Fear Not

or a
Brief Introduction to Parse Transformations

Erlang Factory SF 2012
Yurii Rashkovskii

Continuous Reverse Plastic Surgery

Makes nice code… **uglier**

Why would you do that?

- You don’t want to **write** ugly code
- You don’t want to **see** ugly code
- Yet, you **need** that ugly code!

Example?

SeqBind

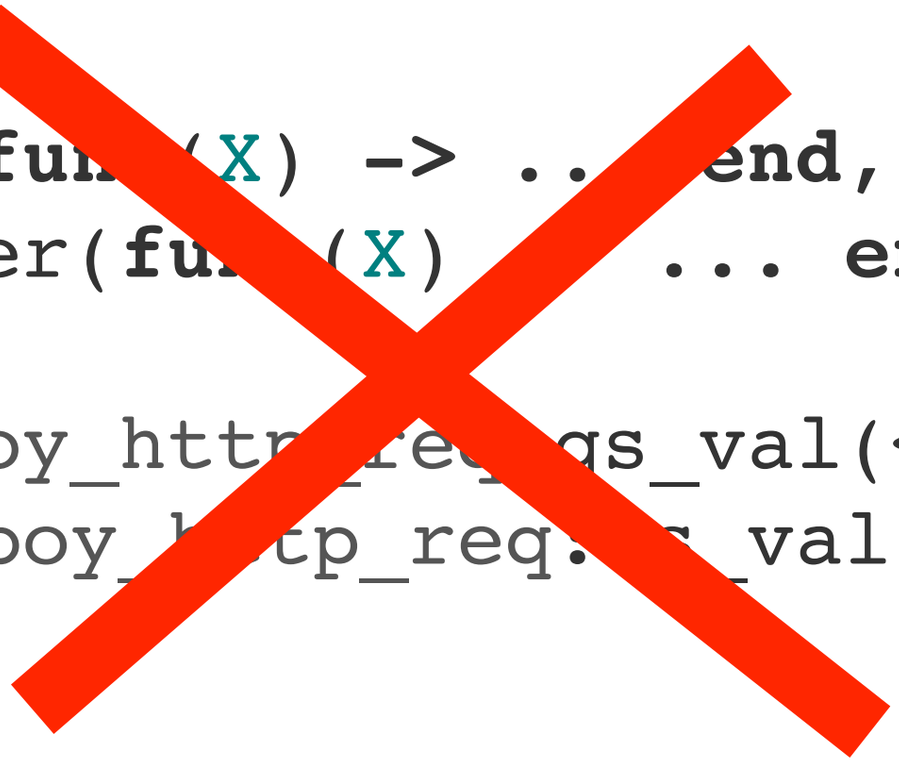
https://github.com/spawngrid/seqbind

```
L1 = lists:map(fun (X) -> ... end, L),
L2 = lists:filter(fun (X) -> ... end, L1)
%% or
{Q,Req1} = cowboy_http_req:qs_val(<<"q">>,Req),
{Id,Req2} = cowboy_http_req:qs_val(<<"id">>,Req1)
```

SeqBind

https://github.com/spawngrid/seqbind

```
L1 = lists:map(fun (X) -> ... end, L),
L2 = lists:filter(fun (X) -> ... end, L1)
%% or
{Q,Req1} = cowboy_http_req:qs_val(<<"q">>,Req),
{Id,Req2} = cowboy_http_req:qs_val(<<"id">>,Req1)
```



SeqBind

https://github.com/spawngrid/seqbind

```
L@ = lists:map(fun (X) -> ... end, L@),
L@ = lists:filter(fun (X) -> ... end, L@)
%% or
{Q,Req@} = cowboy_http_req:qs_val(<<"q">>,Req@),
{Id,Req@} = cowboy_http_req:qs_val(<<"id">>,Req@)
```

SeqBind

https://github.com/spawngrid/seqbind

```
L@1 = lists:map(fun(X) -> ... end, L@0),
L@2 = lists:filter(fun(X) -> ... end, L@1),
%% or
{Q,Req@1} = cowboy_http_req:qs_val(<<"q">>,Req@0),
{Id,Req@2} = cowboy_http_req:qs_val(<<"id">>,
Req@1).
```

So, how do you do this?

Yo man I heard you like Erlang, so we put Erlang in Erlang so you can Erlang while you Erlang

Yo man I heard you like Erlang, so we put Erlang in Erlang so you can Erlang while you Erlang

...in other words, meta programming!

Yo man I heard you like Erlang, so we put Erlang in Erlang so you can Erlang while you Erlang

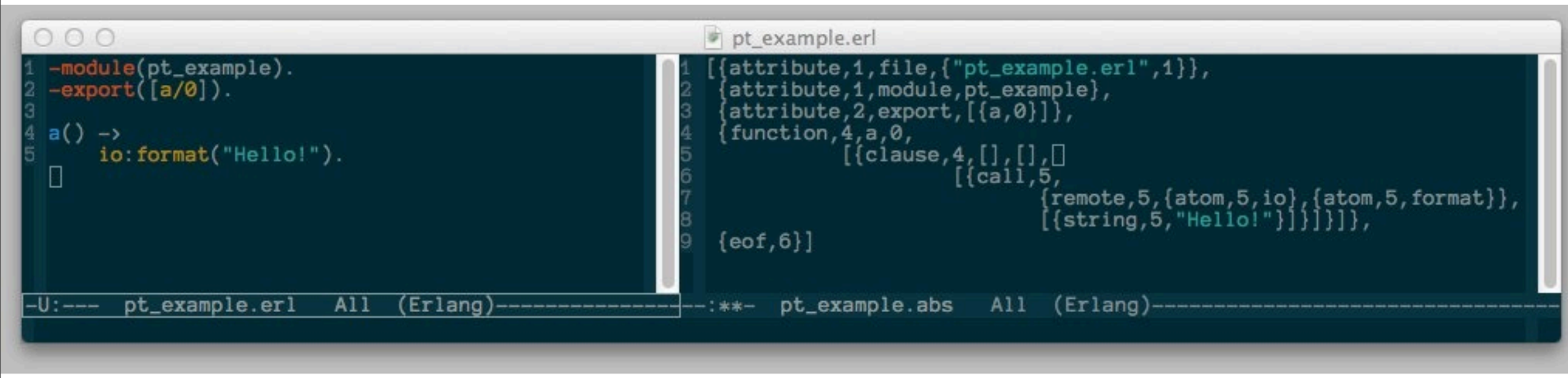
...in other words, meta programming!

well, sort of

~~Love~~ Parse transformation is...

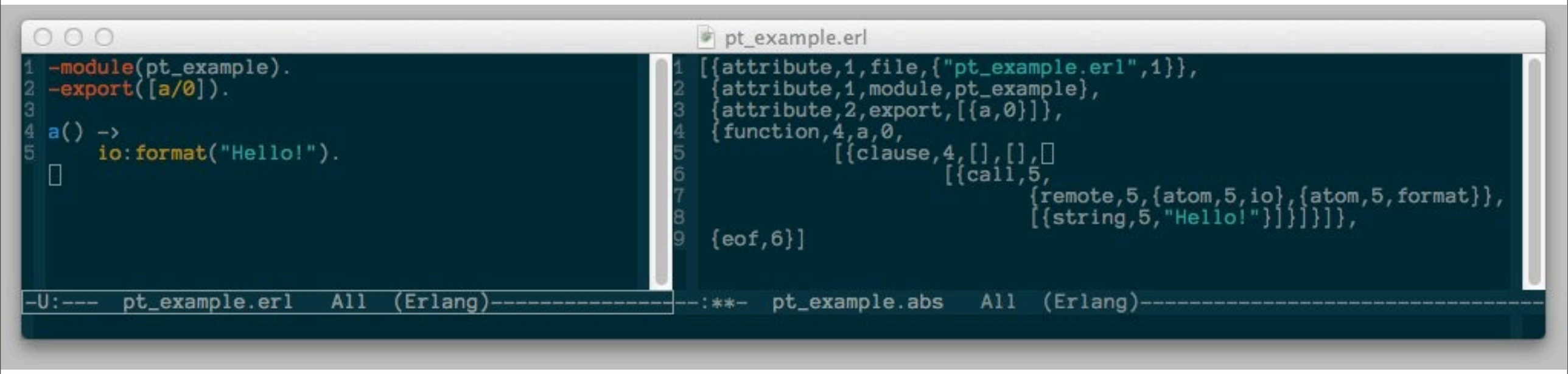
- An Erlang module
- That takes an Erlang module (in abstract format)
- And rewrites it

Abstract Format?



Parse tree representation

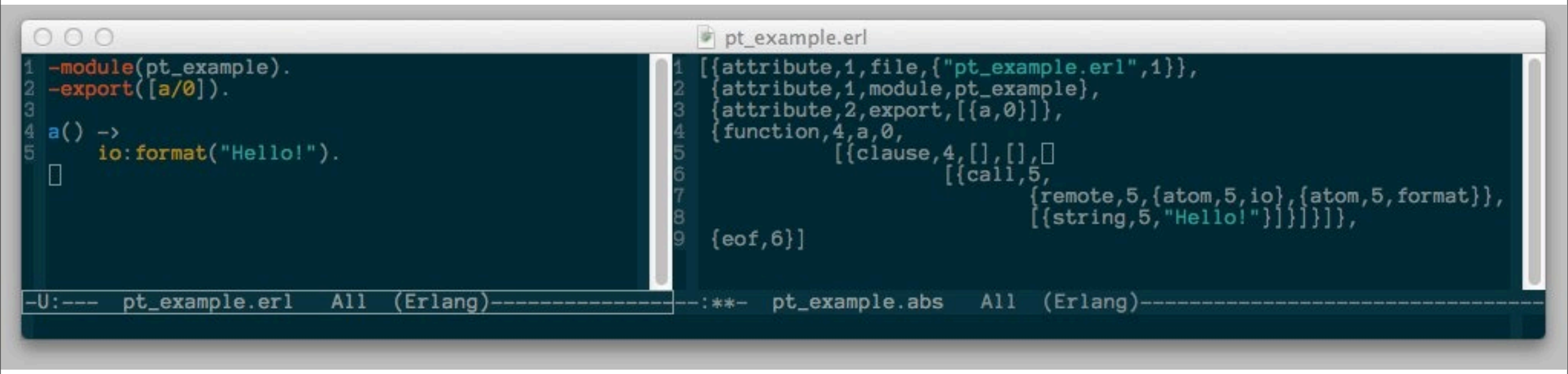
Abstract Format?



Parse tree representation

Essentially, a list of tuples!

Abstract Format?



Parse tree representation

Essentially, a list of tuples!

Specification:

http://goo.gl/T29Cw

Example

```
myfun({command, X}, Args) ->
    io:format(
      "Command: ~p, Args: ~p",
      [X, Args]).
```

Example

```
{function,4,myfun,2,
 [{clause,4,
   [{tuple,4,[{atom,4,command},{var,4,'X'}]},
    {var,4,'Args'}],
   [],
   [{call,5,
     {remote,5,{atom,5,io},{atom,5,format}},
     [{string,5,"Command: ~p, Args: ~p"},
      {cons,5,{var,5,'X'},
              {cons,5,{var,5,'Args'},{nil,5}}}]}]}]}
```

Example

myfun/2

```
{function,4,myfun,2,
 [{clause,4,
   [{tuple,4,[{atom,4,command},{var,4,'X'}]},
    {var,4,'Args'}],
   [],
   [{call,5,
     {remote,5,{atom,5,io},{atom,5,format}},
     [{string,5,"Command: ~p, Args: ~p"},
      {cons,5,{var,5,'X'},
              {cons,5,{var,5,'Args'},{nil,5}}}]}]}]}
```

Example

```
{function,4,myfun,2,
 [{clause,4,
   [{tuple,4,[{atom,4,command},{var,4,'X'}]},
    {var,4,'Args'}],
   [],
   [{call,5,
     {remote,5,{atom,5,io},{atom,5,format}},
     [{string,5,"Command: ~p, Args: ~p"},
      {cons,5,{var,5,'X'},
              {cons,5,{var,5,'Args'},{nil,5}}}]}]}]}
```

{command, X}, Args

Example

```
{function,4,myfun,2,
 [{clause,4,
   [{tuple,4,[{atom,4,command},{var,4,'X'}]},
    {var,4,'Args'}],
   [],
   [{call,5,
     {remote,5,{atom,5,io},{atom,5,format}},
     [{string,5,"Command: ~p, Args: ~p"},
      {cons,5,{var,5,'X'},
              {cons,5,{var,5,'Args'},{nil,5}}}]}]}]}
```

io:format

Example

```
{function,4,myfun,2,
 [{clause,4,
    [{tuple,4,[{atom,4,command},{var,4,'X'}]},
     {var,4,'Args'}],
    [],
    [{call,5,
      {remote,5,{atom,5,io},{atom,5,format}},
      [{string,5,"Command: ~p, Args: ~p"},
       {cons,5,{var,5,'X'},
               {cons,5,{var,5,'Args'},{nil,5}}}]}]}]}
```

[X|[Args|[]]] ... or simply [X, Args]

Fully reconstructable

with `erl_pp:form/1`

```
myfun({command,X}, Args) ->
      io:format(
        "Command: ~p, Args: ~p",
        [X,Args]).
```

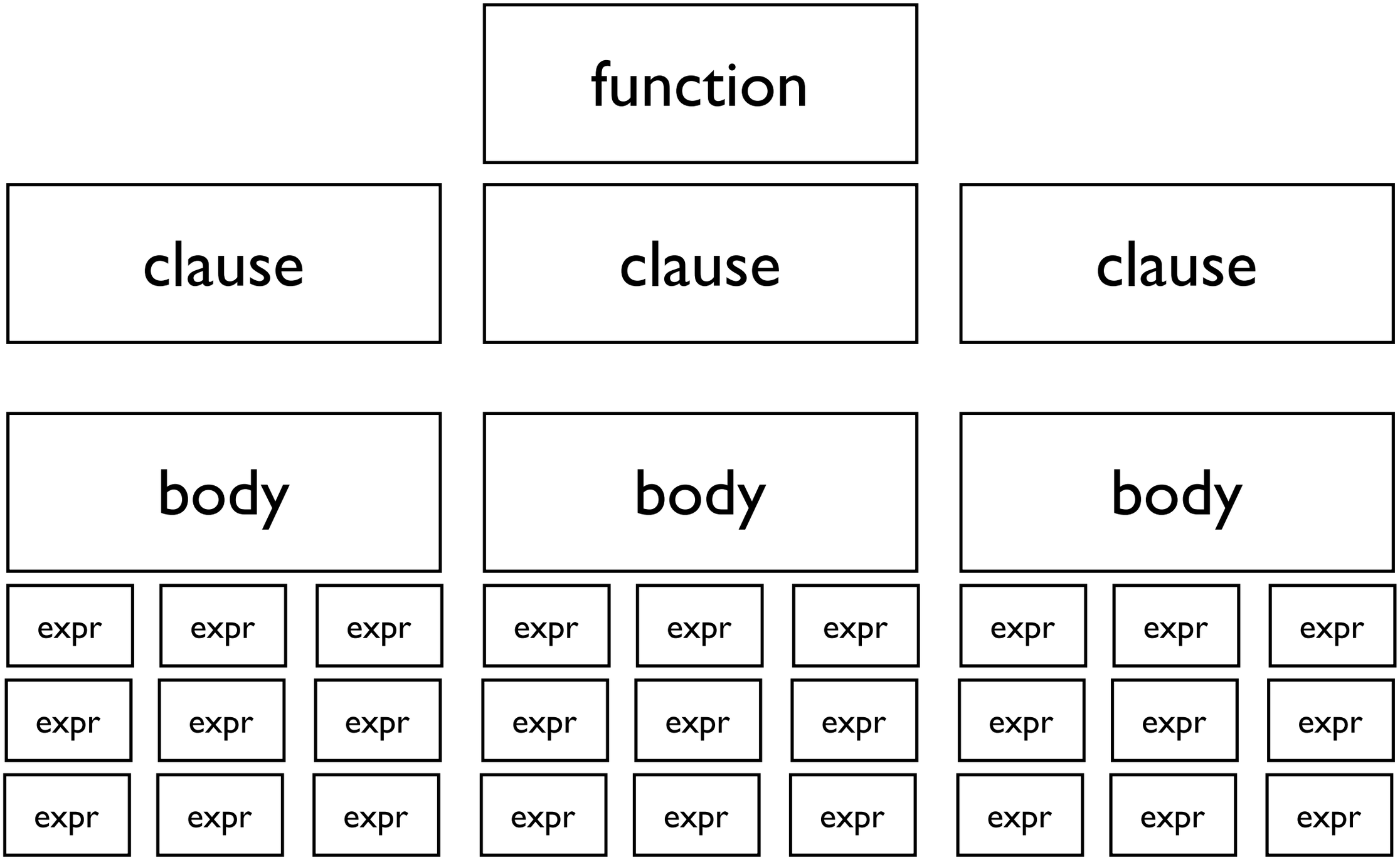
General Workflow

- Traverse the whole parse tree
- When matching something you need to rewrite...
- Rewrite it!

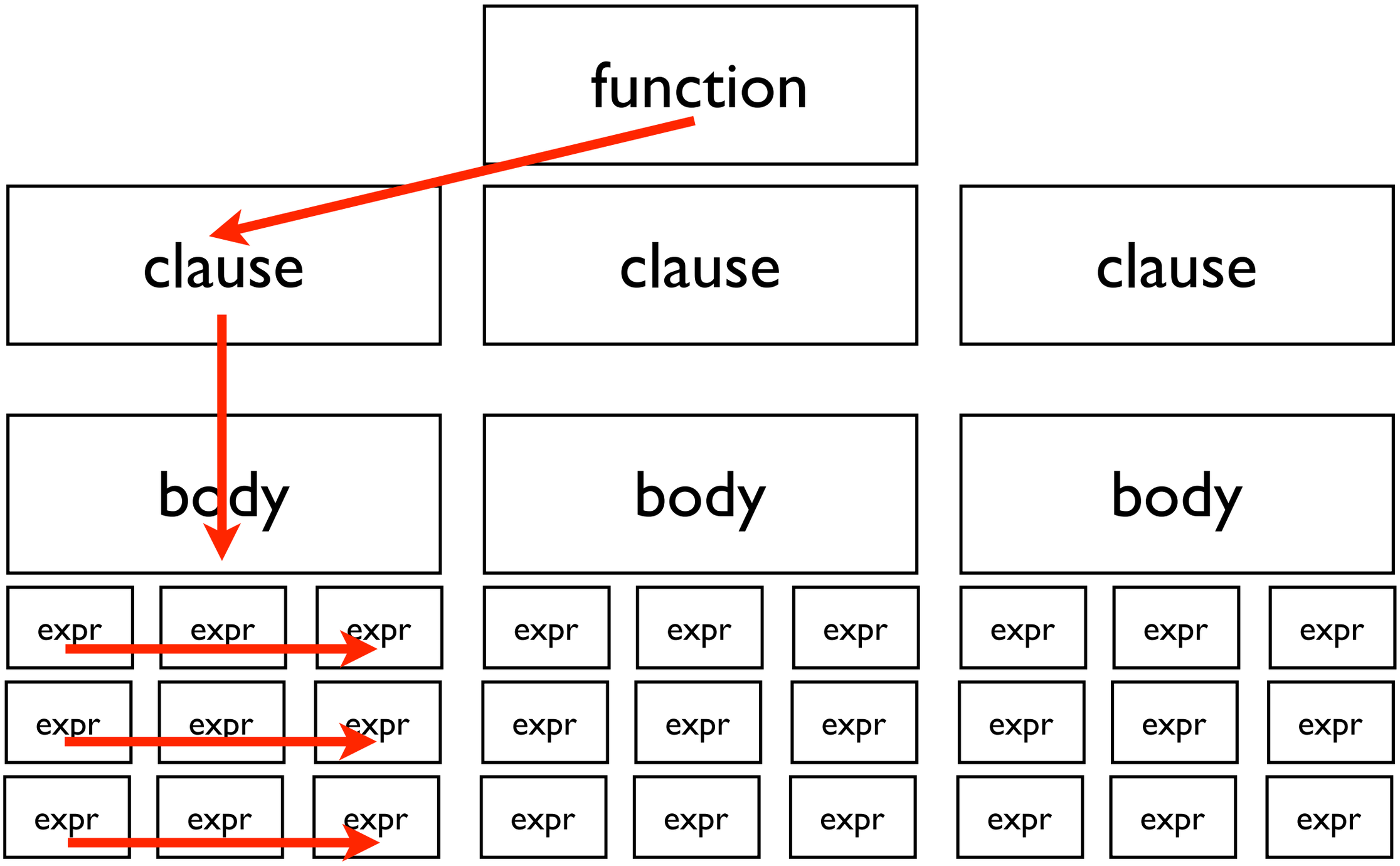
Example: Rewriting an expression

Lets say we want to rewrite `magic_function(...)` calls to `some_module:not_so_magic(...)`

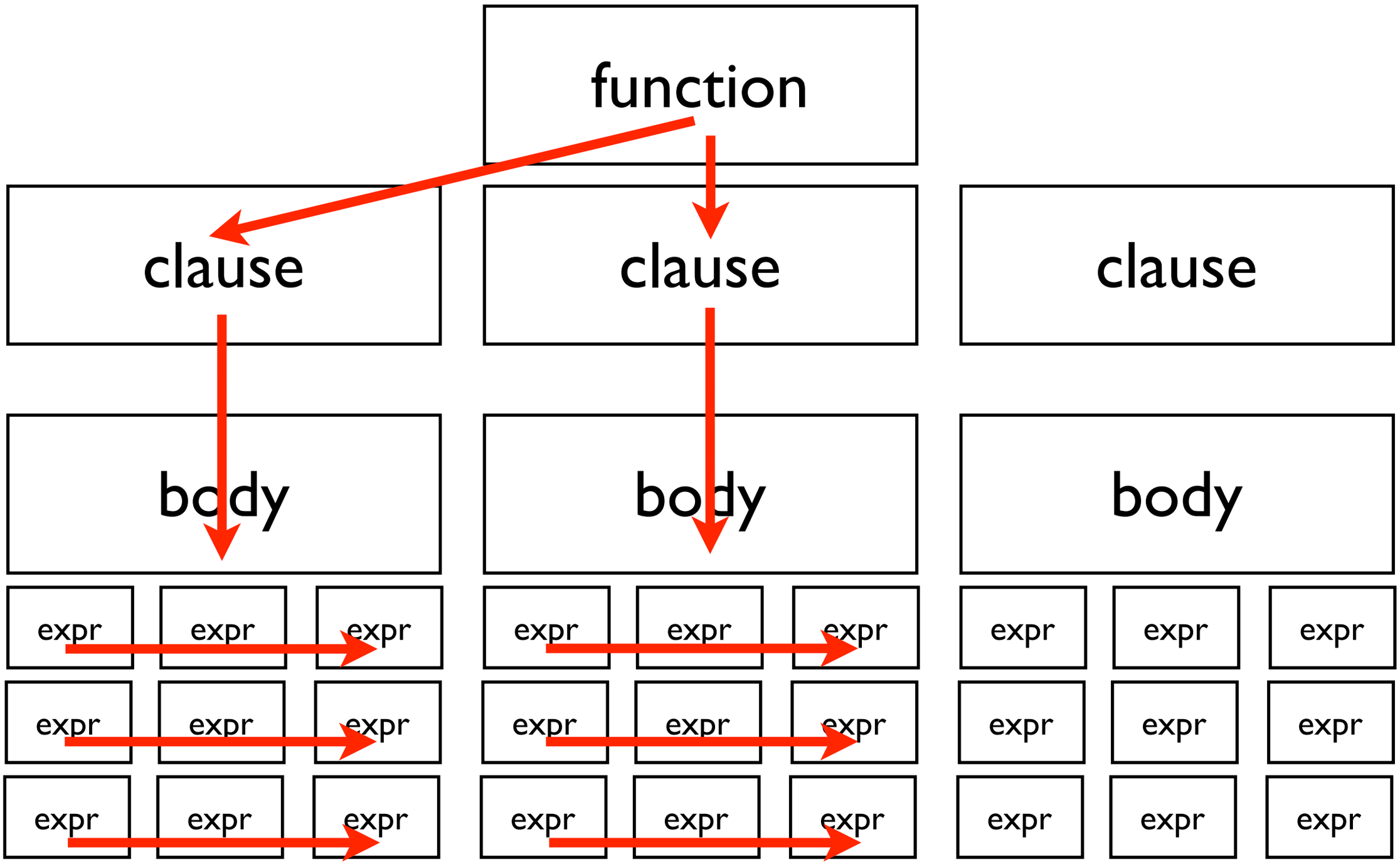
Example: Rewriting an expression



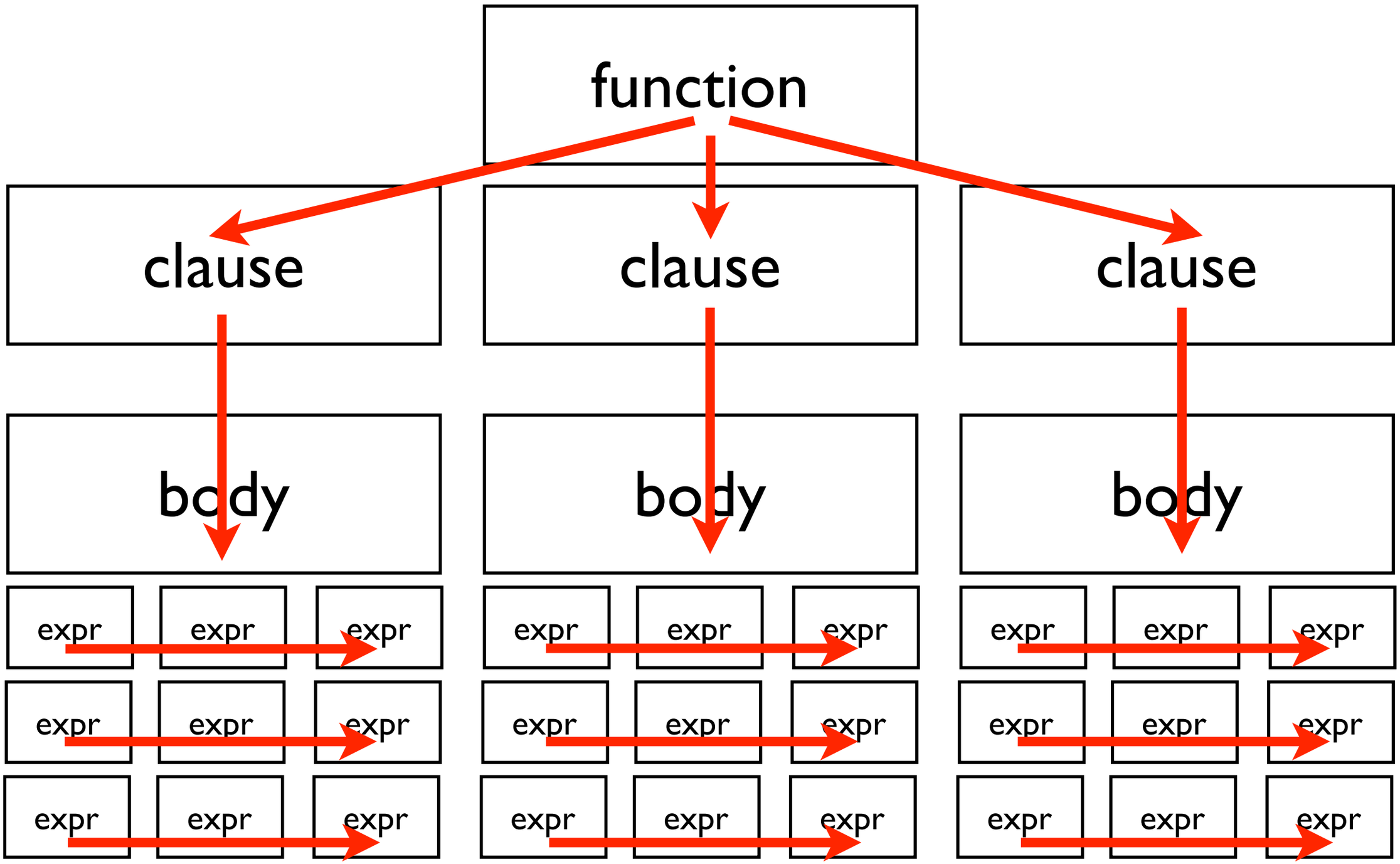
Example: Rewriting an expression



Example: Rewriting an expression



Example: Rewriting an expression



Gah!

(meta programming made hard)

Nah!

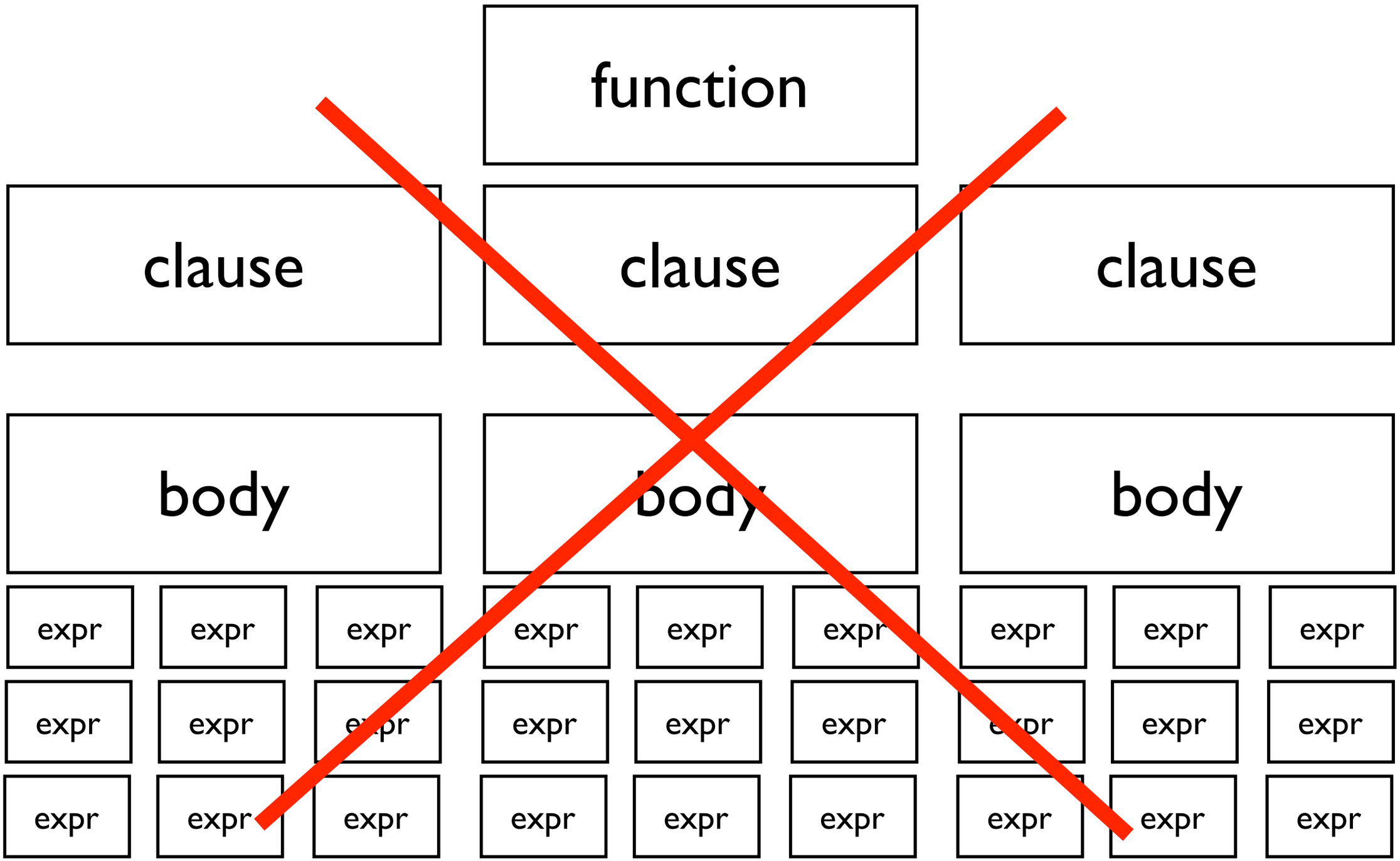
(meta programming shouldn't be hard)

parse_trans

https://github.com/esl/parse_trans

- Does traversing for you. Go deep fast!
- Allows to inject new forms (before/after)
- Comes with interesting examples!

Example: Rewriting an expression



Example: Rewriting an expression

```
do_transform(application,
             {call, L0,
             {atom, L1, magic_function}, Args},
              _Context, State) ->
 {{call, L0, {remote, L1,
              {atom, L1, some_module},
              {atom, L1, not_so_magic}},
             Args},
  true, State}
```

Example: Rewriting an expression

match magic_function(...)

```
do_transform(application,
             {call, L0,
             {atom, L1, magic_function}, Args},
              _Context, State) ->
 {{call, L0, {remote, L1,
              {atom, L1, some_module},
              {atom, L1, not_so_magic}},
             Args},
  true, State}
```

change it to

some_module:not_so_magic(...)

Spotlight: **Exprecs**

- Records is a syntax sugar for tuples
- Exprecs gets you a runtime access to records's structure
- Included into parse_trans

Mad Scientist's Tip

Your input source code doesn't actually have to be compilable as is...

Exportie

https://github.com/spawngrid/exportie

```
-export([f/1]).

%% ...

f(A) when is_list(A) ->
        A;
f(A) when is_binary(A) ->
        [A].
```

Exportie

https://github.com/spawngrid/exportie

```
export@(f(A)) when is_list(A) ->
        A;
export@(f(A)) when is_binary(A) ->
        [A].
```

How does that work?

Luckily, Exportie is very simple.
We can dive into the code right now.

Exportie: State

```
-record(state,
        {
          function_name,
          arity,
          exports = [],
          export = 'export@',
          options
        }).
```

Exportie: Entry Point

```
parse_transform(Forms, Options) ->
    {Forms1, State} = parse_trans:transform(
                                    fun do_transform/4,
                                    #state{ options = Options },
                                    Forms, Options),
    Forms2 = lists:foldl(fun({M,A},Acc) ->
                                parse_trans:export_function(M,A,Acc)
                            end, Forms1, State#state.exports),
    parse_trans:revert(Forms2).
```

Exportie: Custom Name

```
-exportie('export_this').
```

('export@' by default)

```
do_transform(attribute,{attribute, _, exportie, Custom} = Form,
             _Context, #state{} = State) ->
    {Form, false, State#state{ export = Custom }};
```

Exportie: Function

```
export_this(myfun()) -> ok.
```

```
do_transform(function,
             {function, Line, Export, 1, Cs},
             Context,
              #state{ exports = Exports,export = Export }=State) ->
    {Cs1, _Rec, State1} = transform(fun export_transform/4,
                                    State, Cs, Context),
    Form = {function, Line,
            State1#state.function_name, State1#state.arity,
            Cs1},
    {Form,
     false,
     State#state{
      exports = [{State1#state.function_name, State1#state.arity}|
Exports]
     }};
```

Exportie: Function

```
export_this(myfun()) -> ok.
```

```
export_transform(clause,
                 {clause, Line, H, G, B},
                 Context, State) ->
    {H1, Rec, State1} =
        transform(fun export_transform/4, State, H, Context),

    {{clause, Line, H1, G, B}, Rec, State1};

export_transform(application,
                 {call, _Line, {atom, _, Name}, Args},
                 _Context,
                 #state{} = State) ->
    {Args, false,
     State#state{ function_name = Name, arity = length(Args) }};

export_transform(_Type, Form, _Context, State) ->
    {Form, true, State}.
```

Exportie: Done!

```
do_transform(_Type, Form, _Context, State) ->
    {Form, true, State}.
```

Other uses?

- Compile-time analysis and extractions
- Code instrumentation (see https://github.com/hyperthunk/annotations)
- Domain Specific Languages on top of Erlang

Caution!

Do not abuse this technique.

You will regret this.

Thanks!

Questions?