Erlang Solutions Ltd.

Records and Funs

Records and Funs

- Records
- Records and the Shell
- Funs
- Higher Order Functions

Records

- Records are used to store a fixed number of items
  - Similar to a C struct or a Pascal record
- These items are accessed by name
  - Unlike tuples where they are accessed by position
- Items can easily be added or removed without affecting the code not using these items
  - Unlike tuples, where updates are needed everywhere

Records: defining

```
-record(Name, {Field1, [= DefaultValue1],
               ...,
               FieldN, [= DefaultValueN]}).

-record(person, {name, age = 0, phone}).
```

- Fields may be assigned a value when declared
- The default value for a field is the atom **undefined**
- Record definitions should be placed in a header file

Records: defining

- All **shared** record definitions should be placed in include files. Local ones stay in their modules.
- The suffix **.hrl** is recommended but not enforced
- Include files are added to a module 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 list includes the current working directory

Records: creating instances

```
Var = #Name{Field1 = Expression1,
            ...,
            FieldN = ExpressionN}

Person = #person{name = "Joe Armstrong",
                 phone = [1,2,3,4]}
```

- If any of the fields are omitted , the default values in the record definition are assigned to them (including **undefined**)

Records: field selectors

```
FieldVar = RecordVar#Name.FieldName

P = #person{name = "Mike Williams"}

Name = P#person.name           %% Name =:= "Mike Williams"
Age  = P#person.age            %% Age =:= 0
```

- Field and record names may not be variables

Records: Updating

```
NewVar = OldVar#Name{Field1 = Expression1,
                     ...,
                     FieldN = ExpressionN}

P = #person{name = "Mike Williams"}        % age = 0
NewP = P#person{age = 35}                  % age = 35
NewP2 = P#person{name = "Joe"}             % age = 0
```

- Only fields to be changed have to be referred
- Others will return their old values
- Remember that Erlang variables are single assignment!

Records: pattern matching

```
P = #person{name = "Joe", age = 35, phone = [1,2,3]}

#person{name = Name, age = 35, phone = Phone} = P

foo(#person{name = "Joe", age = Age}) -> ...
```

- Records may be used in pattern matching to extract variables or pick the flow of computation

Records: guards

```
foobar(P) when is_record(P, person),
              P#person.name =:= "Joe" -> ...

is the same as, but less efficient than:

foobar(P = #person{name = "Joe"}) -> ...
```

- Record guards may be used to pick the flow of execution in different clauses
- When using guards to inspect a field of a record, use the record guard as well if **P** will not always be a record of type **person**.

Records: nesting

```
-record(name, {first, surname}).

P = #person{name = #name{first = "Robert",
                         surname = "Virding"}},

First = (P#person.name)#name.first.
```

- Record fields may contain other nested records
- Fields in nested records are accessed with one operation

Records: internal representation

```
#person{} =:= {person, undefined, 0, undefined}
```

Warning! Never use the tuple representation of records!

- Records are represented as tuples by the run time system
- The preprocessor translates the creating, updating and selecting operations on records to operations on tuples
- N fields in the record will result in a tuple with N+1 elements
- The first element is the name of the record

Records: information

- record_info(fields, RecType)
  - returns a list of field names
- record_info(size, RecType)
  - returns the size of the tuple (Fields + 1)
- #RecType.Name returns the position of Name in the tuple
- RecType and Name must be atoms, and they may not be variables bound to atoms
- Record information constructs are handled by the precompiler

Records and the Shell

```
1> rr(person).
[person].
2> rl().
-record(person, {name,
                 age = 0,
                 phone}
ok
3> #person{name='Henry',
3>         phone=[0,1,2]}.
#person{name='Henry',
        age=0,
        phone=[0,1,2]}
```

- All record definitions in a module can be loaded using the function shell:rr/1
- The records known to the shell can be listed with shell:rl/0

Records and the Shell

```
4> rd(request, {id, action,
4>   data, stamp=now()}).
request
5> #request{}.
#request{id = undefined,
         action = undefined,
         data = undefined,
         stamp = {1151, 913849,
                  108529}}
6> rf(request).
ok
```

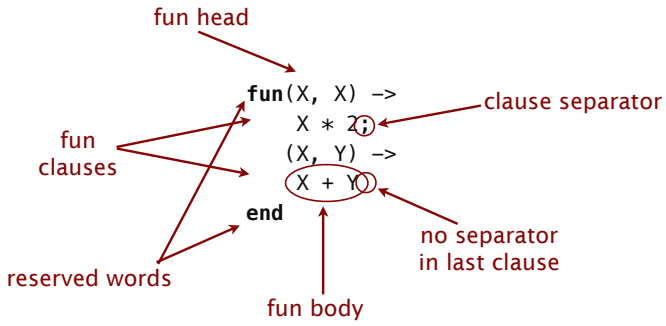
- Records can be defined using shell:rd/2
- Useful for testing and debugging
- A record can also be forgotten like variables by using rf/0 and rf/1

Funs

```
1> Add = fun(X, Y) -> X+Y end.
#Fun<erl_eval>
2> Add(2,3).
5
```

- Funs are data types encapsulating functional objects
- They can be passed as arguments
- They can be the return value of function calls

Functions: syntax



Funs: syntax

```
fun(Var1, ..., VarN) ->
  <Expr1>,
  <Expr2>,
  ...
  <ExprN>;
   (Var1, ..., VarN) ->
  <Expr1>,
  <Expr2>,
   ...
  <ExprN>
end
```

- The syntax is similar to that of functions, only that it starts with the keyword **fun** and ends with the keyword **end**.

Funs: syntax

F = fun Function/Arity
Will bind the local function in the current module to **F**

F = fun Module:Function/Arity
Will bind the function exported in **Module** to **F**.

F = {Module, Function}
Will bind the function exported in **Module** to **F**. This form is deprecated, should **not** be used and is documented in case you work with legacy code (pre R11).

Funs: higher order functions

- Functions taking funs as arguments are called higher order functions
- They encourage the encapsulation of common design patterns, facilitate the re-usage of these functions
- Improves the clarity of the program
- Hides recursive calls
- The process of abstracting out common patterns in programs is called **procedural abstraction**.

Funs: procedural abstraction

Before

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

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

After

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

double(L) ->
   map(fun(X)-> X*2 end, L).

bump(L) ->
   map(fun(X)-> X+1 end, L).
```

Funs: higher order functions

lists:all(Predicate, List) -> true | false
Returns **true** if the **Predicate** fun returns **true** for all elements in **List**

lists:filter(Predicate, List) -> NewList
Returns a list with elements for which **Predicate** is true

lists:foreach(Fun, List) -> ok
Applies **Fun** on every element in the list. Used for side effects

lists:map(Fun, List) -> NewList
Returns a list with the return value of **Fun** applied to all elements in **List**

Funs: examples

```
1> Bump = fun(X) -> X+1 end.
#Fun<erl_eval.19.120858100>
2> lists:map(Bump, [1,2,3,4,5]).
[2,3,4,5,6]
3> Positive = fun(X) -> if X < 0 -> false;
                           X >= 0 -> true
                        end end.
#Fun<erl_eval.19.120858100>
4> lists:filter(Positive, [-2,-1,0,1,2]).
[0,1,2]
5> lists:all(Positive, [0,1,2,3,4]).
true
6> lists:all(Positive, [-1,0,1]).
false
```


Funs: scope of variables

- All variables in the head of the Fun are considered fresh, and bound when the fun is first called
- Variables bound before the Fun can be used in the Fun and in guard tests
- No variables may be exported from the Fun
- Variables in the function head shadow already bound variables in the function the Fun is defined in

Funs: scope of variables

```
foo() ->
  X = 2,
  Bump = fun(X) -> X + 1 end,
  Bump(10).
```

X is shadowed in the fun

```
1> funs:foo().
11
```

Funs: scope of variables

```
bar() ->
  X = 10,
  Bump = fun(Y) -> X + Y end,
  Bump(10).
```

X is not shadowed in the fun

```
1> funs:bar().
20
```

Funs: scope of variables

```
1> GreaterThan = fun(X) ->
1>   fun(Y) -> Y > X end
1> end.
#Fun<erl_eval.6.13229925>
2> GreaterThan(4).
#Fun<erl_eval.6.13229925>
3> (GreaterThan(4))(3).
false
4> (GreaterThan(4))(5).
true
5> lists:filter(GreaterThan(5),
5>         [1,6,8,3,5,0,4,11]).
[6,8,11]
```

- It is possible for a Fun to return another Fun.
- This can be used to introduce a new variable in the Fun's scope to 'wrap' the arguments it would usually need.

Records and Funs

- Records
- Records and the shell
- Funs
- Higher Order Functions