COMCAST

Techniques for Metaprogramming in Erlang

Sean Cribbs
Comcast Cable (T+PD)
@seancribbs

Erlang User Conference
Stockholm
12 June 2015

About me

- **Senior Principal Engineer** at Comcast Cable
- Former **Technical Lead** for Riak at Basho
- Creator of `neotoma`, Erlang **packrat-parser toolkit**

COMCAST

Background

COMCAST

What is Metaprogramming?

- Code writing code
- Programs as data
  - Reflection / reflexivity
  - Homoiconicity

COMCAST

What is Metaprogramming?

- Code writing code
- Programs as data
  - Reflection / reflexivity
  - Homoiconicity

- Run-time

COMCAST

What is Metaprogramming?

- Code writing code
- Programs as data
  - Reflection / reflexivity
  - Homoiconicity

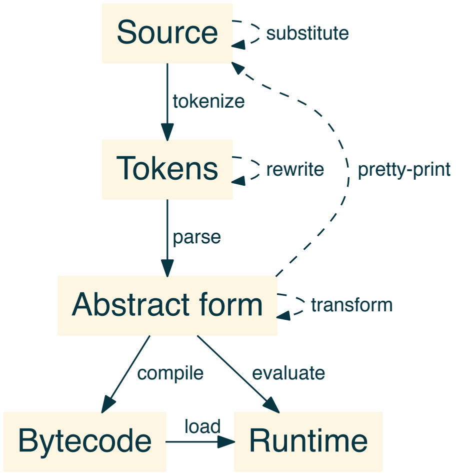
- Run-time
- **Compile-time**

COMCAST

Why Metaprogram?

- Reduce duplication
- Inject optimization
- Simplify APIs
- Improve tools
- Implement DSLs

Metaprogramming Erlang



COMCAST

Technique 1
Macros

COMCAST

Macros

- Generates code in preprocessor (`epp`)
- Operates over **Tokens** (mostly)

```erlang
% static term
-define(TIMEOUT, 5000).
```

COMCAST

Macros

- Generates code in preprocessor (`epp`)
- Operates over **Tokens** (mostly)

```
% static term
-define(TIMEOUT, 5000).
% parameterized
-define(THUNK(A), fun() -> (A) end).
-define(IF(B,T,F),
        begin
        (case (B) of true->(T); false->(F) end)
        end).
```

COMCAST

Macros

- Generates code in preprocessor (`epp`)
- Operates over **Tokens** (mostly)

```
% static term
-define(TIMEOUT, 5000).
% parameterized
-define(THUNK(A), fun() -> (A) end).
-define(IF(B,T,F),
        begin
        (case (B) of true->(T); false->(F) end)
        end).
%% escaped arguments
-define(Quote(A), io_lib:format("~s",[??A])).
```

Using Macros

```
gen_server:call(?MODULE, ping, ?TIMEOUT).
%% gen_server:call(mymodule, ping, 5000).
```

Using Macros

```
gen_server:call(?MODULE, ping, ?TIMEOUT).
%% gen_server:call(mymodule, ping, 5000).
```

```
Nope = ?THUNK(launch(missiles)).
%% Nope = fun() -> (launch(missiles)) end.
```

COMCAST

Using Macros

```
gen_server:call(?MODULE, ping, ?TIMEOUT).
%% gen_server:call(mymodule, ping, 5000).
```

```
Nope = ?THUNK(launch(missiles)).
%% Nope = fun() -> (launch(missiles)) end.
```

```
io:format("The value of ~s is ~p.", [?Quote(Foo), Foo]).
%% io:format("The value of ~s is ~p.",["Foo", Foo]).
```

Macros - eunit

```
-define(assert(BoolExpr),
        begin
        ((fun () ->










            end
          end)())
        end).
```

Macros - eunit

```
-define(assert(BoolExpr),
        begin
        ((fun () ->
            case (BoolExpr) of
                true -> ok;









            end
          end)())
        end).
```

Macros - eunit

```
-define(assert(BoolExpr),
        begin
        ((fun () ->
            case (BoolExpr) of
                true -> ok;
                __V -> erlang:error({assertion_failed,
                                     [{module, ?MODULE},
                                      {line, ?LINE},
                                      {expression, (??BoolExpr)},
                                      {expected, true},
                                      {value, case __V of false -> __V;
                                                  _ -> {not_a_boolean,__V}
                                              end}]})
            end
          end)())
        end).
```

Macros - eunit

```
fizzbuzz_test() ->
    ?assert(fizz =:= fizzbuzz(3)),
    ?assert(buzz =:= fizzbuzz(5)),
    ?assert(fizzbuzz =:= fizzbuzz(15)),
    ?assert(10 =:= fizzbuzz(10)).
```

Macros - eunit

```erlang
fizzbuzz_test() ->
    ?assert(fizz =:= fizzbuzz(3)),
    ?assert(buzz =:= fizzbuzz(5)),
    ?assert(fizzbuzz =:= fizzbuzz(15)),
    ?assert(10 =:= fizzbuzz(10)).
```

```
1> eunit:test(mymodule).
mymodule: fizzbuzz_test (module ’mymodule’)...*failed*
in function mymodule:’-fizzbuzz_test/0-fun-3-’/0 (mymodule.erl, line 18)
**error:{assertion_failed,[{module,mymodule},
                     {line,18},
                     {expression,"10 =:= fizzbuzz ( 10 )"},
                     {expected,true},
                     {value,false}]}
```

Macros - Summary

Pros:

- Easy and familiar
- Inline with program
- Syntax draws attention

COMCAST

Macros - Summary

Pros:

- Easy and familiar
- Inline with program
- Syntax draws attention

Cons:

- Limited expressivity
- Appearance

COMCAST

Macros - Summary

Pros:

- Easy and familiar
- Inline with program
- Syntax draws attention

Cons:

- Limited expressivity
- Appearance

Good for:

- Small API wrappers like in `eunit` or `eqc`
- Naming constants
- Compile-time feature-switching (OTP upgrades)
- Debugging statements

COMCAST

Technique 2
Parse Transforms

COMCAST

Parse Transforms

- Generates or transforms code after parsing
- Operates over **Abstract Form** (AST)

```
%% In your module:
-compile([{parse_transform, the_transform_module}]).

%% In the parse transform module:
parse_transform(Forms, _Options) ->
    %% 'Forms' is the AST. 'Options' are the compiler options.
    %% Traverse/modify 'Forms' and return it
    Forms.
```

COMCAST

Parse Transforms

- Generates or transforms code after parsing
- Operates over **Abstract Form** (AST)

```
%% In your module:
-compile([{parse_transform, the_transform_module}]).

%% In the parse transform module:
parse_transform(Forms, _Options) ->
    %% 'Forms' is the AST. 'Options' are the compiler options.
    %% Traverse/modify 'Forms' and return it
    Forms.
```

```
$ erlc -P mymodule.erl
$ cat mymodule.P
```

COMCAST

Parse Transforms - `lager`

github.com/basho/lager

- Rewrites calls to `lager:SYSLOG_SEVERITY_LEVEL`
- Injects producer-side filtering and call-site metadata

```
lager:warning("Resource threshold exceeded ~p:~p", [Used, Available]).
```

COMCAST

Parse Transforms - `lager`

github.com/basho/lager

- Rewrites calls to `lager:SYSLOG_SEVERITY_LEVEL`
- Injects producer-side filtering and call-site metadata

```
lager:warning("Resource threshold exceeded ~p:~p", [Used, Available]).
%% Becomes equivalent of:
case {whereis(lager_event), lager_config:get(loglevel, {0, []})} of
    {undefined, _} -> {error, lager_not_running};
    {Pid, {Level, Traces}} when (Level band 16) /= 0 orelse Traces /= [] ->
         lager:do_log(warning,[{module, mymodule}, {function, myfunc},
                              {line, 5}, {pid, pid_to_list(self())},
                              {node, node()} | lager:md()],
                     "Resource threshold exceeded ~p:~p",
                     [Used, Available], Level, Traces, Pid);
     _ -> ok
end.
```

COMCAST

Parse Transforms - `lager`

Understanding the AST

```
{ok, Bin} = file:read_file("lager_snippet.erl"),
{ok, Tokens, _} = erl_scan:string(unicode:characters_to_list(Bin)),
{ok, AST} = erl_parse:parse_exprs(Tokens),
AST.
```

Parse Transforms - `lager`

Understanding the AST

```erlang
{ok, Bin} = file:read_file("lager_snippet.erl"),
{ok, Tokens, _} = erl_scan:string(unicode:characters_to_list(Bin)),
{ok, AST} = erl_parse:parse_exprs(Tokens),
AST.
```

```erlang
[{'case',1,
     {tuple,1,
            [{call,1,{atom,1,whereis},[{atom,1,lager_event}]},
             {call,1,
                   {remote,1,{atom,1,lager_config},{atom,1,get}},
                   [{atom,1,loglevel},{tuple,1,[{integer,1,0},{nil,1}]}]}]},
     [{clause,2,
              [{tuple,2,[{atom,2,undefined},{var,2,'_'}]}],
              [],
              [{tuple,2,[{atom,2,error},{atom,2,lager_not_running}]}]},
      {clause,3,
              [{tuple,2
```

COMCAST

Parse Transforms - `lager`

`lager_transform` module

```
parse_transform(AST, Options) ->
    TruncSize = proplists:get_value(lager_truncation_size, Options,
                                    ?DEFAULT_TRUNCATION),
    Enable = proplists:get_value(lager_print_records_flag, Options, true),
    put(print_records_flag, Enable),
    put(truncation_size, TruncSize),
    erlang:put(records, []),
    %% .app file should either be in the outdir, or the same dir
    %% as the source file
    guess_application(proplists:get_value(outdir, Options), hd(AST)),
    walk_ast([], AST).
```

Parse Transforms - `lager`

Recursing through the AST

```
walk_ast(Acc, [{function, Line, Name, Arity, Clauses}|T]) ->
    put(function, Name),
    walk_ast([{function, Line, Name, Arity,
                walk_clauses([], Clauses)}|Acc], T);
```

COMCAST

Parse Transforms - `lager`

Recursing through the AST

```
walk_ast(Acc, [{function, Line, Name, Arity, Clauses}|T]) ->
    put(function, Name),
    walk_ast([{function, Line, Name, Arity,
                walk_clauses([], Clauses)}|Acc], T);
```

```
walk_clauses(Acc, []) ->
    lists:reverse(Acc);
walk_clauses(Acc, [{clause, Line, Arguments, Guards, Body}|T]) ->
    walk_clauses([{clause, Line, Arguments, Guards, walk_body([], Body)}|Acc], T).
```

Parse Transforms - `lager`

Recursing through the AST

```
walk_ast(Acc, [{function, Line, Name, Arity, Clauses}|T]) ->
    put(function, Name),
    walk_ast([{function, Line, Name, Arity,
                walk_clauses([], Clauses)}|Acc], T);
```

```
walk_clauses(Acc, []) ->
    lists:reverse(Acc);
walk_clauses(Acc, [{clause, Line, Arguments, Guards, Body}|T]) ->
    walk_clauses([{clause, Line, Arguments, Guards, walk_body([], Body)}|Acc], T).
```

```
walk_body(Acc, []) ->
    lists:reverse(Acc);
walk_body(Acc, [H|T]) ->
    walk_body([transform_statement(H)|Acc], T).
```

Parse Transforms - `lager`

Transforming matching calls

```erlang
transform_statement({call, Line, {remote, _Line1, {atom, _Line2, lager},
            {atom, _Line3, Severity}}, Arguments0} = Stmt) ->
    case lists:member(Severity, ?LEVELS) of
        false -> Stmt;   %% NB: Don't modify if it isn't a severity level!
        true -> %%...
    end;
```

Parse Transforms - `lager`

Checking the logging conditions

```
LevelVar = make_varname("__Level", Line),
TracesVar = make_varname("__Traces", Line),
PidVar = make_varname("__Pid", Line),
%% case {whereis(lager_event),
%%       lager_config:get(loglevel,{?LOG_NONE, []})} of
{'case', Line,
   {tuple, Line,
       [{call, Line, {atom, Line, whereis}, [{atom, Line, lager_event}]},
        {call, Line, {remote, Line, {atom, Line, lager_config}, {atom,
              Line, get}},
          [{atom, Line, loglevel}, {tuple, Line, [{integer, Line, 0},
                                                 {nil, Line}]}]}]},
   [
     %% case clauses ...
   ]}
```

Parse Transforms - `lager`

The log dispatch clause

```
{clause, Line,
    %% Match
    [{tuple, Line, [{var, Line, PidVar}, {tuple, Line, [{var, Line, LevelVar},
                                                         {var, Line,
                                                           TracesVar}]}]}],
    % ...
    %
    %
    %
    %
    %
    %
    %
    %
    %
    %
    %
```

Parse Transforms - `lager`

The log dispatch clause

```
{clause, Line,
    %% Match
    [{tuple, Line, [{var, Line, PidVar}, {tuple, Line, [{var, Line, LevelVar},
                                                          {var, Line,
                                                            TracesVar}]}]}],
    %% Guards
    [[{op, Line, 'orelse',
          {op, Line, '/=', {op, Line, 'band', {var, Line, LevelVar},
                                                {integer, Line,
                                                  SeverityAsInt}},
                            {integer, Line, 0}},
          {op, Line, '/=', {var, Line, TracesVar}, {nil, Line}}}]],
    % ...
    %
    %
    %
    %
```

COMCAST

Parse Transforms - `lager`

The log dispatch clause

```
{clause, Line,
    %% Match
    [{tuple, Line, [{var, Line, PidVar}, {tuple, Line, [{var, Line, LevelVar},
                                                         {var, Line,
                                                           TracesVar}]}]}],
    %% Guards
    [[{op, Line, 'orelse',
          {op, Line, '/=', {op, Line, 'band', {var, Line, LevelVar},
                                             {integer, Line,
                                               SeverityAsInt}},
                           {integer, Line, 0}},
          {op, Line, '/=', {var, Line, TracesVar}, {nil, Line}}}]],
    %% Statements
    [
    %% do the call to lager:dispatch_log
    {call, Line, {remote, Line, {atom, Line, lager}, {atom, Line, do_log}},
        [             % ...
```

Parse Transforms - `parse_trans`
github.com/uwiger/parse_trans

Don’t worry...
Ulf has your back!

Parse Transforms - `parse_trans`

Rewriting lager’s transform with `parse_trans`

- Write transformations as “visitors” instead of manual recursion
- Return `NewForm` to replace the current form
- Return `continue` to recurse into subexpressions

```
parse_transform(AST, Options) ->
    %% Previously: walk_ast([], AST)
    parse_trans:plain_transform(fun do_transform/1, AST).
```

COMCAST

Parse Transforms - `parse_trans`

Rewriting lager’s transform with `parse_trans`

- Write transformations as “visitors” instead of manual recursion
- Return `NewForm` to replace the current form
- Return `continue` to recurse into subexpressions

```
parse_transform(AST, Options) ->
    %% Previously: walk_ast([], AST)
    parse_trans:plain_transform(fun do_transform/1, AST).
```

```
do_transform({call, _Line, {remote, _Line1, {atom, _Line2, lager},
                            {atom, _Line3, _Severity}}, _Arguments0} = Stmt) ->
    %% Do what we did before...
    transform_statement(Stmt);
do_transform(_) ->
    continue.
```

COMCAST

Parse Transforms - `parse_trans`

Other cool stuff!

- `ct_expand` - compile-time evaluation
- `exprecs` - generates record-accessor functions

COMCAST

Parse Transforms - Summary

Pros:

- Powerful
- Erlang syntax
- Compile-time computation

Parse Transforms - Summary

Pros:

- Powerful
- Erlang syntax
- Compile-time computation

Cons:

- Hides “magic”
- Difficult to write/debug
- Only modifies current module

COMCAST

Parse Transforms - Summary

Pros:

- Powerful
- Erlang syntax
- Compile-time computation

Cons:

- Hides “magic”
- Difficult to write/debug
- Only modifies current module

Good for:

- Injecting optimizations or new semantics
- Embedded DSLs
- Generating code in same module

Technique 3

Syntax Trees

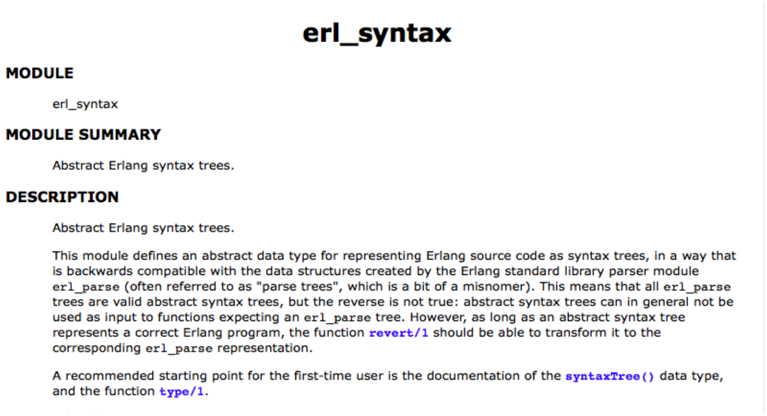
COMCAST

Syntax Trees

- Generates code by constructing syntax trees
- Operates over **Abstract Forms**

Syntax Trees - `erl_syntax`

- Datatype for **Abstract forms**
- Functions for every construct



erl_syntax

MODULE

erl_syntax

MODULE SUMMARY

Abstract Erlang syntax trees.

DESCRIPTION

Abstract Erlang syntax trees.

This module defines an abstract data type for representing Erlang source code as syntax trees, in a way that is backwards compatible with the data structures created by the Erlang standard library parser module `erl_parse` (often referred to as "parse trees", which is a bit of a misnomer). This means that all `erl_parse` trees are valid abstract syntax trees, but the reverse is not true: abstract syntax trees can in general not be used as input to functions expecting an `erl_parse` tree. However, as long as an abstract syntax tree represents a correct Erlang program, the function `revert/1` should be able to transform it to the corresponding `erl_parse` representation.

A recommended starting point for the first-time user is the documentation of the `syntaxTree()` data type, and the function `type/1`.

COMCAST

Syntax Trees - `erl_syntax`

- Creating nodes:

```
integer/1 float/1 atom/1 variable/1

list/2 cons/2 tuple/1

block_expr/1 clause/2,3 fun_expr/1

conjunction/1 disjunction/1

function/2 attribute/2 form_list/1
```

Syntax Trees - `erl_syntax`

- Creating nodes:

  `integer/1 float/1 atom/1 variable/1`

  `list/2 cons/2 tuple/1`

  `block_expr/1 clause/2,3 fun_expr/1`

  `conjunction/1 disjunction/1`

  `function/2 attribute/2 form_list/1`

- Inspecting nodes:

  `type/1 float_value/1 attribute_name/1`

COMCAST

Syntax Trees - `erl_syntax`

- Creating nodes:

  `integer/1 float/1 atom/1 variable/1`

  `list/2 cons/2 tuple/1`

  `block_expr/1 clause/2,3 fun_expr/1`

  `conjunction/1 disjunction/1`

  `function/2 attribute/2 form_list/1`

- Inspecting nodes:

  `type/1 float_value/1 attribute_name/1`

- Converting:

  `abstract/1 revert/1 revert_forms/1`

COMCAST

Syntax Trees - `erl_syntax`

- Creating nodes:

  `integer/1 float/1 atom/1 variable/1`

  `list/2 cons/2 tuple/1`

  `block_expr/1 clause/2,3 fun_expr/1`

  `conjunction/1 disjunction/1`

  `function/2 attribute/2 form_list/1`

- Inspecting nodes:

  `type/1 float_value/1 attribute_name/1`

- Converting:

  `abstract/1 revert/1 revert_forms/1`

- Traversing: `subtrees/1`

COMCAST

Syntax Trees - Neotoma v1

Getting out my shinebox

COMCAST

Syntax Trees - Neotoma v1

github.com/seancribbs/neotoma

```
quoted_string <- single_quoted_string / double_quoted_string
%{
  used_combinator(p_string),
  lists:flatten(["p_string(<<\"",
   escape_string(unicode:characters_to_list(proplists:get_value(string, Node))),
   "\">>)"])
%};
```

COMCAST

Syntax Trees - Neotoma v1

github.com/seancribbs/neotoma

```
quoted_string <- single_quoted_string / double_quoted_string
%{
  used_combinator(p_string),
  lists:flatten(["p_string(<<\"",
   escape_string(unicode:characters_to_list(proplists:get_value(string, Node))),
   "\">>)"])
%};
```

```
generate_module_attrs(ModName, Combinators) ->
    ["-module(", atom_to_list(ModName) ,").\n",
     "-export([parse/1,file/1]).\n",
     [ generate_combinator_macro(C) || Combinators /= undefined,
                                       C <- Combinators ],
     "\n"
     ].
```

Syntax Trees - Neotoma v2

```
quoted_string <- single_quoted_string / double_quoted_string
%{
  #string{string = unicode:characters_to_binary(proplists:get_value(string,
                                                                     Node)),
          index = Idx}
%};
```

Syntax Trees - Neotoma v2

```erlang
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

COMCAST

Syntax Trees - Neotoma v2

```erlang
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

```erlang
generate(#string{string=S}, InputName, Success, Fail) ->
```

COMCAST

Syntax Trees - Neotoma v2

```
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

```
generate(#string{string=S}, InputName, Success, Fail) ->
    Literal = abstract(S), % convert term to syntaxTree()
    RestName = variable(new_name("Input")),
```

COMCAST

Syntax Trees - Neotoma v2

```
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

```
generate(#string{string=S}, InputName, Success, Fail) ->
    Literal = abstract(S),
    RestName = variable(new_name("Input")),
    case_expr(InputName, % case ... of ... end
```

COMCAST

Syntax Trees - Neotoma v2

```
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

```
generate(#string{string=S}, InputName, Success, Fail) ->
    Literal = abstract(S),
    RestName = variable(new_name("Input")),
    case_expr(InputName,
              [clause([binary([binary_field(Literal, [atom("binary")]),
                               binary_field(RestName, [atom("binary")])])],
                      none,
                      Success(Literal, RestName)), % success path!
```

Syntax Trees - Neotoma v2

```
case Input of
    <<"a string"/binary, Input1/binary>> -> % ...do the success path;
    _ -> % ...do the failure path
end
```

```
generate(#string{string=S}, InputName, Success, Fail) ->
    Literal = abstract(S),
    RestName = variable(new_name("Input")),
    case_expr(InputName,
              [clause([binary([binary_field(Literal, [atom("binary")]),
                               binary_field(RestName, [atom("binary")])])],
                      none,
                      Success(Literal, RestName)),
               clause([underscore()], none,
                      Fail(InputName, error_reason({string, S})))]); % fail path!
```

COMCAST

Syntax Trees - `mochiglobal`

github.com/mochi/mochiweb

- Memoizes frequently used values in code
- Good for high-read, low-write scenarios

```
%% @doc Store term V at K, replaces an existing term if present.
put(K, V) ->
    put(K, V, key_to_module(K)).
```

COMCAST

Syntax Trees - `mochiglobal`

github.com/mochi/mochiweb

- Memoizes frequently used values in code
- Good for high-read, low-write scenarios

```
%% @doc Store term V at K, replaces an existing term if present.
put(K, V) ->
    put(K, V, key_to_module(K)).

put(_K, V, Mod) ->
    Bin = compile(Mod, V),
    code:purge(Mod),
    {module, Mod} = code:load_binary(Mod, atom_to_list(Mod) ++ ".erl", Bin),
    ok.
```

Syntax Trees - `mochiglobal`

```
-spec compile(atom(), any()) -> binary().
compile(Module, T) ->
    {ok, Module, Bin} = compile:forms(forms(Module, T),
                                      [verbose, report_errors]),
    Bin.
```

COMCAST

Syntax Trees - `mochiglobal`

```
-spec compile(atom(), any()) -> binary().
compile(Module, T) ->
    {ok, Module, Bin} = compile:forms(forms(Module, T),
                                      [verbose, report_errors]),
    Bin.
```

```
forms(Module, T) ->
    [erl_syntax:revert(X) || X <- term_to_abstract(Module, term, T)].
```

Syntax Trees - `mochiglobal`

```
term_to_abstract(Module, Getter, T) ->
    [%% -module(Module).
     erl_syntax:attribute(
       erl_syntax:atom(module),
       [erl_syntax:atom(Module)]),
```

Syntax Trees - `mochiglobal`

```erlang
term_to_abstract(Module, Getter, T) ->
    [%% -module(Module).
     erl_syntax:attribute(
       erl_syntax:atom(module),
       [erl_syntax:atom(Module)]),
     %% -export([Getter/0]).
     erl_syntax:attribute(
       erl_syntax:atom(export),
       [erl_syntax:list(
         [erl_syntax:arity_qualifier(
            erl_syntax:atom(Getter),
            erl_syntax:integer(0))])]),
```

Syntax Trees - `mochiglobal`

```erlang
term_to_abstract(Module, Getter, T) ->
    [%% -module(Module).
     erl_syntax:attribute(
       erl_syntax:atom(module),
       [erl_syntax:atom(Module)]),
     %% -export([Getter/0]).
     erl_syntax:attribute(
       erl_syntax:atom(export),
       [erl_syntax:list(
         [erl_syntax:arity_qualifier(
            erl_syntax:atom(Getter),
            erl_syntax:integer(0))])]),
     %% Getter() -> T.
     erl_syntax:function(
       erl_syntax:atom(Getter),
       [erl_syntax:clause([], none, [erl_syntax:abstract(T)])])].
```

Syntax Trees - `merl`

github.com/richcarl/merl

Don’t worry…

Richard has your back!

COMCAST

Syntax Trees - `merl`

github.com/richcarl/merl

Don't worry...

Richard has your back!

Combines strategies of:

- Macros - `?Q(Text)`, `?Q(Text,Env)`
- Parse Transforms
- Syntax Tree Generation

Syntax Trees - `merl`

github.com/richcarl/merl

Don't worry. . .
Richard has your back!

Combines strategies of:

- Macros - `?Q(Text)`, `?Q(Text,Env)`
- Parse Transforms
- Syntax Tree Generation

Included in OTP 18!!!!

COMCAST

Syntax Trees - erlydtl

github.com/erlydtl/erlydtl

- Implements Django-style templates
- Moved from using `erl_syntax` to `merl` last year

```
Function1 = erl_syntax:function(
              erl_syntax:atom(FunctionName),
              [erl_syntax:clause(
                 [erl_syntax:variable("_Variables")],
                 none,
                 [erl_syntax:application(
                    none, erl_syntax:atom(FunctionName),
                    [erl_syntax:variable("_Variables"), erl_syntax:list([])])
                 ])
              ]),
```

COMCAST

Syntax Trees - erlydtl

github.com/erlydtl/erlydtl

- Implements Django-style templates
- Moved from using `erl_syntax` to `merl` last year

```
Function1 = erl_syntax:function(
              erl_syntax:atom(FunctionName),
              [erl_syntax:clause(
                 [erl_syntax:variable("_Variables")],
                 none,
                 [erl_syntax:application(
                    none, erl_syntax:atom(FunctionName),
                    [erl_syntax:variable("_Variables"), erl_syntax:list([])])
                 ])
              ]),
```

```
Function1 = ?Q("_@FunctionName@(_Variables) -> _@FunctionName@(_Variables, [])"),
```

Syntax Trees - Summary

Pros:

- Most versatile
- Powerful tools
- Multiple output destinations

COMCAST

Syntax Trees - Summary

Pros:

- Most versatile
- Powerful tools
- Multiple output destinations

Cons:

- Verbose
- Many manual steps
- AST understanding may be needed

COMCAST

Syntax Trees - Summary

Pros:

- Most versatile
- Powerful tools
- Multiple output destinations

Cons:

- Verbose
- Many manual steps
- AST understanding may be needed

Good for:

- Implementing new languages & External DSLs
- “Run-time” code generation

Conclusion

Metaprogramming Erlang is great!

COMCAST

Conclusion

Metaprogramming Erlang is great!

Use `erl_syntax`, `parse_trans`, and `merl`!

COMCAST

Conclusion

Metaprogramming Erlang is great!

Use `erl_syntax`, `parse_trans`, and `merl`!

Build cool tools!

COMCAST

Thanks!

Twitter / Github: seancribbs