa de correio» para as suas mensagens. Os processos em Erlang possuem um baixo peso, sendo que assim podemos usar um grande número de processos nos nossos programas.

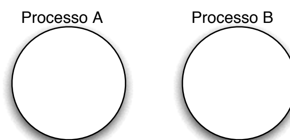

Figura 4.1: Dois processos

Código no Processo A para criar um novo Processo :

```
PidB=spawn(Módulo,Função,ListadeArgumentos)
```

A identidade do segundo processo (`PidB`) é apenas conhecida do Processo A, isto é do processo pai do processo que foi criado.

A chamada `spawn(Módulo,Função,ListadeArgumentos)` cria um novo processo, isto é uma nova linha de execução, começando na chamada da função fornecida pelos argumentos. Por exemplo o código seguinte cria um processo com a função `m:f(1,2,3)`:

```
Pid=spawn(m,f,[1,2,3])
```

Devemos ter o cuidado de exportar a função usada pela função `spawn/3` para criar o processo. Para um principiante na programação em Erlang isto conduz a erros difíceis de detectar uma vez que a função `spawn/3` nunca falha, mesmo que a função usada esteja errada.

Em condições normais, um processo termina quando não houver mais código para executar. Um processo também termina (como é lógico) quando aparecerem erros na sua execução.

4.2 Mensagens

4.2.1 Envio de mensagens

Os processos comunicam entre si enviando e recebendo mensagens. O envio de uma mensagem nunca falha (do ponto de vista do remetente). A mensagem pode ser qualquer termo legal em Erlang (como um àtomo, um tuplo ou lista). Normalmente é um tuplo em que um dos elementos do tuplo é um àtomo que serve de «etiqueta» para tornar mais claro o conteúdo da mensagem.

As mensagens são enviadas usando `!`, com o destino da mensagam antes do `!` e a mensagem a transmitir depois. No

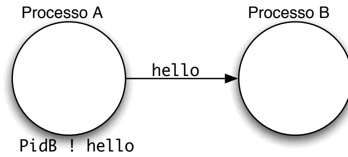

Figura 4.2: Envio de mensagens

caso da figura 4.2 `PidB` é o «*process id*» do processo de destino e a mensagem a ser transmitida é apenas o àtomo `hello`.

4.2.2 Recepção de mensagens

As mensagens são recebidas usando `receive`. A sintaxe é similar à sintaxe do `case`:

```
receive
{Pid, hello} ->
    io:format("~p: got hello from ~p~n",
                  [self(), Pid]),
    f();
{Pid,message,Msg} ->
    io:format("~p: got ~p from ~p~n",
                [self(), Msg, Pid]),
    g();
quit ->
    true
end
```

A instrução `receive` suspende o processo até que uma mensagem correspondente seja recebida. As mensagens que não correspondem vão sendo guardadas. Isto pode ser usado para a recepção selectiva de mensagens.

No caso da figura 4.3 a mensagem `hello` é recebida antes da mensagem `bye` qualquer que seja a ordem pela qual elas

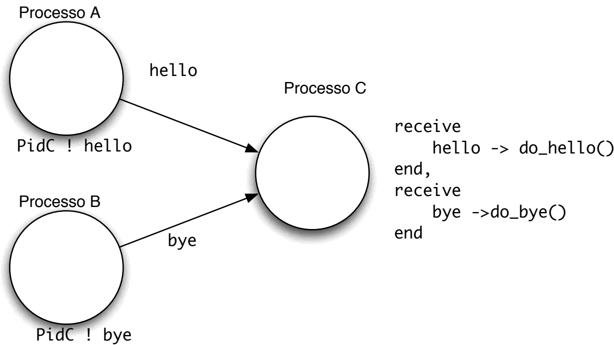

Figura 4.3: Recepção selectiva de mensagens

forem enviadas. Até agora apenas vimos mensagens constituídas apenas por átomos, mas estas podem ser mais complexas, para levarem dados para posterior processamento como podemos ver na figura seguinte.

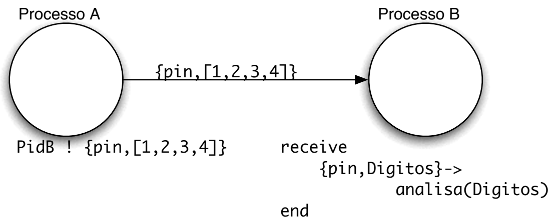

Figura 4.4: Mensagens com dados

Para a comunicação entre processos cada processo «pai» sabe os «*process ids*» dos filhos que criou, o que serve para o «pai»

comunicar com os «filhos». Para a transmissão de mensagens no sentido contrário é necessário fornecer aos «filhos» o «*process id*» do «pai». Para fazermos isto, temos a função `self()` que devolve o «*process id*» do processo que executou esta função. Assim um processo pode saber o seu *Pid* para transmitir a outro

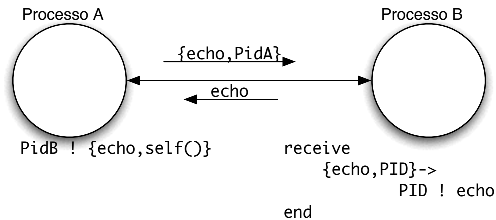

Figura 4.5: Mensagens nos dois sentidos

Um padrão comum de código para um servidor é o seguinte:

```
start() -> spawn(m,init,[...]).
init(...) -> <initialization>,
          loop(...).

loop(...) -> receive
    stop -> true;
    Pattern1 ->
        <actions>
        loop(...);
        ...
    PatternN ->
        <actions>
        loop(...)
end.
```

Eis um exemplo de um servidor que implementa um contador:

```
loop(Counter)-> receive
    stop->true;
    {inc,From} ->
        NewCounter=Counter+1,
        loop(NewCounter);
    {dec,From} ->
        NewCounter=Counter-1;
        loop(NewCounter);
    {query,From} ->
        From ! {value,Counter},
        loop(Counter).
```

Eis um exemplo do «eco» entre dois processos:

```
-module( echo).
-export([ start/0]).
-export([ pid1/1, pid2/0]).
start() -> Pid2 = spawn(echo, pid2, []),
    spawn( echo, pid1, [Pid2]).
pid1(Pid2) ->
    Pid2 ! {self(), hello},
    receive
    {Pid2, Msg} ->
        io:format(" P1 got echo from P2~n", []),
        Pid2 ! stop
end.

pid2() ->
    receive
    {Pid1, Msg} -> Pid1 ! {self(), Msg},
      pid2();
    stop ->
    true
end.
```

4.2.3 Processos registados

Se quisermos que um processo fique acessível a todos os outros do ponto de vista das comunicações, a forma de fazer isso mais fácil é registar esse processo.

O registo consiste em associar um nome a esse processo, através da função `register(Nome,Pid)` que regista o processo `Pid` com o nome global `Nome`. Assim qualquer processo pode enviar uma mensagem a um processo registado, uma vez que pode enviar para enviar uma mensagem para o seu nome em vez do *Pid*.

Exemplo:

```
...
register(ernie,Pid),
...
...
ernie ! Hello
....
```

4.2.4 Prazos de recepção

Na recepção de mensagens podemos impor um prazo máximo de tempo para a instrução `receive` aguardar pelas mensagens. O intervalo de tempo máximo (ou «*timeout*») é especificado em milisegundos.

Exemplo:

```
receive
    hello ->  io:format("hello",[])
    after 1000 -> true
end
% 1000 ms = 1 Segundo
```

Note-se também o uso de um comentário no excerto de código anterior. Um comentário em Erlang vai desde o `%` até ao fim da linha.

Entre os vários usos possíveis dos prazos máximos de recepção encontram-se a suspensão de processos ou a activação de alarmes.

```
%%% sleep(T) - processo suspenso durante T ms.
sleep(T) ->
    receive
    after
    T ->true
end.

%%% set_alarm(T,Alarm) - A mensagem Alarm é
%%% enviada ao processo que o activou
%%% daí a T ms.

set_alarm(T,Alarm) ->
   spawn(timer,alarm,[self(),T,Alarm]).

alarm(Pid,T,Alarm) ->
    receive
        after T  ->
        Pid ! Alarm
    end.
```

5 Erros

5.1 Filosofia dos erros em Erlang

A ideia subjacente ao tratamento de erros em Erlang é deixar que os processos «estourem» o mais cedo possível. Isto pode parecer irresponsável numa primeira aproximação, mas como vamos ver é fundamental para a fiabilidade do sistema. Nós só conseguimos construir um sistema fiável se soubermos onde estão os erros e se estes forem detectados. Ao assumir conscientemente que os processos podem «estourar», estamos a aumentar a robustez do sistema. Se partíssemos do pressuposto que os processos nunca «estouram» estaríamos a aumentar a insegurança do sistema.

Ao tornar visíveis e detectáveis os erros estamos a aumentar a robustez do sistema, porque os problemas mais graves não acontecem quando sabemos que um sistema tem problemas, mas sim quando aparentemente está tudo bem.

5.2 *Links*

5.2.1 Introdução

Os processos podem ser ligados uns aos outros (um a um).

Os *links* são criados explicitamente usando `link(Pid)` ou então usando `spawn_link(Módulo,Função,Argumentos)` quando se cria o processo pretendido. Os links são bidireccionais. Podem ser removidos usando `unlink(Pid)`.

Figura 5.1: Processos ligados entre si

5.2.2 Sinais de fim

Quando um processo termina, são enviados sinais de fim (*exit*) a todos os processos aos quais o processo esteja conectado por *links*.

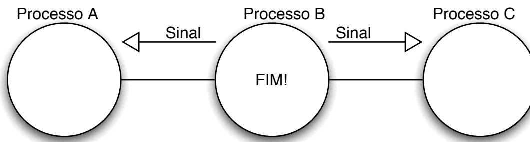

Figura 5.2: Fim de um processo

Note-se a diferença entre mensagens e sinais. Uma mensagem vai de um processo origem para um processo de destino, quando no processo de origem há um envio de uma mensagem para o processo de destino. O envio de uma mensagem é sempre de um único processo para um único processo.

Um sinal de fim é enviado sempre que um processo termina para todos os processos a que este se encontra ligado através de *links*. O sinal pode não ser enviado para nenhum processo, se o processo que terminou não estiver ligado a nenhum pro-

cesso. Por outro lado, os sinais podem ser enviados para mais do que um processo se o processo que terminou, se encontrar ligado a vários processos.

Um processo pode terminar por três motivos distintos:

- Normalmente (não tem mais código para executar).
- Devido a um erro de run-time: uma chamada errada a uma função,uma instanciação errada, etc..
- Quando encontrar uma instrução para terminar.

Quando um processo termina de uma forma anormal, envia sinais de fim a todos os processos com os quais possui links:

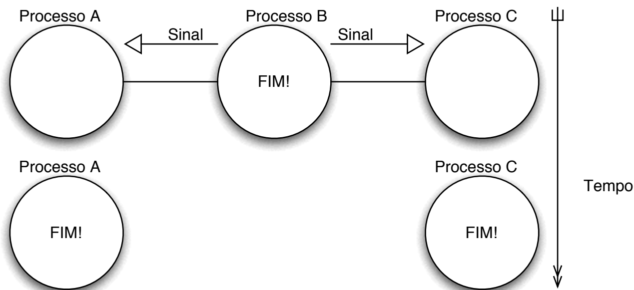

Figura 5.3: Fim de um processo

Os processos ligados também terminam, passando sinais de fim aos que se encontram ligados, propagando dessa forma os sinais de fim.

Desta forma (para já) conseguimos que todo o nosso sistema se desmorone como um castelo de cartas, quando acontecer um erro, o que aumenta a visibilidade do erro. Vamos ver mais à frente usos mais construtivos para os links.

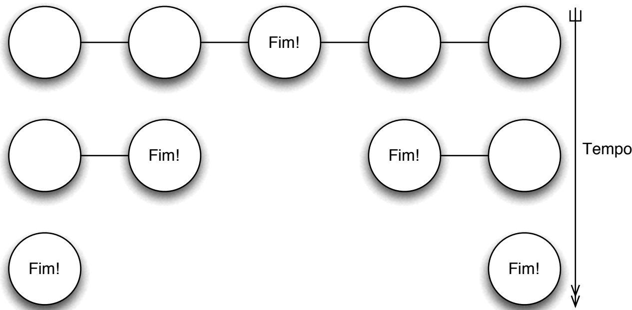

Figura 5.4: Fim de vários processos em cascata

5.2.3 A função *exit*

Na realidade existem duas funções: `exit/1` e `exit/2`. A primeira termina um processo emitindo um sinal de fim com uma determinada razão especificada no seu argumento.

```
exit(Razão)
```

Esta função termina sempre o processo que a executa, e emite sinais com a razão especificada para todos os processos (zero ou mais) a que o seu processo esteja ligado por *links*.

A função `exit/2` tem um comportamento diferente uma vez que emite um sinal de fim com uma razão especificada para um único processo do qual epecificamos o *Pid*.

```
exit(Pid,Razão)
```

A função `exit/2` não tem efeitos no processo que a chama, não terminando o processo que a chama.

Os processos não precisam de estar conectados por *links* para usarmos esta função, e esta função envia sempre um sinal para um único processo.

Tipos das razões para `exit/2`:

- `normal` – Emitida quando um processo termina normalmente. Sinais de fim com a razão `normal` não se propagam. Isto é, um processo que receba um sinal destes não termina, nem o envia para outros.
- `kill` – Tem de ser emitida usando `exit/2`. Termina o processo que recebe incondicionalmente. O processo que o recebe, envia sinais com a razão `killed`.
- `Outros` – Qualquer dos outros tipos é uma razão anormal, e propaga-se sem que a sua razão se altere.

5.3 Detecção de erros

5.3.1 Funcionamento

Os processos podem detectar sinais de fim, através da seguinte instrução:

```
process_flag(trap_exit,true)
```

Assim, os sinais de exit são transformados em mensagens, com a seguinte forma:

```
{'EXIT',Pid,Razão}
```

Desta forma um processo recebe mensagens em vez de sinais, podendo processar as mensagens da forma que entender. Passa assim a ser avisado sempre que um processo que se encontre ligado a ele termine, e pode efectuar as acções correctivas necessárias.

Um processo termina de uma forma anormal, enviando sinais de fim. Mas dos processos ligados a ele, o processo sombreado tem a detecção de erros activa:

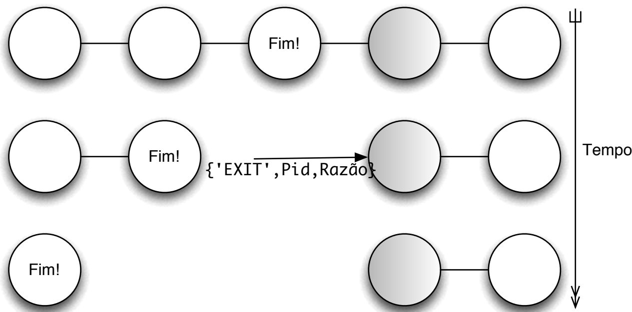

Figura 5.5: Fim de um processo com detecção de erros activa

Enquanto um processo termina propagando o sinal de fim, o outro irá apenas receber uma mensagem, que terá a forma {'EXIT',Pid,Razão}. Este último não irá propagar o sinal.

Assim podemos fazer uma tabela que nos ajuda a perceber o que acontece quando um processo sem detecção de erros, recebe um sinal:

Razão	**Comportamento do processo**
normal	Ignora o sinal
kill	Morre e emite sinal com razão killed
Outra	Morre e emite sinal com razão Outra

Em paralelo podemos fazer a mesma tabela para processos com a detecção de erros activa:

Razão	**Comportamento do processo**
normal	Recebe a mensagem {'EXIT',Pid,normal}
kill	Morre e emite um sinal com a razão killed
Outra	Recebe a mensagem {'EXIT',Pid,Outra}

Assim é fácil a um processo vigiar o estado de outros, uma vez que basta que o processo tenha detecção de erros activa e possua um *link* para cada um dos processos a «vigiar». Desta forma irá ser avisado sempre que um processo «vigiado» por si termine, seja de uma forma anormal ou não.

Podemos assim ver como pode ser feita a estrutura em camadas de um sistema robusto, organizando-o em camadas de «supervisores» que terão a detecção de erros activa, e «trabalhadores» que serão «vigiados» pelos seus supervisores, que terão a responsabilidade de os reinicializar assim que for necessário.

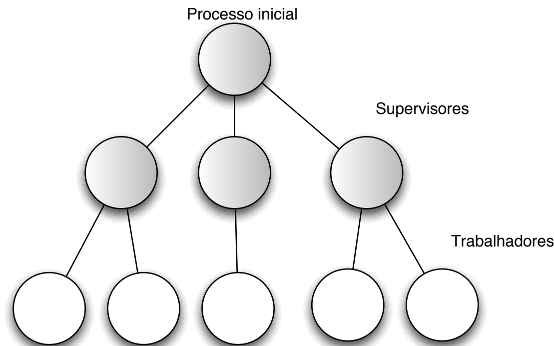

Figura 5.6: Estrutura de um sistema robusto

5.3.2 Um servidor robusto

Vamos agora analisar em detalhe como podemos estruturar o código de um servidor de recursos robusto. Por razões de simplicidade do código vamos supor que os recursos são todos iguais entre si.

Esta simplificação permite que no pedido de um recurso, não seja necessário especificar o recurso pretendido, e que o servidor possa atribuir a um cliente um recurso qualquer (desde que este esteja livre, como é óbvio).

```
-module(robust_server).
-export([start/1,init/1]).

start(Resources) ->
    spawn(robust_server,init,[Resources]).

init(Resources) ->
    process_flag(trap_exit, true),
    loop(Resources,[]).
```

Este módulo chama-se `robust_server` e as funções exportadas são as funções `start/1` e `init/1`. A primeira serve de interface para o servidor, porque é a função que arranca o servidor, e a segunda é exportada para que possamos fazer o seu `spawn`. Têm as duas um único argumento que é a lista de recursos que o servidor tem para alocar pelos seus clientes.

A primeira coisa que o processo servidor faz é assinalar que quer receber mensagens em vez de sinais, dos processos que ficarem ligados a ele[1]. De seguida começa o ciclo principal do nosso servidor, que possui dois argumentos, sendo o primeiro a lista dos recursos livres e o segundo a lista dos recursos alocados pelos clientes, estando a segunda vazia quando se arranca o servidor.

[1]Um dos erros mais comuns no início do trabalho com processos, é colocar esta instrução no processo errado. Os principiantes têm alguma tendência a colocar esta instrução antes do arranque do processo, ficando a detecção de sinais de fim, activa no processo «pai» do processo pretendido.

```
loop(Free,Allocated) ->
receive
  {alloc,Pid}->{Resource,NewFree,NewAllocated} =
             alloc(Pid,Free,Allocated),
             Pid ! {resource,Resource},
             link(Pid),
             loop(NewFree, NewAllocated);
  {free,Resource,Pid} -> {NewFree,NewAllocated} =
             free( Pid, Resource,Free,Allocated),
             unlink( Pid),
             loop(NewFree,NewAllocated);
  {'EXIT',Pid,_Reason} ->{NewFree,NewAllocated} =
             free_all(Pid,Free,Allocated),
             loop(NewFree,NewAllocated)
end.
```

O servidor comporta-se de acordo com as mensagens que receber:

- `{alloc,Pid}` O servidor aloca o recurso ao processo que pediu um recurso, calculando os novos recursos livres e alocados (2 listas), envia o recurso para o processo que o pediu e cria um *link* para o processo cliente.
- `{free,Resource,Pid}` O servidor liberta o recurso devolvido pelo cliente, e desfaz o *link* para o cliente.
- `{'EXIT',Pid,_Reason}` Se o servidor receber esta mensagem, isto quer dizer que o cliente «estourou» por qualquer razão. Neste caso o servidor deve libertar todos os recursos que o processo cliente possuía.

```
alloc(Pid,[Resource|Rest],Allocated) ->
    {Resource,Rest,[{Pid,Resource}|Allocated]}.
```

Alocar um recurso (supondo que todos são iguais) é retirá-lo da lista dos livres, e juntar à lista dos recursos alocados um tuplo de dois elementos em que o primeiro elemento é o *process*

id do processo em causa e o segundo elemento é o recurso alocado. Assim podemos saber a que processo é que cada recurso está alocado.

```
free(Pid,Resource,Free,Allocated) ->
  NewAllocated=lists:delete({Pid, Resource}, Allocated),
  {[Resource|Free],NewAllocated}.
```

Libertar um recurso da lista de processos é apagar a entrada correspondente na lista de recursos alocados, e adicionar o recurso à lista dos recursos livres.

```
free_all(Pid,Free,Allocated) ->
   free_all(Pid,Free,[],Allocated).

free_all(Pid,Free,Allocated,[{Pid,Resource}|Rest]) ->
   free_all(Pid,[Resource|Free],Allocated,Rest);
free_all(Pid,Free,Allocated,[First|Rest]) ->
   free_all(Pid,Free,[First|Allocated],Rest);
free_all(Pid,Free,Allocated,[]) ->
   {Free,Allocated}.
```

A função `free_all/3` é a mais complexa mas também a mais pedagógica. Para retirarmos da lista dos recursos alocados todos os recursos do `Pid` que recebemos, vamos ter de percorrer a lista dos recursos alocados, e tirar dessa lista para a lista dos recursos livres todos os que pertencem ao `Pid` que recebemos.

Para isso usamos uma função auxiliar `free_all/4`, que nos simplifica o código. A primeira função possui três argumentos:

- O `Pid` do processo em causa.
- A lista dos recursos livres (apenas recursos).

- A lista dos recursos alocados. Cada elemento dessa lista é um tuplo de dois elementos com o `Pid` do processo e o recurso alocado.

No caso da segunda função os quatro argumentos são os seguintes:

- O `Pid` do processo em causa.
- A lista dos recursos livres (apenas recursos).
- A lista dos recursos alocados já «examinados».
- A lista dos recursos alocados «por examinar».

Note-se que a variável `Allocated` da primeira função não é a mesma variável (com o mesmo nome) que aparece na segunda função. Mais uma vez: os nomes da variáveis são «locais» a cada cláusula, não podendo o programador «confiar» no nome das variáveis, entre duas cláusulas diferentes, e muito menos entre duas funções diferentes!

O comportamento da segunda função é então o seguinte (cláusula a cláusula):

1. Se à cabeça da lista por processar, estiver um recurso alocado ao processo, essa alocação desaparece da lista dos recursos alocados «por examinar» e o recurso passa para a lista dos recursos livres.
2. Se entramos nesta cláusula, isto quer dizer que a primeira não é válida. Se à cabeça da lista «por examinar», não está um recurso alocado ao processo, este recurso deve passar para a lista dos recursos já «examinados».
3. Se a lista dos recursos «por examinar» estiver vazia, isso significa que já examinamos os recursos todos, e a nossa função deve terminar, devolvendo a nova lista de recursos livres e alocados.

6 Extensões

6.1 Records

Os «records» em Erlang são o equivalente dos «structs» em C ou dos «Records» em Pascal, permitindo referenciar os elementos de uma estrutura de dados pelo seu nome. Os tuplos permitem referenciar pelo seu número (por exemplo):

```
Nome=elem(1,Tuplo)
```

Mas se mudamos a ordem dos elementos, retiramos um elemento, ou adicionamos um elemento, temos de alterar o código. Os «records» permitem que os seus elementos sejam referenciados pelo nome o que ajuda na legibilidade e manutenção do código.

Podemos ver o seguinte exemplo de definição de um «record»:

```
-record(pessoa,{nome,telefone,endereço}).
```

Se P fôr uma variável do tipo pessoa, podemos usar a seguinte sintaxe:

```
Nome=P#pessoa.nome,
Endereço=P#pessoa.endereço,
...
```

Um «record» é definido da seguinte maneira:

```
-record(NomedoRecord,
     {Campo1[=ValorPorDefeito1],
      Campo2[=ValorPorDefeito2],
     ...
     CampoN[=ValorPorDefeitoN]}).
```

O nome do record e os nomes dos campos devem ser átomos. Os valores por defeito (opcionais) são usados se ao criar um «record» não dermos valores aos campos. Se não existir um valor por defeito o campo assume o valor `undefined`.

Exemplo de definição de um «record»:

```
-record(pessoa,{nome="",tel=[],ender}).
```

Se o record fôr usado em vários módulos a sua definição deve ser colocada num ficheiro com a extensão `.hrl` e os módulos que usam essa definição devem incluir esse ficheiro. Como exemplo temos a seguinte linha:

```
-include("os_meus_records.hrl").
```

Como é óbvio a definição de um «record» deve vir antes da sua criação e do seu uso.

Um record é criado da seguinte maneira:

```
#NomeDoRecord{Campo1=Valor,
              ...,
              CampoM=ValorM}.
```

Se algum dos campos fôr omitido, é usado o valor por defeito.

Exemplo de criação de um «record»:

```
P=#pessoa{tel=[2,2,8,3,4,0,5,0,0],nome="José"}.
```

No uso de records podemos seleccionar apenas um campo:

```
Variavel#nome_do_record.campo
```

Exemplo:

```
P#pessoa.tel
```

Podemos alterar apenas o valor de um campo (na passagem para uma nova variável):

```
P2=P1#pessoa{tel=[2,2,8,3,4,0,5,0,1]}
```

Podemos ainda testar se um record é ou não de um dado tipo:

```
record(Variável,Tipo)
```

Exemplo:

```
funcao(P) when record(P,pessoa)->pessoa;
funcao(_)->nao_e_pessoa.
```

6.1.1 Representação dos «Records»

Os records são representados internamente em Erlang como tuplos, em que o primeiro membro do tuplo é o nome do tipo de «record». Existe uma função que nos ajuda a trabalhar melhor com records:

`record_info(fields,Record)`	lista dos campos
`record_info(size,Record)`	número de campos
Como exemplo:	
`record_info(fields,pessoa)`	devolve [nome,tel,ender]
`record_info(size,pessoa)`	devolve 3

Exemplos do uso de records

```
%% File: person.hrl
%%-----------------------------------------------
%% Data Type: person
%% where:
%% name: A string (default is undefined).
%% age: An integer (default is undefined).
%% phone: A list of integers (default is []).
%% dict: A dictionary containing various information
%% about the person.
%% A {Key, Value} list (default is the empty list).
%%-----------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

```
%% File: person.erl
-module(person).
-include("person.hrl").
-compile(export all). % For test purposes only.

%% This creates an instance of a person.
%% Note: The phone number is not supplied so the
%% default value [] will be used.
make_hacker_without_phone(Name, Age) ->
            #person{name = Name, age = Age,
            dict = [{computer_knowledge, excellent},
            {drinks, coke}]}.

%% This demonstrates matching in arguments
print(#person{name = Name, age = Age,phone = Phone,
        dict = Dict}) - >
io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
"Dictionary: ~w.~n", [Name, Age, Phone, Dict]).

%% Demonstrates type testing, selector, updating.
birthday(P) when record(P, person) ->
            P#person{age = P#person.age + 1}.

register_two_hackers() ->
        Hacker1=make_hacker_without_phone("Joe", 29),
        OldHacker=birthday(Hacker1),
% The central register server should have
% an interface function for this.
        central_register_server!
        {register_person, Hacker1},
        central_register_server ! {register person,
        OldHacker#person{name = "Robert",
        phone = [0,8,3,2,4,5,3,1]}}.
```

6.2 Operações com Listas

Como forma mais simples de escrever a concatenação de duas listas e a subtracção de duas listas, temos os operadores ++ e --. O operador ++ efectua a concatenação das duas listas. O operador -- produz uma lista que é o resultado da primeira lista, depois de retiradas as primeiras ocorrências de cada um dos elementos da segunda lista.

Deste modo se um elemento da segunda lista aparecer mais do que uma vez na primeira lista, só a sua primeira ocorrência é que é retirada, mantendo-se as outras inalteradas. Exemplos:

```
L4=L1 ++ L2 ++ L3
L5=L1--L2
L6=[1,1,2,2,2] -- [1,2]
```

No último caso a lista `L6` será a lista `[1,2,2]`.

No caso de utilizarmos uma mistura destes operadores, devemos usar parênteses para garantir a sequência das operações, uma vez que não é garantido que estas sejam efectuadas da esquerda para direita.

6.3 Compreensão de Listas

O mecanismo de compreensão de listas permite-nos uma notação sucinta para gerar os elementos de uma lista. Corresponde à noção de «*Set Compreension*» na teoria dos conjuntos de Zermelo-Frankel, e às expressões ZF nas linguagens de programação Miranda e SASL. São ainda similares[1] ao `setof` e `findall` na linguagem Prolog.

A sua sintaxe é a seguinte, observando-se as seguintes regras:

```
[Expressão || Qualificador1, Qualificador2, ... ]
```

[1]Similares quer dizer parecido mas não quer dizer igual!

- Expressão é uma expressão arbitrária e cada qualificador pode ser um Gerador ou um Filtro.
- Um Gerador é escrito como Padrão<-ListaExpr.
- ListExpr deve ser uma expressão que dá uma lista de termos.
- Um Filtro é um predicado ou uma expressão booleana.
- Um predicado é uma função que devolve os átomos `true` ou `false`.

Como de costume, isto entende-se melhor através do uso de alguns exemplos:

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].
[a,4,b,5,6]
```

Isto deve ler-se: «A lista dos X tais que X é retirado da lista [1,2,...] e X é maior do que 3».

- `X <- [1,2,a,3,4,b,5,6]` é um gerador
- `X > 3` é um filtro

Podemos adicionar mais filtros:

```
>[X || X <- [1,2,a,3,4,b,5,6],integer(X),X>3].
[4,5,6]
```

Ou podemos combinar geradores para ter o produto cartesiano de duas listas:

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

Uma aplicação interessante da compreensão de listas pode-se encontrar na seguinte forma de codificar em Erlang o algoritmo de ordenação «*Quick Sort*».

```
sort([Pivot|T]) ->
       sort([ X || X <- T, X < Pivot]) ++
       [Pivot] ++
       sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

Outro exemplo interessante é a geração das permutações de elementos de uma lista:

```
perms([]) -> [[]];
perms(L) -> [[H|T] || H <- L, T <- perms(L--[H])].
```

Chamam-se triplos pitagóricos aos conjuntos de três números inteiros que cumprem a condição: $A^2 + B^2 = C^2$ A função `pyth(N)` gera a lista dos inteiros que cumprem essa condição para $A + B + C <= N$. Exemplo mais simples[2]:

```
pyth(N) ->[ {A,B,C} ||
    A <- lists:seq(1,N),
    B <- lists:seq(1,N),
    C <- lists:seq(1,N),
    A+B+C =< N,
    A*A+B*B == C*C ].
```

Podemos usar uma forma mais eficiente:

```
pyth1(N) ->[{A,B,C} ||
    A <- lists:seq(1,N),
    B <- lists:seq(1,N-A+1),
    C <- lists:seq(1,N-A-B+2),
    A+B+C =< N,
    A*A+B*B == C*C ].
```

Na compreensão de listas há algumas regras sobre o uso das variáveis que temos de observar.

- Todas as variáveis que ocorrem num gerador são variáveis *novas* («frescas»).

[2]A função `lists:seq/2` gera uma lista com uma sequência de números inteiros desde o seu primeiro argumento até ao segundo.

- Todas as variáveis que estavam definidas antes da compreensão de listas e que são usadas em filtros possuem o valor que tinham antes.

- Não se pode exportar o valor de nenhuma variável.

Exemplo: Escrever a função `select` que selecciona certos elementos de uma lista de tuplos.

```
select(X,L)->[Y||{X,Y}<-L].
```

A nossa ideia era fazer uma lista com todos os `Y` da lista `L` que constituem os segundos elementos dos tuplos onde `X` é o primeiro elemento. Mas ao compilar o programa aparece-nos a seguinte mensagem:

```
./FileName.erl:Line: Warning:
        variable 'X' shadowed in generate
```

O nosso problema é que o `X` no padrão não é o mesmo `X` da cabeça da função. Se usarmos a função o resultado não vai ser aquele que esperávamos:

```
>select(b,[{a,1},{b,2},{c,3},{b,7}]).
[1,2,3,7]
```

Podemos corrigir o erro da seguinte forma:

```
select(X, L) -> [Y || {X1, Y} <- L, X == X1].
```

O gerador contém apenas variáveis não instanciadas, e o teste é feito no filtro, e agora tudo funciona:

```
>select(b,[{a,1},{b,2},{c,3},{b,7}]).
[2,7]
```

Como conclusão podemos observar que certas operações de «*pattern matching*» devem estar nos filtros e não podem ficar nos geradores. Não devemos escrever:

```
f(...) -> Y = ...,
[ Expressão || PadrãoComY <- Expr, ...]
...
```

Mas devemos escrever antes:

```
f(...) -> Y = ...,
[ Expressão || PadrãoComY1 <- Expr,Y == Y1,...]
...
```

6.4 Macros

Os macros podem ser escritos em Erlang usando a seguinte sintaxe:

```
-define(Constante,Valor).
-define(fun(Var1,Var2,...,Var),Substituição).
```

Os macros são expandidos quando se usa o nome do macro com um ponto de interrogação antes na forma `?Nome_do_Macro`. Se definirmos por exemplo:

```
-define(timeout,200).
```

A expressão `?timeout` será então substituída por `200` sempre que aparecer. Podemos ter macros com argumentos, como por exemplo:

```
-define(macro1(X,Y),{a,X,b,Y})
```

Podemos usar este macro da seguinte forma:

```
bar(X)->
  ?macro1(a,b),
  ?macro1(X,123).
```

Este macro será expandido para:

```
bar(X)->
  {a,a,b,b},
  {a,X,b,123}.
```

Encontram-se já pré-definidos os seguintes macros:

`?MODULE`	– nome do módulo
`?FILE`	– nome do ficheiro
`?LINE`	– número da linha corrente
`?MACHINE`	– nome da máquina virtual[3]

Podemos usar macros condicionais em Erlang, estando definidas as seguintes directivas:

`-undef(Macro).`	- remove a definição do Macro
`-ifdef(Macro).`	- faz as linhas seguintes se Macro estiver definido
`-ifndef(Macro).`	- faz as linhas seguintes se Macro não estiver definido
`-else.`	- macro de «else».
`-endif.`	- macro de «endif».

Os macros condicionais usam-se normalmente agrupados da seguinte forma:

```
-ifdef(debug).
-define(...).
-else.
-define(...).
-endif.
```

Podemos ver isso no seguinte exemplo:

```
-define(debug, true).
-ifdef(debug).
-define(trace(Str, X),
  io:format("Mod:~w line:~w ~p ~p~n",
               [?MODULE,?LINE,Str,X])).
-else.
-define(trace(X, Y), true).
-endif.
```

Dadas estas definições a expressão `?trace(?X=?,X).` na linha 100 do módulo foo, será expandida para:

```
io:format("Mod:~w line:~w ~p ~p~n",[foo,100,"X=",[X]]).
```

Se removermos o define de debug, a expressão será expandida para `true`.

Dado que a expansão de macros pode trazer consequências imprevistas, convém ter uma forma de ver qual o resultado do seu funcionamento. Uma forma de expandir macros é através do seguinte código:

```
-module(mexpand).
-export([file/1]).
-import(lists, [foreach/2]).
file(File) ->
    case epp:parse_file(File ++ ".erl", [],[]) of
    {ok, L} ->
{ok, Stream} = file:open(File ++ ".out", write),
            foreach(fun(X) ->
    io:format(Stream,"~s~n",[erl_pp:form(X)])end,L),
file:close(Stream)
end.
```

Outra forma de expandir macros é compilar o ficheiro com a opção `'P'`. A linha de comandos seguinte produz uma listagem com o nome `Ficheiro.P` em que podemos ver o resultado de qualquer expansão de macros.

```
compile:file(Ficheiro,['P'])
```

6.5 Binários

Um binário (*Bin*) é em Erlang uma sequência de bytes de baixo nível, que pode se utilizada para modelar por exemplo pacotes de transmissão de dados.

Temos o seguinte exemplo da sua sintaxe:

```
Bin=<<E1,E2,...En>>
```

Pontos importantes dos binários:

- Cada elemento especifica um certo segmento do binário.
- Um segmento é um conjunto de bits contíguos.
- A fronteira dos segmentos não se encontra alinhada ao byte.
- Um binário tem um número certo de bytes.

Exemplos do uso de Binários:

```
Bin1 = <<1,17,42>>
Bin2 = <<"abc">>
A=1, B=17, C=42, Bin3=<<A,B,C:16>>
```

Se fizermos:

```
<<D:16,E,F/binary>>=Bin3
```

isto dará:

```
D=273, E=00, F=<<42>>
```

Isto é:

- `D` será os dois primeiros bytes de Bin3, O primeiro tem o valor 1, o segundo o valor 17, logo D vale 273 que é 1*256+17.
- `E` será o terceiro byte de Bin3 que vale zero, porque corresponde à parte mais significativa de C que tem 16 bits.
- `F` é um binário que vale 42, porque é esse o valor do byte menos significativo de C (e também o valor C por acaso, uma vez que o valor de C cabe em 8 bits).

Na especificação de segmentos podemos detalhar a composição de cada segmento, uma vez que cada segmento tem a seguinte especificação:

```
Valor:Tamanho/Lista_especificação_tipos
```

O único campo obrigatório é o tamanho e a lista de especificação dos tipos é uma lista de tipos separada por - com os seguintes componentes:

Tipo:	`integer`, `float` ou `binary`
Sinal:	`signed` ou `unsigned`
«*Endianness*»:	`big` ou `little`
Unit:	tamanho de cada unidade que vai ser multiplicado por Tamanho para dar o tamanho real.

Vamos ver alguns exemplos do uso da especificação de binários:

`X:4/signed-integer-unit:8`	Um elemento com um tamanho de 32 bits, e que contém um inteiro com sinal.
`<<X:1,Y:6>>`	Dá erro, um binário deve ter no total um número certo de bytes.
`<<X:1,Y:6,Z:1>>`	Já dá um número certo de bytes (1 byte = 8 bits).
`<<X+1:8>>`	Dá erro (problema do compilador).
`<<(X+1):8>>`	É a forma de escrever a expressão anterior que o compilador aceita.
`<<"hello">>`	É equivalente a `<<$h,$e,$l,$l,$o>>`.

Se quisermos «retirar» de um binário o resto do binário, retirando um dado cabeçalho podemos usar a seguinte sintaxe:

```
<<A:6,B:2,Resto/binary>>
```

Para que isto funcione temos de ter atenção porque Resto tem de ter um número certo (inteiro) de bytes, para poder ser um binário.

Um exemplo mais complexo é a definição de um pacote TCP/IP:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).
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/binary>>
when HLen >= 5, 4*HLen =< DgramSize ->
OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
<<Opts:OptsLen/binary,Data/binary>> = RestDgram,
...
end.
```

Um pacote TCP/IP é constituído pelos seguintes campos:

Versão do protocolo	4 bits
Comprimento do cabeçalho	4 bits
Tipo de serviço	8 bits
Comprimento total do pacote	16 bits
Identificador (para a fragmentação)	16 bits
Flags (para a fragmentação)	3 bits
Offset do fragmento	13 bits
«Time To Live»	8 bits
Protocolo	8 bits
Checksum do cabeçalho	16 bits
Endereço de origem	32 bits
Endereço de destino	32 bits
Opções e vários	Tamanho variável
Dados	Tamanho Variável

Como o comprimento do cabeçalho (que é dado em múltiplos de 32 bits) inclui as opções que possuem um tamanho variável, para saber o tamanho das opções temos de descontar do total do cabeçalho o tamanho mínimo de um cabeçalho TCP/IP. Assim podemos extrair um binário que contém apenas os dados.

6.6 Objectos Funcionais

Os objectos funcionais (funs) são um novo tipo de dados introduzido na versão 4.4 do Erlang. Podemos passar funs como argumentos a uma função. Podemos escrever funções que devolvem funs. As funs permitem encapsular padrões comuns de design em formas funcionais chamadas funções de ordem elevada. Ficamos com programas mais curtos e mais claros (se os percebermos!).

Para ilustrar o que são funs vamos tentar encontrar um ponto comum entre duas funções diferentes. Se quisermos escrever uma função que duplique o valor de cada um dos elementos de uma lista, o nosso código será o seguinte:

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

Se quisermos uma função que some um a cada um dos elementos de uma lista, o nosso código será o seguinte:

```
add_one([H|T]) -> [H+1|add_one(T)];
add_one([]) -> [].
```

Estas funções possuem muito em comum, podemos escrever uma função `map/2` que exprime o comum nas duas funções:

```
map(F,[H|T]) -> [F(H)|map(F,T)];
map(F,[]) -> [].
```

A função chama-se `map` porque pega nos elementos de uma lista, e «mapeia» os seus elementos noutra através da aplicação de uma função a cada um dos elementos. Podemos então reescrever as funções do seguinte modo (entre outros):

```
double(L) -> map(fun(X) -> 2*X end, L).
add_one(L) -> map(fun(X) -> 1 + X end, L).
```

Repetindo, `map(F,L)` é uma função que aceita uma função `F` e uma lista `L` como argumentos e devolve a nova lista que é obtida aplicando `F` a cada um do elementos de `L`. O processo que nos permite «extrair» o que há de comum em diferentes programas chama-se «abstracção procedimental».

Esta técnica de programação serve para escrever diferentes funções, reutilizando código comum da seguinte forma:

1. Escrevendo uma função que representa as partes comuns.

2. Exprimindo as diferenças em termos de funções que são passadas como argumento.

Vamos a outro exemplo. Se quisermos imprimir todos os elementos de uma lista num «*stream*», utilizaremos o seguinte código:

```
print_list(Stream, [H|T]) ->
          io:format(Stream, "~p~n", [H]),
          print_list(Stream, T);
print_list(Stream, []) ->true.
```

Se quisermos enviar uma mensagem a uma lista de processos, o código será o seguinte:

```
broadcast(Msg, [Pid|Pids]) ->
                  Pid ! Msg,
          broadcast(Msg, Pids);
broadcast(_, []) ->true.
```

Nos dois casos anteriores a estrutura das funções é similar. As duas funções usam cada um dos elementos da lista para fazer «alguma coisa». Ao contrário das funções anteriores, o resultado da execução da função em cada um dos elementos da lista não nos interessa, apenas nos interessando que a função seja executada.

Assim, vamos criar uma função chamada `foreach` ou «para cada um», cujo código será o seguinte:

```
foreach(F, [H|T]) ->
          F(H),
          foreach(F, T);
foreach(F, []) -> ok.
```

Poderemos então escrever em substituição:

```
print_list(S,L)->
  foreach(fun(X) -> io:format(S,"~p~n~,[X]) end,L).

broadcast(M,L)->foreach(fun(Pid) -> Pid ! M end,L).
```

As vantagens desta técnica de programação são as seguintes:

- O programa fica mais curto (é óbvia).
- As funções que aceitam funs como argumentos são muito mais fáceis de reutilizar.
- O programa fica mais legível (se o soubermos ler).
- A intenção do programador fica expressa no código.

Na lista anterior, a última vantagem necessita de uma certa explicação. Enquanto que no código original tínhamos todas as acções necessárias para fazer o pretendido, no novo código temos apenas coisas do estilo: «pegar nesta lista e mapear esta função em todos os seus elementos» ou «para cada um dos elementos desta lista fazer isto». Assim o que fica no código é

mais próximo das intenções do programador, ou se quisermos dizer o mesmo de outra maneira, é «de mais alto nível».

Podemos criar uma fun usando as seguintes três formas para a sua criação:

```
F = fun (Arg1, Arg2, ... argN) -> ... end
```

Se a função já está escrita no mesmo módulo podemos usar:

```
F = fun NomedaFuncao/Aridade
```

Se a função já está escrita noutro módulo:

```
F= {modulo,funcao}
```

Eis um exemplo da definição de funs, usando os três diferentes estilos:

```
-module(fun_test).
-export([t1/0, t2/0, t3/0, t4/0, double/1]).
-import(lists, [map/2]).
double(X) -> X * 2.
t1() -> map(fun(X) -> 2 * X end, [1,2,3,4,5]).
t2() -> map(fun double/1, [1,2,3,4,5]).
t3() -> map({?MODULE, double}, [1,2,3,4,5]).
```

Todas as variáveis que ocorrem na cabeça de uma fun são variáveis «frescas». Variáveis que estejam definidas antes, e que ocorram em chamadas a funções ou «guards» possuem o valor que tinham antes da fun. Não se pode exportar variáveis de uma fun.

De seguida vamos ver mais alguns exemplos tirados do módulo `lists` que interessam para vermos como se trabalha com funs, e também porque as funções definidas são muito úteis.

any – vê se existe algum elemento da lista para o qual um predicado seja verdadeiro:

```
any(Pred, [H|T]) ->
      case Pred(H) of
      true -> true;
      false -> any(Pred, T)
      end;
any(Pred, []) ->false.
```

all – vê se um dado predicado é verdadeiro para todos os elementos de uma lista

```
all(Pred, [H|T]) ->
  case Pred(H) of
          true -> all(Pred, T);
          false -> false
  end;
all(Pred, []) ->true.
```

foldl – aceita uma função de dois argumentos, uma lista e um acumulador. A função deve devolver um acumulador que é usado de cada vez que a função é chamada.

```
foldl(F, Accu, [Hd|Tail]) ->
              foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) -> Accu.
```

Exemplo de utilização:

```
9 > L = ["I","like","Erlang"].
["I","like","Erlang"]
10>lists:foldl(fun(X,Sum)->length(X)+Sum end,0,L).
11
```

«*Foldl*» que dizer dobrar uma lista. Há uma função que aproveita o seu resultado anterior através do segundo argumento do *foldl*, e é aplicada a cada um dos elementos da lista até que

esta chegue ao fim. No fim da lista o valor do segundo argumento do *foldl* é o resultado da função.

mapfoldl – faz map e foldl ao mesmo tempo

```
  mapfoldl(F, Accu0, [Hd|Tail]) ->
            {R,Accu1} = F(Hd, Accu0),
            {Rs,Accu2} = mapfoldl(F, Accu1, Tail),
            {[R|Rs], Accu2};
mapfoldl(F, Accu, []) -> {[], Accu}.
```

Exemplo de uso:

```
  11 > Upcase = fun(X) when $a = < X,
                X = < $z - > X + $A - $a;
                (X) - > X
                end.
  #Fun < erl eval >
  12 > Upcase_word =fun(X)->lists:map(Upcase, X)
            end.
  #Fun < erl eval >
  13 > Upcase_word("Erlang").
  "ERLANG"
  14 > lists:map(Upcase_word, L).
  ["I","LIKE","ERLANG"]
  14 > lists:mapfoldl(fun(Word, Sum) - >
  14 > {Upcase word(Word), Sum + length(Word)}
  14 > end, 0, L).
  {["I","LIKE","ERLANG"],11}
```

filter – devolve os elementos de uma lista para os quais um dado predicado é verdadeiro.

```
  filter(F, [H|T]) ->
    case F(H) of
    true -> [H|filter(F, T)];
    false -> filter(F, T)
    end;
  filter(F, []) -> [].
```

takewhile – devolve o grupo de elementos contíguos colocados na cabeça de uma lista para os quais um dado predicado é verdadeiro.

```
takewhile(Pred, [H|T]) ->
  case Pred(H) of
  true -> [H|takewhile(Pred, T)];
  false -> []
end;
takewhile(Pred, []) ->[].
```

dropwhile – retira de uma lista o grupo de elementos contíguos colocados na sua cabeça para os quais um dado predicado é verdadeiro.

```
dropwhile(Pred, [H|T]) ->
  case Pred(H) of
  true -> dropwhile(Pred, T);
  false -> [H|T]
end;
dropwhile(Pred, []) ->[].
```

first – devolve o primeiro elemento da lista para o qual um predicado é verdadeiro.

```
first(Pred, [H|T]) ->
          case Pred(H) of
          true ->{true, H};
          false ->first(Pred, T)
          end;
first(Pred, []) ->false.
```

splitlist – parte uma lista em duas, a primeira com o o grupo de elementos contíguos colocados na sua cabeça para os quais um dado predicado é verdadeiro, e a segunda com os elementos restantes.

```
splitlist(Pred, L) ->splitlist(Pred, L, []).
splitlist(Pred, [H|T], L) ->
          case Pred(H) of
          true -> splitlist(Pred, T, [H|L]);
          false -> {reverse(L), [H|T]}
end;
splitlist(Pred, [], L) ->{reverse(L), []}.
```