parse_transform
EXPLAINED

Denys Gonchar
denys.gonchar@erlang-solutions.com

1.

WHAT IS ~~PARSE~~ TRANSFORMATION

DON'T TELL ME THAT, I HAVEN'T **WARNED** YOU :)

“

Programmers are strongly advised not to engage in parse transformations. No support is offered for problems encountered.

PARSE_TRANSFORM IS A METAPROGRAMMING TECHNIQUE

“Parse transformations are used if a programmer wants to use Erlang syntax, but with different semantics. The original Erlang code is then transformed into other Erlang code.”

© erlang documentation

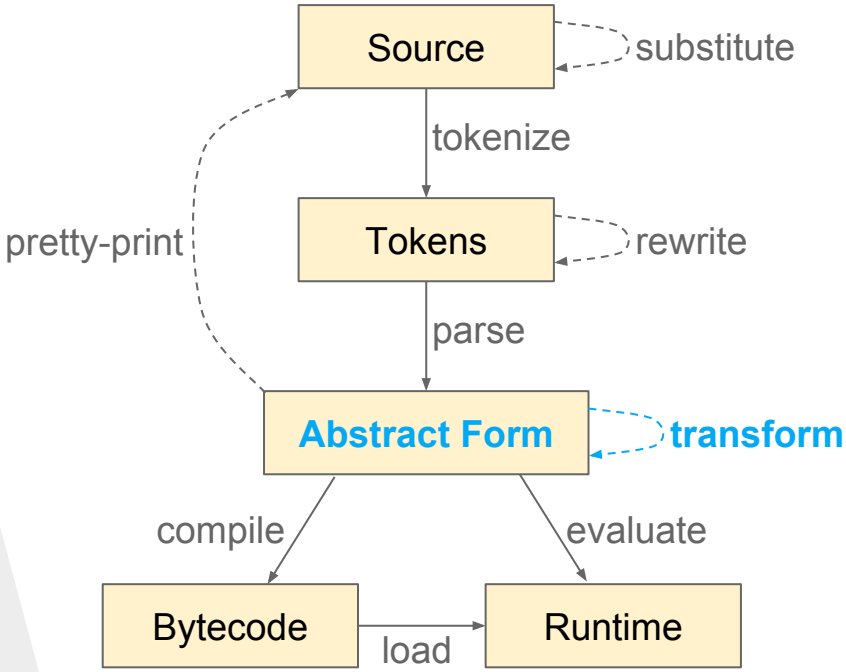
BUT DOES THIS
MAKE MUCH SENSE?

PARSE_TRANSFORM IN SIMPLE WORDS

- Parse is buzzword, you don't do parsing
- The code is parsed into AST before you enter the game
- You can transform AST before the next compilation step

AST is Abstract Syntax Tree, Abstract Format (AF) is just another name for it.

Reference for Abstract Format is here



AST EXAMPLE

```
-module(test).
-compile([{parse_transform, my_transform},export_all]).

print(X) when is_atom(X) ->
  io:format("~p", [X]).
```

```
[ {attribute,1,file,{".../test.erl",1}},
  {attribute,1,module,test},
  {attribute,2,compile,[export_all]},
  {function,4,print,1,[
    {clause,4,
      [{var,4,'X'}],
      [[{call,4,{atom,4,is_atom},[{var,4,'X'}]}]],
      [
        {call,5,{remote,5,{atom,5,io},{atom,5,format}},[
          {string,5,"~p"},
          {cons,5,{var,5,'X'},{nil,5}} ]} ]} ]},
  {eof,7}
]
```

2.

IS ANYONE USING IT IN THE REAL WORLD?

WHY SHOULD I CARE ABOUT ABSTRACT FORM

Obviously, if you need to create parse_transform module. but even if you don't, it's still valuable to take a look on Abstract Form. it is a pretty basic thing in erlang world:

- debug _info added to beam files is in a fact AF.
- AF can be (and it is) used for static code analysis (e.g. xref).
- AF can be used for dynamic code/modules generation (erlang shell, meck?).

WHERE PARSE_TRANSFROM IS USED

Just a small list of project with links to code

- lager - yes, yes, yes. your nice logging is based on parse_tranform
- QuckCheck/PropEr - they both use parse_transform but QuickCheck does it much more. I will tell more about it later.
- Erlando
- seqbind
- guardian

3.

HOW CAN I USE IT?

IMPLEMENTING PARSE_TRANSFORM MODULE

To create your custom parse_transfrom module you need to do the next:

- Implement and export **Module:parse_transform/2** function
- Pass **{parse_transform,Module}** option to compiler or add **-compile({parse_transform,Module}).** line in you erl/hrl file.

Also keep in mind that it's not that easy to read this code, give others a chance to understand your transformations:

- implement transformation logic in a separate (dedicated for this only) module.
- It's always better to have multiple small & pretty parse_transform modules than one big & monstrous.

IMPLEMENTING PARSE_TRANSFORM MODULE
EXAMPLE OF PARSE_TRANSFORM MODULE

```
-module(my_transform).
-export([parse_transform/2]).

parse_transform(Code, Opts) ->
  io:format("~p~n", [{Code, Opts}]),
  Code.


%% Also you can check erl_id_trans module,
%% which is included in OTP as an example
```

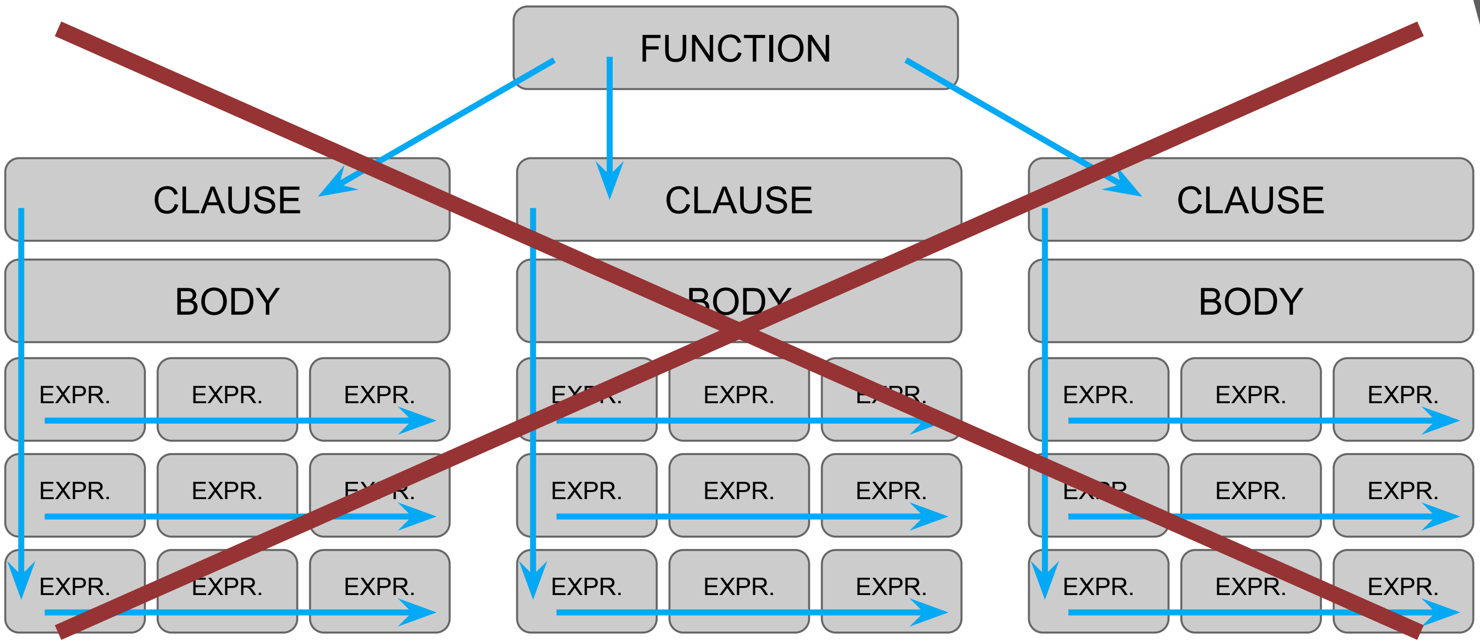
IMPLEMENTING PARSE_TRANSFORM MODULE
EXAMPLE OF PARSE_TRANSFORM MODULE

Before reinvent the bicycle (if it's not just for fun), check what other libraries offers to you.

Mr. Ulf Wiger did some nice job in parse_trans.

Also it's worth to mention standard erl_syntax library.

IMPLEMENTING PARSE_TRANSFORM MODULE
TRAVERSING THE ABSTRACT SYNTAX TREE



If you do it **manually,** mostly like you do it **wrong**

IMPLEMENTING PARSE_TRANSFORM MODULE
TRAVERSING THE ABSTRACT SYNTAX TREE

For the right approach, check this parse_trans example.
It converts all instances of P ! Msg to gproc:send(P, Msg):

```
parse_transform(Forms, _Options) ->
  parse_trans:plain_transform(fun do_transform/1, Forms).

do_transform({'op', L, '!', Lhs, Rhs}) ->
  [NewLhs] = parse_trans:plain_transform(fun do_transform/1, [Lhs]),
  [NewRhs] = parse_trans:plain_transform(fun do_transform/1, [Rhs]),
  {call, L, {remote, L, {atom, L, gproc}, {atom, L, send}},
         [NewLhs, NewRhs]};
do_transform(_) ->
  continue.
```

IMPLEMENTING PARSE_TRANSFORM MODULE
STUFF YOU WOULD WANT TO READ BEFORE YOU START

I find these 3 presentations very helpful:

- Techniques for Metaprogramming in Erlang
- HERE BE DRAGONS: charting parse transforms in Erlang
- Fear Not or a Brief Introduction to Parse Transformations

Unfortunately this topic is not well described in official documentation, sometimes it even seems to be intentional.

The general recommendation - learn thru practice.

4.

EXPOSING WHAT IS DONE TO YOUR CODE

THE FUN PART OF PARSE TRANSFORMATION

If you want to check how the code (yes, I mean erlang code here, not AST) looks like after all transformations, you can try the "-P" compilation option.

And it works like a charm... as long as you have just one parse_transform applied, but what if you have more?

DEALING WITH MULTIPLE TRANSFORMATION APPLIED TO YOUR CODE

Here we have 2 problems:

- There's no straightforward way to identify which parse transformations have been applied and in which sequence. Erlang compiler is not verbose about it
- "-P" restores code for you from the final AST, after all transformations are done. but how to get what was changed where?

It's really funny, but to solve these 2 problems we would need to... parse_transform

LISTING ALL THE APPLIED TRANSFORMATIONS

compile.erl source code can answer a lot of your questions, including this one.

The order of the applied transformations is the following:

- All the transformation mentioned externally as compilation options (in the same sequence as they are passed to compiler)
- All the transformations mentioned in the source file (in the sequence as they appear in code)

So knowing compilation options and setting trace for foldl_transform function you can reconstruct the full list of transformations. The code is here.

FIGURING OUT WHAT WAS CHANGED WHERE

The nice thing about AST is that you can reconstruct code from it.

Also compiler doesn't check how many times you apply one and the same transformation.

So what you need is parse_transform module that restores the source code for you. Than you just add your parse_transform compilation options between other transformations and comparing the code. It's as simple as that :)

5.

CONCLUSION

THING YOU MUST AWARE OF

- Abstract Format can change!
- Application of different transformations can have some unexpected effects
- **parse_transform mechanism is a potential security threat. Someone could still your code or add some backdoors. So use only trusted parse_transform libraries!**

6.

DEMO