Erlang Solutions Ltd.

Advanced Constructs

© 1999-2011 Erlang Solutions Ltd.

Advanced Constructs

- List Comprehensions
- Binaries
- References
- Operators
- Macros
- Include Files

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions

```
[Expression || Generator1 , ..., GeneratorN,
               Filter1, ..., FilterN]
```

- A feature common in functional programming languages
- Analogous to set comprehensions in the Zermelo-Frankel set theory
- A syntactical and semantical notation to generate lists

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions

```
[X || X <-[1,2,3,4], X < 3]
```

- The above example should be read as the list of X where X comes from the list [1,2,3,4] and X is less than 3
- **Pattern <- List** is the generator
- Filter is either a boolean expression or a function which returns true or false

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions: examples

```
1> [Int || Int <- [zero, 1, two, three, 4, 5],
1>         is_integer(Int)].
[1,4,5]
2> [{X,Y} || X <- [1,2,3], Y <- [a,b,c]].
[{1,a},{1,b},{1,c},{2,a},{2,b},{2,c},{3,a},{3,b},{3,c}]
3> [X || X <- [1,2,3,4], Y <- [3,4,5,6], X == Y].
[3,4]
4> [X+1 || X <- [1,2,3,4], X rem 2 == 0].
[3,5]
```

- Filtering, cartesian products, intersections, and selective mapping using list comprehensions

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions: examples

```
map(Fun, List) ->
    [Fun(X) || X <- List].
filter(Predicate, List) ->
    [X || X <- List, Predicate(X)].
append(ListOfLists) ->
    [X || List <- ListOfLists, X <- List].
```

- Rewriting lists library functions using list comprehensions

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions: examples

```
perm([])  ->
    [[]];
perm(List) ->
    [[H|T] || H <- List, T <- perm(List -- [H])].
```

- perm([c,a,t]) -> [[c,a,t],[c,t,a],[a,c,t],[a,t,c],[t,c,a],[t,a,c]]
- We take **H** from **List** in all possible ways, and append all permutations of **List** with **H** removed to it

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions: variables

- All variables in the generator pattern are considered fresh
- Bound variables in the generator and before the LC expression which are used in the filter retain their value
- No variable can be exported from a LC expression
- The compiler gives a warning when you shadow variables

© 1999-2011 Erlang Solutions Ltd.

List Comprehensions: variables

```
1> X = 1, Y = 2.
2
2> [{X, Y} || X <- lists:seq(1,3)].
[{1,2},{2,2},{3,2}]
3> List = [{1,one}, {2,two}, {3,three}].
[{1,one},{2,two},{3,three}]
4> [Z || {X1, Z} <- List, X1 == X].
[one]
```

© 1999-2011 Erlang Solutions Ltd.

Binaries

- A binary is a reference to a chunk of untyped memory
- Originally used by the ERTS for code loading over the network
- Effective when moving large amounts of data among processes
- BIFs
  - **binary_to_term/1, term_to_binary/1, binary_to_list/1, split_binary/2, concat_binary/1**
  - **is_binary/1** as a guard

© 1999-2011 Erlang Solutions Ltd.

Binaries

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

Bin = <<1, 2, 3>>
binary_to_list(Bin) == [1,2,3]
```

- A **Bin** is a low level sequence of bytes
- They can be used to construct and pattern match Binaries
- Each element specifies a segment of the binary
- A segment is a set of bits, not necessarily a byte

© 1999-2011 Erlang Solutions Ltd.

Binaries

```
Bin = <<E1:[Size/Type], E2:[Size/Type], ..., En>>

Bin = <<1,2,3:16>> == list_to_binary([1,2,0,3])
```

- **Size** and **Type**s can be specified or omitted
- **Size** is in bits. Total size must be a multiple of 8 (a byte)
- **Type** is a list of type specifiers separated by hyphens
- Valid types are **integer**, **float** or **binary**

© 1999-2011 Erlang Solutions Ltd.

Binaries

- Valid signs are **signed** and **unsigned** (Default)
  - If the first bit is 0, the integer is positive
  - If it is 1, it is negative
- Valid endian values are **big** (Default) and **little**
  - Little endian, the first byte is the most significant
  - Big endian, the first byte is the least significant
- Default size of
  - **integers** is 8
  - **floats** is 64
  - **binaries** is the size of the binary

© 1999-2011 Erlang Solutions Ltd.

Binaries

```
<< 5:4/little-signed-integer-unit:8>> == <<5,0,0,0>>
  << 5:4/big-signed-integer-unit:8>> == <<0,0,0,5>>
```

- **unit:Val** is the default size of the type times the unit
  - Val is an integer between 1 - 255
  - Default unit is type dependent. It is 1 for float / integer, 8 for binary
- The element created has a size of (4*8) 32 bits

© 1999-2011 Erlang Solutions Ltd.

Binaries

```
A = 1, Bin = <<A, 17, 42:16>>
<<D:16, E, F/binary>> = Bin

D = 273, E = 0, binary_to_list(F) = [42]
```

- A Bin can be used to pattern match binaries
- Length and types can be specified or omitted
- Default types are unsigned integers
- Binary segments types must have a size divisible by 8

© 1999-2011 Erlang Solutions Ltd.

Binaries

- `B=<<1>>` will not compile. It is interpreted as `B =< <1>>`. Write `B = <<1>>`
- `<<X+1:8>>` will not compile. Write `<<(X+1):8>>`
- `<<"hello">>` is the same as writing `<<$h,$e,$l,$l,$o>>`
- `foo(N, <<X:N, T/binary>>) -> ...` will not compile. The two instances of N are unrelated
- `<<X:7/binary-unit:1, Y/binary>>` will never match. The size and unit, when multiplied, must be divisible by 8.

© 1999-2011 Erlang Solutions Ltd.

Binaries

- `<<X:7/binary, Y:1/binary>> = Z` will match because binaries have a default unit of 8. It is equivalent to `<<X:7/binary-unit:8, Y:1/binary-unit:8>> = Z`. 7*8 = 56 bits binary.
- `<<X:7/integer, Y/binary>> = Z` will fail because integers have a default unit of 1. When multiplied by 7, it isn't divisible by 8.
- `<<X:7/bitstring-unit:1, Y/bitstring>> = Z` will work, because the **bitstring** type doesn't require 8-bit alignment.

© 1999-2011 Erlang Solutions Ltd.

Binaries: examples

```
1> <<5:4, 5:4>>.
<<"U">>
2> <<Int1:2, Int2:6>> = <<128>>.
<<128>>
3> {Int1, Int2}.
{2,0}
4> <<5:4/little-signed-integer-unit:4>>.
<<5,0>>
5> <<5:4/big-signed-integer-unit:4>>.
<<0,5>>
6> <<5:2,5:8>>.
<<65,1:2>>
```

© 1999-2011 Erlang Solutions Ltd.

Binaries: examples

```
1> A = 1.
1
2> Bin = <<A, 17, 42:16>>.
<<1,17,0,42>>
3> <<D:16, E, F/binary>> = Bin.
<<1,17,0,42>>
4> [D,E,F].
[273,0,<<"*">>]
5> <<X:7/bitstring,Y:1/bitstring>> = <<42:8>>.
<<"*">>
6> {X, Y}.
{<<21:7>>,<<0:1>>}
```


© 1999-2011 Erlang Solutions Ltd.

Binaries

- Concatenating binaries will copy the chunks and create a completely new binary
- Concatenate your binary once you have all the chunks
- There is no need to append binary chunks sent to a port
- There is no need to flatten lists of binary chunks

© 1999-2011 Erlang Solutions Ltd.

Binaries: binary comprehensions

```
[ ... || X <- List, Test, ... ]
<< <<...>> || <<X>> <= Bin, Test, ... >>
```

- Structure similar to list comprehensions
- **Bin** must be a binary rather than a list
- A list generator can be used in a binary comprehension and a binary generator can be used in a list comprehension

© 1999-2011 Erlang Solutions Ltd.

Binaries: binary comprehensions

```
1> << <<X>> || <<X>> <= <<0,1,2,3>> >>.
<<0,1,2,3>>
2> << <<(X+1)/integer>> || <<X>> <= <<3,7,5,4,7>> >>.
<<4,8,6,5,8>>
3> << <<(bnot X)>> || <<X>> <= <<0,1,2,3>> >>.
<<"ÿþýü">>
4> << <<X>> || X <- [0,1,2,3] >>.
<<0,1,2,3>>
5> [ X || <<X>> <= <<0,1,2,3>>].
[0,1,2,3]
6> << <<X>> || <<X>> <= <<1,2,3,4>>, X rem 2 =:= 0 >>.
<<2,4>>
```

© 1999-2011 Erlang Solutions Ltd.

References

- References can be created using the BIF **make_ref/0**
- They can be compared for equality
- Erlang references are unique[ish]
  - No two references created by different calls will match for the life of a node*
- References can be used to guarantee unique replies to messages, and not just any message complying to the specified protocol

© 1999-2011 Erlang Solutions Ltd.

References

```
call(Message) ->
  Ref = make_ref(),
  frequency ! {request,
          {Ref,self()},Message},
  receive
     {reply, Ref, Reply} ->
        Reply
  end.

reply({Ref, Pid}, Message) ->
   Pid ! {reply, Ref, Message}.
```

- Recall the robust server call? How do we differentiate between messages following the same protocol?

© 1999-2011 Erlang Solutions Ltd.

Operators: logical

```
not ((1 < 3) or
 lists:member(Key,List))

and         andalso
or          orelse
xor         not
```

- Booleans are represented by the atoms **true** and **false**.

© 1999-2011 Erlang Solutions Ltd.

Operators: bitwise

- Bit level operations can be applied on integers
- The operands have the same precedence as + or -
- Logical comparisons on a bit level include:
  - **band**, **bor**, **bxor** and **bnot**
- Shifting bits left or right can also be done:
  - **bsl**, **bsr**

© 1999-2011 Erlang Solutions Ltd.

Operators: bitwise

```
1> 9 band 17.
1
2> 9 bor 17.
25
3> 9 bxor 17.
24
4> bnot 9.
-10
5> bnot (bnot 9).
9
6> 6 bsr 1.
3
7> 6 bsl 1.
12
```

© 1999-2011 Erlang Solutions Ltd.

Macros

```
-define(Name, Replacement).
-define(Name(Var1, ..., VarN),
        Replacement).


-define(TIMEOUT, 1000).
...

  receive
       after ?TIMEOUT -> ok
  end
```

- Macros are defined using the define construct
- They are used whenever the **?Name** syntax is encountered

© 1999-2011 Erlang Solutions Ltd.

Macros

```
-define(DOUBLE(Y), Y*2).


foo(X) -> ?DOUBLE(X).
%% Becomes
foo(X)  -> X*2.
```

- All occurrences of the **?DOUBLE** macro will be expanded upon compilation by the pre-processor to **X*2**

```
1> macro:foo(10).
20
```

© 1999-2011 Erlang Solutions Ltd.

Macros

```
-define(CALLSTRING(Call), print(??Call)).

?CALLSTRING(foo(X))
%% Becomes
print("foo(X)")
```

- All occurrences of the **??Name** will expand to a string containing the tokens of the macro

© 1999-2011 Erlang Solutions Ltd.

Macros: predefined macros

?MODULE
Expands to the name of the module it is used in

?MODULE_STRING
Expands to the module string

?FILE
Expands to the file name it is placed in

?LINE
Expands to the line of code it is defined in

?MACHINE
Expands to the current machine name

© 1999-2011 Erlang Solutions Ltd.

Macros: example of flow control

```
-ifdef(debug).
  -define(SYS_DEBUG(Str, Args), io:format(Str, Args)).
-else.
  -define(SYS_DEBUG(Str, Args), ok).
-endif.
```

- In the code use it as follows:
  - ?SYS_DEBUG("~p:call(~p) called~n",[?MODULE, Request])
- To turn on the debug print outs, compile using c/2
  - c(Module, [{debug, true}]) or compile:file/2
- To view a file with expanded macros use the 'P' directive
  - c(Module, ['P']) or compile:file/2

© 1999-2011 Erlang Solutions Ltd.

Macros: flow of control

-undef(Macro)
Macro behaves as if it had never been defined

-ifdef(Macro)
if Macro is defined, execute the statements which follow

-ifndef(Macro
If Macro is not defined, execute the statements which follow

-else
If not **ifdef** or not **ifndef**, then execute the statements

-endif
terminates the **ifdef** and **ifndef** constructs

© 1999-2011 Erlang Solutions Ltd.

Macros: flow control

```
-ifdef(Macro).
  -define(Name1, Replacement1).
-else.
  -define(Name2, Replacement2).
-endif.
```

- Macro directives have to be properly nested
- They may not be placed in functions

© 1999-2011 Erlang Solutions Ltd.

Include Files

- All shared record and macro definitions should be placed in include files.
- The suffix **.hrl** is recommended but not enforced
- Include files are added in a file using the **-include("File.hrl").** directive "quotes" included
- The compiler will look for the include file in the compiler include path list
- By default, the include path lists includes the current working directory

© 1999-2011 Erlang Solutions Ltd.

Advanced Constructs

- List Comprehensions
- Binaries
- References
- Operators
- Macros
- Include Files

© 1999-2011 Erlang Solutions Ltd.