CS11 – Erlang

Winter 2012–2013

Lecture 3

Large Tuples

- Create a module to manage user profiles
- Need a way to actually store user data…
- Use tuples:

```
User = {user, "bob", "Bob", "Smith",
    {address, "123 Smith Way",
        "Springfield", "CA", "98765"},
    ["555-1234", "555-5678"],
    ["bob@host1.com", "bob@host2.com]"}.
```

- Issues:
  - Large and complicated to extract data from, etc.
  - What do various components actually represent?!

Example: Phone Numbers

- Our tuple:

```
User = {user, "bob", "Bob", "Smith",
        {address, "123 Smith Way",
            "Springfield", "CA", "98765"},
        ["555-1234", "555-5678"],
        ["bob@host1.com", "bob@host2.com]"}.
```

- To extract the list of phone numbers:

```
{user, _, _, _, _, PhoneNums, _} = User.
```

  - `PhoneNums` = `["555-1234", "555-5678"]`

- Or, use **`element()`** BIF:

```
PhoneNums = element(User, 6).
```

Records

- Erlang provides record abstraction to simplify use of large tuples
  - Records are implemented as tuples, but provide additional semantics and capabilities over tuples
- Define a user record:

```
-record(user,
        {username, first_name, last_name,
         address, phone_nums, email_addrs}).
```

  - Name of record-type is **`user`**
  - **`username`**, **`first_name`**, etc. are record fields
- Note: **`-record`** is a module attribute
  - Cannot be specified directly in the Erlang shell

Records (2)

- Can also define an address record:
  `-record(address, {street,city,state,zipcode}).`
  - An **address** record-type, with four fields
- To create a record:

```
User = #user{username="bob",
    first_name="Bob", last_name="Smith",
    address=#address{street="123 Smith Way",
                     city="Springfield", state="CA",
                     zipcode="98765"},
    phone_nums=["555-1234", "555-5678"],
    email_addrs=["bob@host1.com", "bob@host2.com"]}.
```

  - About twice as long as before…
  - But, at least everything is clearly named.

Unspecified Fields

- Don't have to specify all fields when creating a record
- Example: Only specify Bob's name, username

```
User = #user{username="bob",
        first_name="Bob", last_name="Smith"}.
```

- Result:

```
#user{username="bob", first_name="Bob",
  last_name="Smith", address=undefined,
  phone_nums=undefined, email_addrs=undefined}
```

- Unspecified fields are set to **`undefined`** atom

Unspecified Fields (2)

- Can specify named fields in any order

```
User = #user{last_name="Smith",
        first_name="Bob", username="bob"}.
```

- Erlang requires that name is included when specifying field values
  - e.g. can't rely on field position to indicate field

Empty Records

- Example: make an empty user record

```
NewUser = #user{}.
```

- Result:

```
#user{username = undefined,
      first_name = undefined,
      last_name = undefined,
      address = undefined,
      phone_nums = undefined,
      email_addrs = undefined}
```

  - All fields are set to **undefined**

Specifying Default Values

- Want **`phone_nums`** and **`email_addrs`** to be empty lists when unspecified
  - …rather than **`undefined`** atom!
- Can specify default values in record definition

```
-record(user,
        {username, first_name, last_name,
         address, phone_nums=[], email_addrs=[]}).
```

- If a field doesn't specify a default value, then **`undefined`** is used as the default

Specifying Default Values (2)

- Now when we create Bob's user record:

```
User = #user{username="bob",
             first_name="Bob", last_name="Smith"}.
```

- Result:

```
#user{username = "bob",
      first_name = "Bob",
      last_name = "Smith",
      address = undefined,
      phone_nums = [],
      email_addrs = []}
```

Our `users` Module

- Use our records to start building a **`users`** module

```
-module(users).
-export([make_user/3]).

-record(user, {username, first_name, last_name,
    address, phone_nums=[], email_addrs=[]}).
-record(address, {street, city, state, zipcode}).

make_user(FirstName, LastName, Username) ->
    #user{first_name=FirstName, last_name=LastName,
          username=Username}.
```

- Save in **`users.erl`**, compile it, etc.

Using Our `users` Module

- Go ahead and use this from the Erlang shell:

```
1> Bob = users:make_user("Bob","Smith","bob").
{user,"bob","Bob","Smith",undefined,[],[]}
2>
```

- We get back a tuple, not a record?!
  - Reason: Erlang shell doesn't automatically load the record definitions from the module
- Use shell command **`rr(Name)`**
  - "Read records"
  - Takes a module name (atom) or a file name (string)

Using Our `users` Module (2)

- Second try:

```
1> rr(users).
[address,user]
2> Bob = users:make_user("Bob","Smith","bob").
#user{username="bob", first_name="Bob",
      last_name="Smith", address=undefined,
      phone_nums=[], email_addrs=[]}
3>
```

- Now the shell knows about our record types
  - Note: **`rr(`*`Name`*`)`** returns list of record types that were found

Accessing Record Fields

- To access a specific field of a record:

```
1> rr(users).
[address,user]
2> U = #user{last_name="Smith",
     first_name="Bob", username="bob"}.
...
3> U#user.username.
"bob"
4> U#user.phone_nums.
[]
5>
```

- Syntax: ***VarName#RecName.Field***

Copying Records

- Erlang provides special syntax for updating records

```
U1 = #user{last_name="Smith",
     first_name="Bob", username="bob"}.
...
U2 = U1#user{first_name="Robert",
     phone_nums=["555-1234"]}.
```

  - Creates a new record by duplicating the original record (`U1`), and replacing specified fields
  - Result is a *copy* of `U1`; `U1` and `U2` are independent records

Records and Patterns

- Can also use records in pattern matching
- Syntax is same as for creating records

```
#Name(Field1=Pattern1, Field2=Pattern2, ...)
```

- Difference from record creation:
  - Field-values may also contain unbound variables
- A simple example:

```
get_username(#user{username=U}) -> U.
```

Erlang Header Files

- A more general issue:
  - How do other Erlang modules use our record types?
- Solution:
  - Put record declarations into a header file
  - Other modules can include the header file as well
- Erlang header files use **`.hrl`** extension
- Create **`users.hrl`**:

```
% Record types for the users module

-record(user, {username, first_name, last_name,
    address, phone_nums=[], email_addrs=[]}).

-record(address, {street, city, state, zipcode}).
```

Including Header Files

- Update **`users.erl`** to include **`users.hrl`**

```
-module(users).
-export([make_user/3]).

-include("users.hrl").

make_user(FirstName, LastName, Username) ->
    #user{first_name=FirstName,
          last_name=LastName,
          username=Username}.
```

- **`-include`** attr takes a string, not an atom!
  - Specifies relative or absolute path to header file

Using Header Files

- Now, other modules can use our **`user`** and **`address`** record types
  - Just add **`-include("users.hrl")`** to the module's **`.erl`** file
- Shell still needs to use "read records" function
  - Both **`rr("users.hrl")`** and **`rr(users)`** will work

Erlang Macros

- Used to define constants or macro-functions
  - Very similar to C/C++ macro capability
- Macros can only be declared/used in modules
  - Macros are processed at compile-time
  - Cannot use in the Erlang shell
- Constants:

```
-define(CONST, Replacement).
```

- Functions:

```
-define(FUNC(Var1, Var2, ...), Replacement).
```

- Usual convention is **ALL_CAPS** for macros

Erlang Macros (2)

- To use:

  `?CONST`

  `?FUNC(Arg1, Arg2, ...)`
  - Question-mark before macro name
- Several predefined macros for you to use:
  - **`?MODULE`** – name of current module, as an atom
  - **`?MODULE_STRING`** – name of current module, as a string
  - **`?FILE`** – filename of the current module
  - **`?LINE`** – current line-number

Example Macro Usage

- Spawning a process from within a module

```
-module(rss_queue).
...

% Start an rss_queue server and return the PID
start() -> spawn(?MODULE, server, []).
```

  - Uses the module/function/arguments (MFA) version of spawn
  - Avoids having to hard-code module name into call

Example Macro Usage (2)

- Defining your own constants:

```
% Timeout period for all receive operations
-define(RECV_TIMEOUT, 1000).
...
% Start a task on server, then recv the result
do_task(ServerPid, Spec) ->
    ServerPid ! {task, Spec},
    receive
        Result -> ...
    after ?RECV_TIMEOUT ->
        {error, timeout}
    end.
```

Example Macro Usage (3)

- Defining helper functions:

```
-define(TRACE(X), io:format("{~p,~p}: ~p~n",
          [?MODULE,?LINE,X])).
...
server() ->
    ?TRACE("Starting server."),
    ...
```

- This is a very simple trace-output facility
- Would want many additional features!
  - Ability to turn off debug messages at compile-time, different logging levels, richer message formatting

Strings in Erlang

- Erlang strings are simply lists of numbers

```
1> [104, 101, 108, 108, 111].
"hello"
```

  - `"hello"` is shorthand for the list of integers for each character
- If list contains only numbers, and all values correspond to printable characters, then list is displayed as a string

```
2> [1, 104, 101, 108, 108, 111].
[1, 104, 101, 108, 108, 111]
```

  - `1` doesn't correspond to a printable character

Characters and Strings

- Can also specify character literals using **$**

```
1> $h.
104
2> [$h, $e, $l, $l, $o].
"hello"
```

- **$** followed by a character evaluates to the character's value

```
3> $$.
36
4> $..
46
```

String Concatenation

- Erlang automatically concatenates adjacent string literals

```
1> "Hello world!".
"Hello world!"
2> "Hello " "world!".
"Hello world!"
```

  - Useful if you need to write long strings that span multiple lines
- Can also use other list processing operations

```
3> "Hello " ++ "world!".
"Hello world!"
```

String-Related BIFs

- Can convert between atoms and strings
- **`atom_to_list`** converts an atom into a string

```
1> atom_to_list(hello).
"hello"
```

- **`list_to_atom`** converts a string into an atom

```
2> list_to_atom("hello").
hello
3> list_to_atom("Hello").
'Hello'
```

  - Remember: atoms that don't start with lowercase letter must be enclosed with single-quotes

Erlang XML Libraries

- Erlang includes broad support for XML
  - Parsing and generating XML documents
  - XML schema validation
  - XML transforms using XSLT
  - XPath queries against XML documents
- Modules:
  - **`xmerl`**, **`xmerl_scan`**, **`xmerl_xsd`**, **`xmerl_xpath`**, etc.
- XML elements/attributes/etc. represented by various records
  - Most important records defined in **`xmerl.hrl`**

Including `xmerl.hrl`

- Location of **`xmerl.hrl`** depends on where Erlang was installed!
- Use **`-include_lib`** to include system headers

  `-include_lib("xmerl/include/xmerl.hrl").`
- Process:
  - First part **`"xmerl"`** used to find path to current version of **`xmerl`** library
    - Makes the call: **`code:lib_dir(xmerl)`**
      - String converted to an atom, then passed to function
    - e.g. "/usr/local/erlang-R15B02/lib/erlang/lib/xmerl-1.3.2"
  - Within this dir, the file "include/xmerl.hrl" is loaded

Example `xmerl` Record

- XML elements:

```
-record(xmlElement,{
    name,                       % atom()
    expanded_name = [],         % ...
    nsinfo = [],                % {Prefix, Local} | []
    namespace=#xmlNamespace{},
    parents = [],               % [{atom(),integer()}]
    pos,                        % integer()
    attributes = [],            % [#xmlAttribute()]
    content = [],
    ...
}).
```

  - Element name is stored as an atom
  - Attributes stored as a list of **`#xmlAttribute`** records
  - Element's contents stored as a list of records of various types

Processing XML Documents

- Can use pattern-matching expressions to look for specific elements in a document
- **`case`** statement will probably be very helpful!
  - Syntax is nearly identical to **`receive`** statement
- Syntax:

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...
    PatternN [when GuardSeqN] ->
        BodyN
end
```

`case` Statement

- Syntax:

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...
    PatternN [when GuardSeqN] ->
        BodyN
end
```

- First matching pattern is chosen
  - ***Body*** expression is evaluated
  - Result of **`case`** is result of the ***Body*** expression

`case` Statement (2)

- Like **`if`** statement, if no branch matches then a **`case_clause`** error is reported
- Can provide a "catch-all" clause at end

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...
    _Else ->
        ElseBody
end
```

- Final clause will match anything

This Week's Lab

- Start the RSS feed aggregator
- Perform some simple XML parsing and feed-item extraction
  - Some RSS 2.0-format XML files will be supplied for testing