Hacking Erlang

building strange and magical creations

Things Worth Trying:

- code injection
- meta programming
- reverse engineering byte code
- anything that makes Ericsson cringe...

Step 1

understanding the abstract format

The Abstract Format

- a tree-like structure representing parsed Erlang code

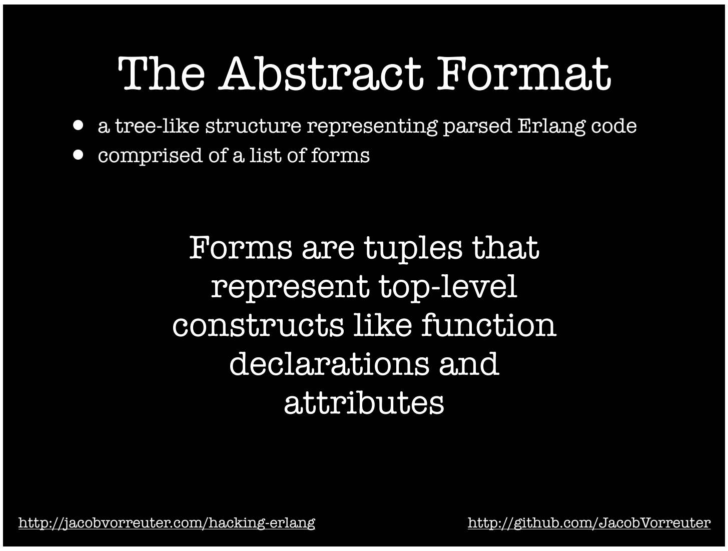
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

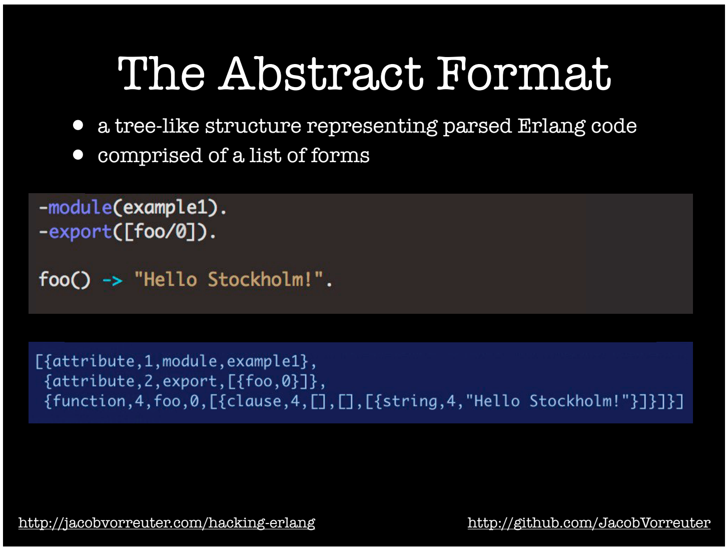
What are forms?



The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Forms are tuples that represent top-level constructs like function declarations and attributes

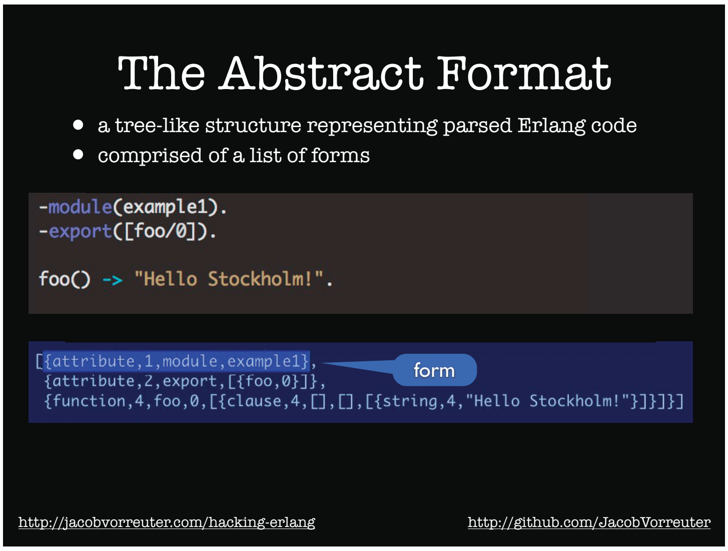


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

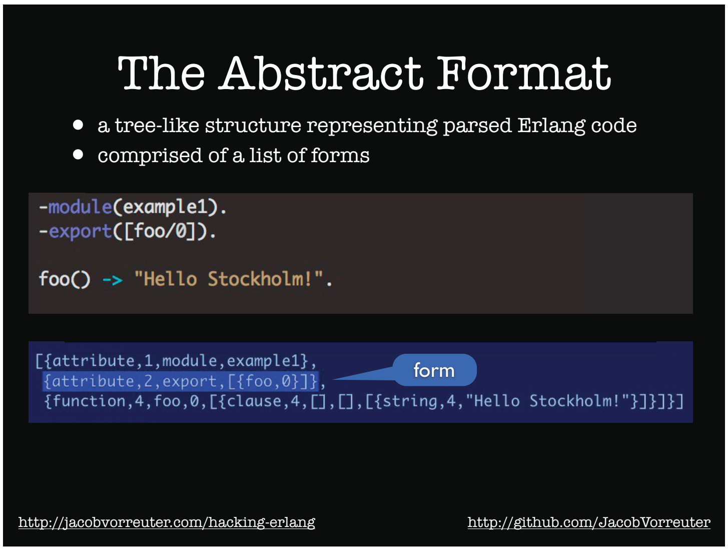


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

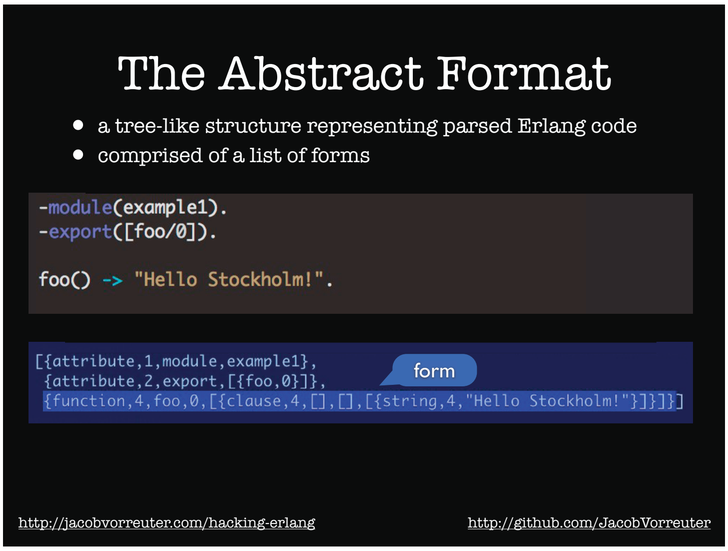


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

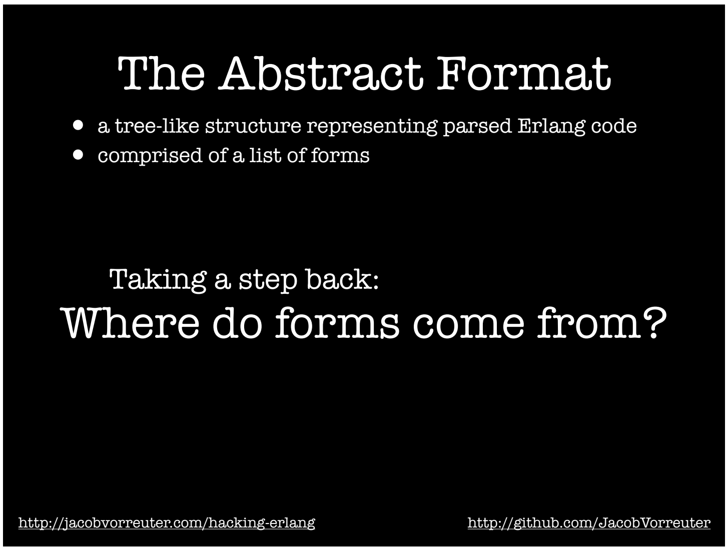


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```



The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Taking a step back:

Where do forms come from?

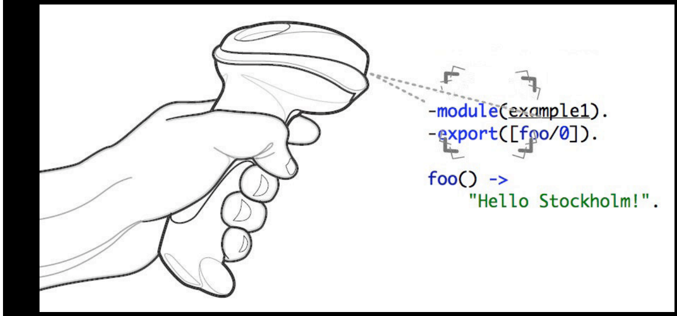
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Forms are generated by grouping and interpreting tokens scanned from source code.

Scanning Source Code

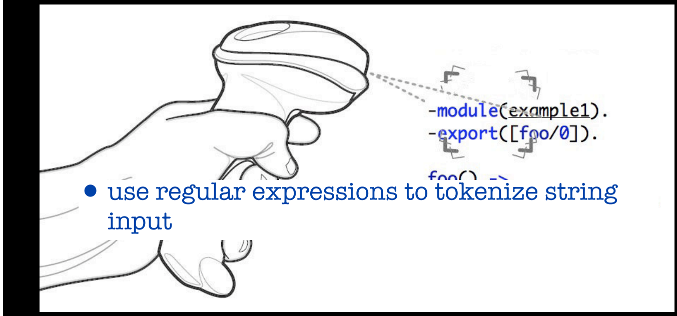
the first step in compiling



```
-module(example1).
-export([foo/0]).

foo() ->
    "Hello Stockholm!".
```

Scanning Source Code

the first step in compiling

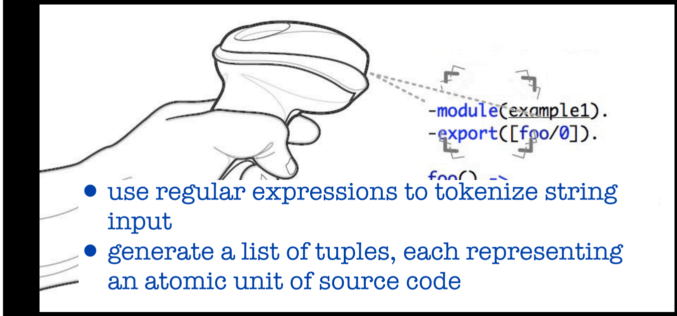


```
-module(example1).
-export([foo/0]).
```

- use regular expressions to tokenize string input

Scanning Source Code

the first step in compiling



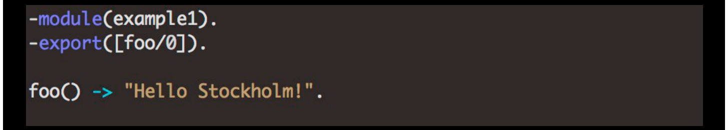
```
-module(example1).
-export([foo/0]).
```

- use regular expressions to tokenize string input
- generate a list of tuples, each representing an atomic unit of source code

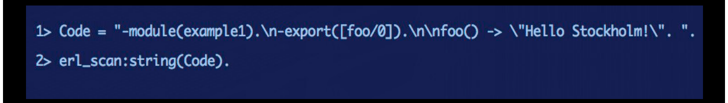
erl_scan

This module contains functions for tokenizing characters into Erlang tokens.

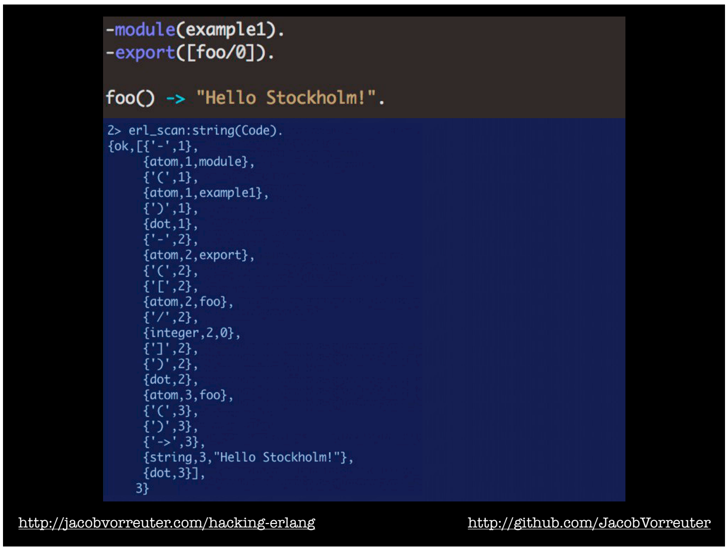
erl_scan



```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

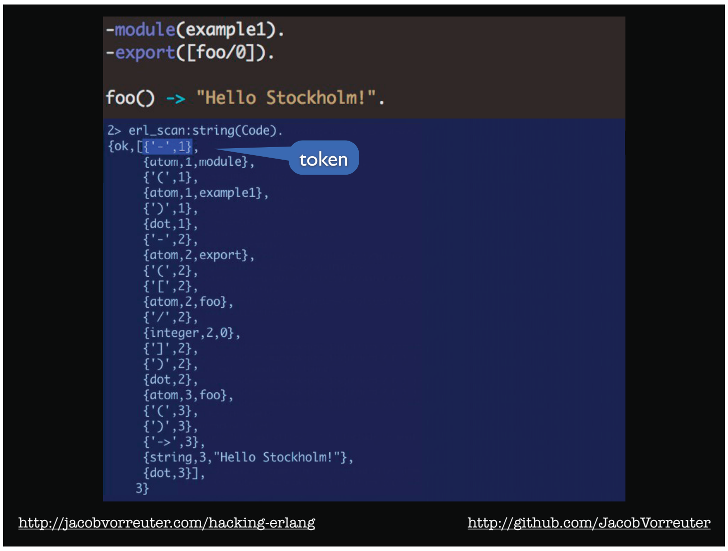


```
1> Code = "-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\". ".
2> erl_scan:string(Code).
```



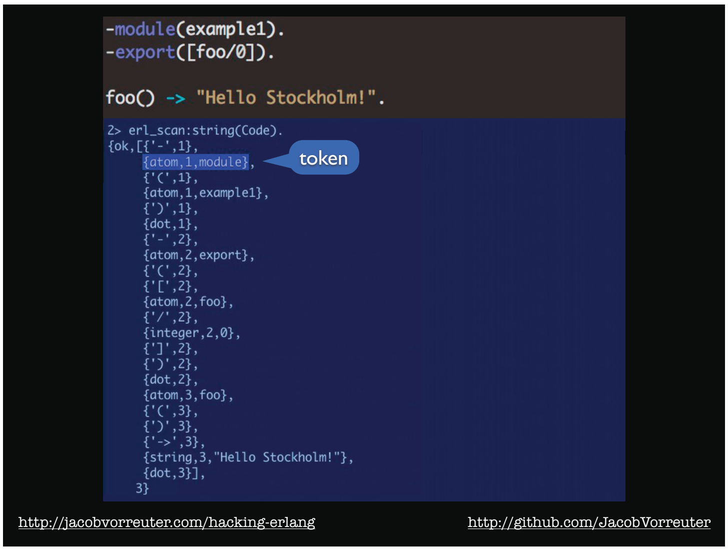
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
    3}
```



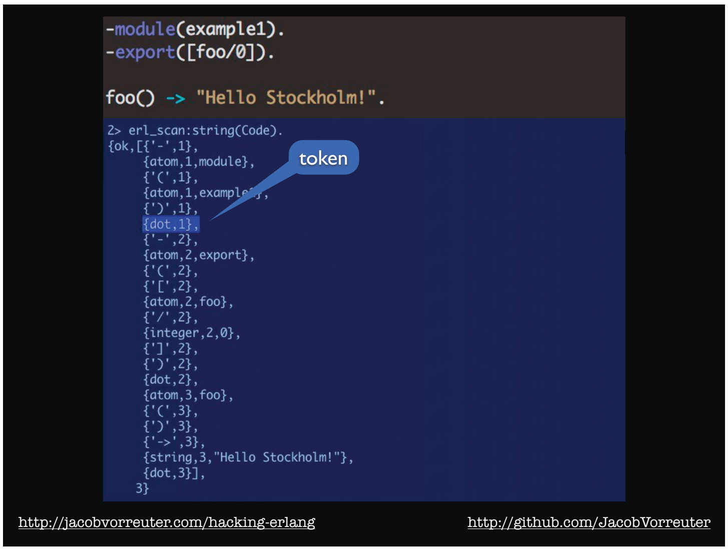
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
    3}
```



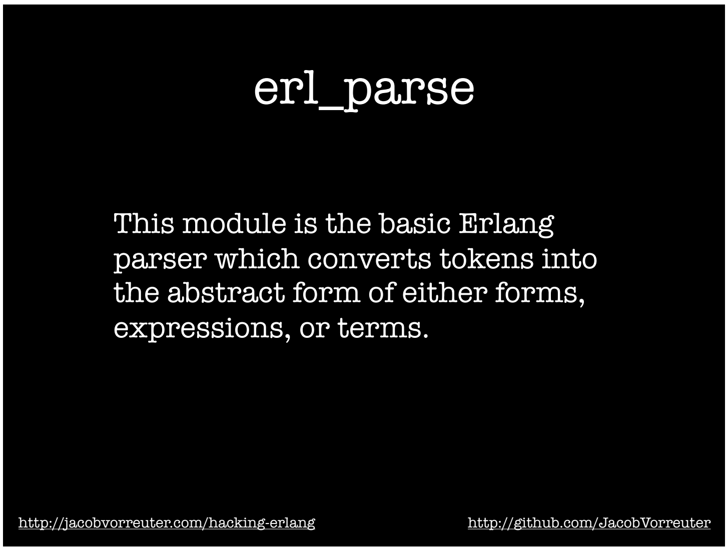
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
    3}
```



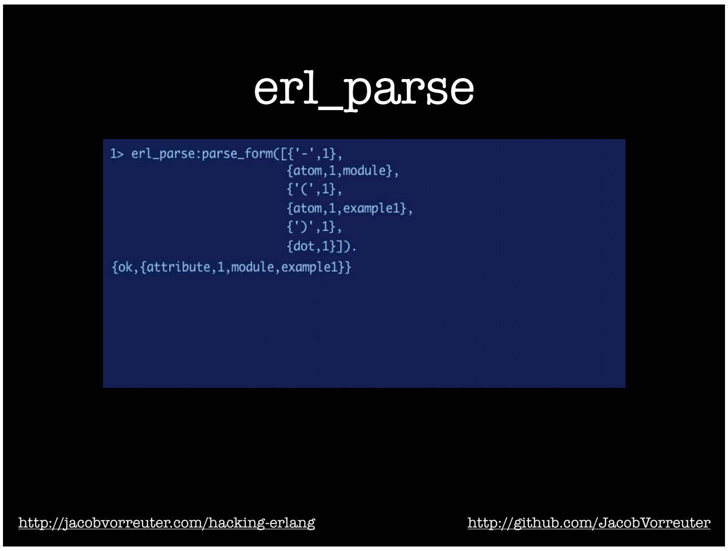
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
    3}
```



erl_parse

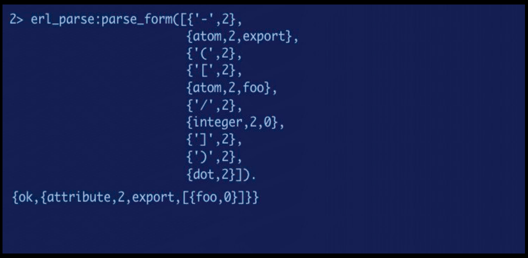
This module is the basic Erlang parser which converts tokens into the abstract form of either forms, expressions, or terms.



erl_parse

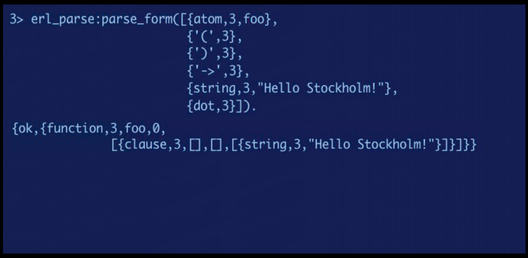
```
1> erl_parse:parse_form([{'-',1},
                         {atom,1,module},
                         {'(',1},
                         {atom,1,example1},
                         {')',1},
                         {dot,1}]).
{ok,{attribute,1,module,example1}}
```

erl_parse



```
2> erl_parse:parse_form([{'-',2},
                        {atom,2,export},
                        {'(',2},
                        {'[',2},
                        {atom,2,foo},
                        {'/',2},
                        {integer,2,0},
                        {']',2},
                        {')',2},
                        {dot,2}]).
{ok,{attribute,2,export,[{foo,0}]}}
```

erl_parse

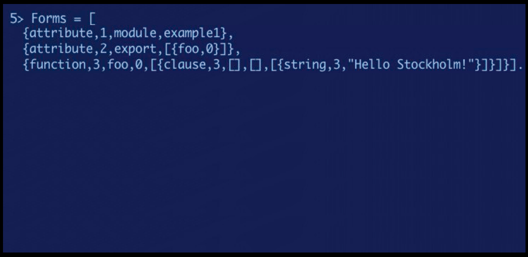


```
3> erl_parse:parse_form([{atom,3,foo},
                        {'(',3},
                        {')',3},
                        {'->',3},
                        {string,3,"Hello Stockholm!"},
                        {dot,3}]).
{ok,{function,3,foo,0,
              [{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}}
```

compile

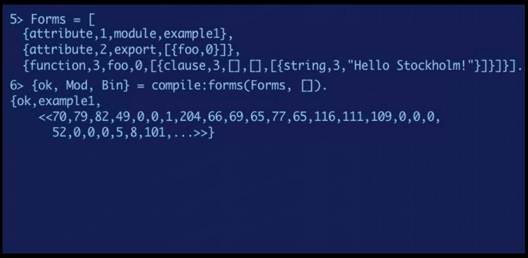
This module provides an interface to the standard Erlang compiler. It can generate either a new file which contains the object code, or return a binary which can be loaded directly.

compile



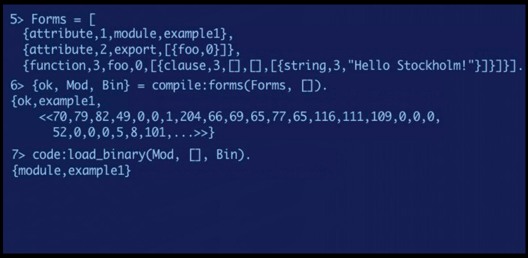
```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
```

compile



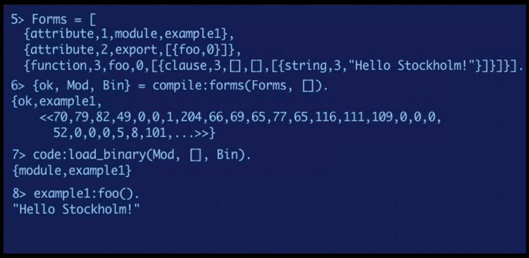
```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
```

compile



```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
7> code:load_binary(Mod, [], Bin).
{module,example1}
```

compile



```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
7> code:load_binary(Mod, [], Bin).
{module,example1}
8> example1:foo().
"Hello Stockholm!"
```

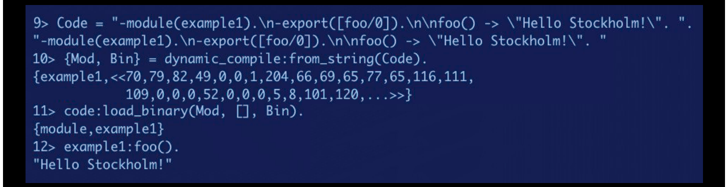
IS THERE A MODULE THAT CAN PERFORM ALL OF THOSE STEPS FOR ME?!?!?

dynamic_compile

The dynamic_compile module performs the actions we've just seen, plus takes care of macro expansion and inclusion of external header files.

http://github.com/JacobVorreuter/dynamic_compile

dynamic_compile

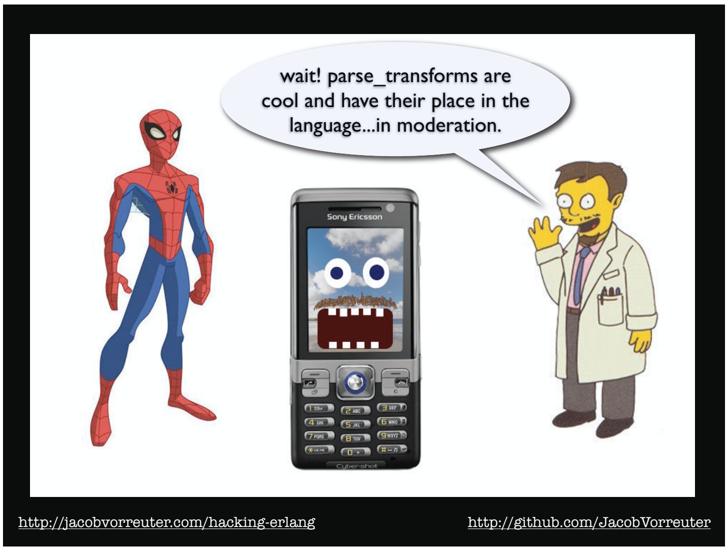


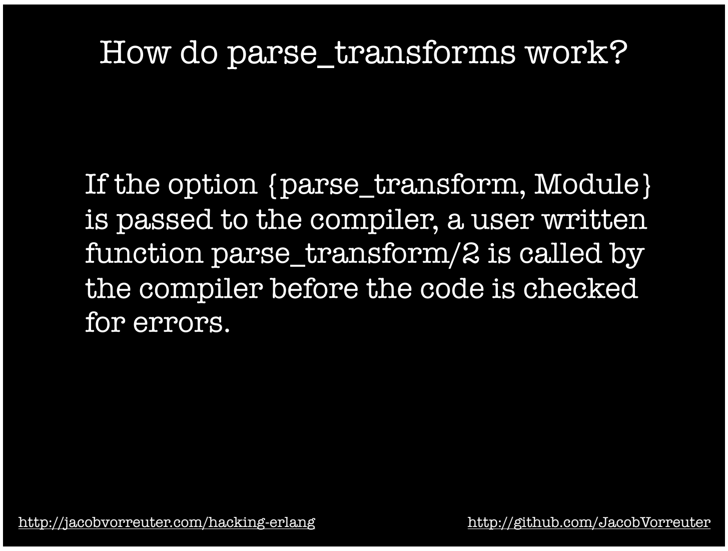
```
9> Code = "-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\". ".
"-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\". "
10> {Mod, Bin} = dynamic_compile:from_string(Code).
{example1,<<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,
            109,0,0,0,52,0,0,0,5,8,101,120,...>>}
11> code:load_binary(Mod, [], Bin).
{module,example1}
12> example1:foo().
"Hello Stockholm!"
```

moving on...

the parse_transform debate...

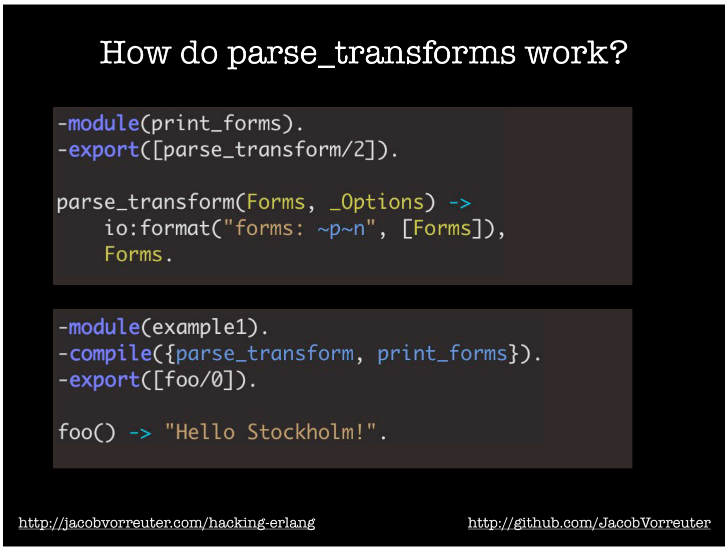
Programmers are strongly advised NOT to engage in parse transformations
Sony Ericsson
Cyber-shot


wait! parse_transforms are cool and have their place in the language...in moderation.
Sony Ericsson
Cyber-shot



How do parse_transforms work?

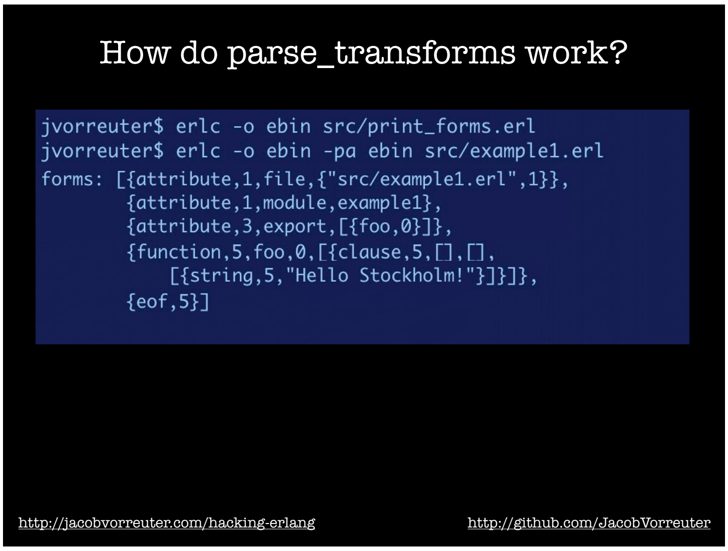
If the option {parse_transform, Module} is passed to the compiler, a user written function parse_transform/2 is called by the compiler before the code is checked for errors.



How do parse_transforms work?

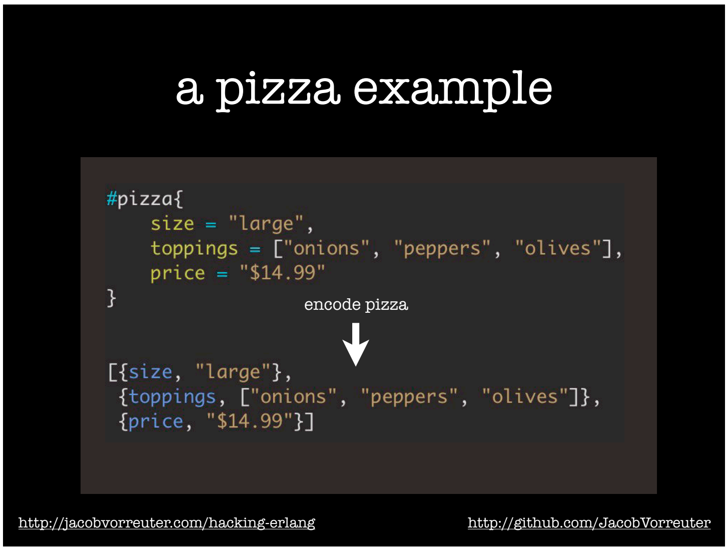
```
-module(print_forms).
-export([parse_transform/2]).

parse_transform(Forms, _Options) ->
    io:format("forms: ~p~n", [Forms]),
    Forms.
```

```
-module(example1).
-compile({parse_transform, print_forms}).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```



How do parse_transforms work?

```
jvorreuter$ erlc -o ebin src/print_forms.erl
jvorreuter$ erlc -o ebin -pa ebin src/example1.erl
forms: [{attribute,1,file,{"src/example1.erl",1}},
        {attribute,1,module,example1},
        {attribute,3,export,[{foo,0}]},
        {function,5,foo,0,[{clause,5,[],[],
            [{string,5,"Hello Stockholm!"}]}]},
        {eof,5}]
```



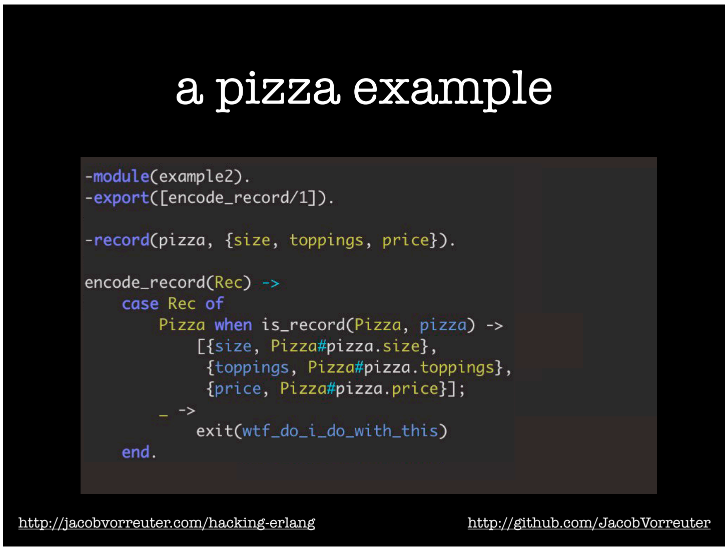
a pizza example

```
#pizza{
    size = "large",
    toppings = ["onions", "peppers", "olives"],
    price = "$14.99"
}
```

encode pizza

```
[{size, "large"},
 {toppings, ["onions", "peppers", "olives"]},
 {price, "$14.99"}]
```

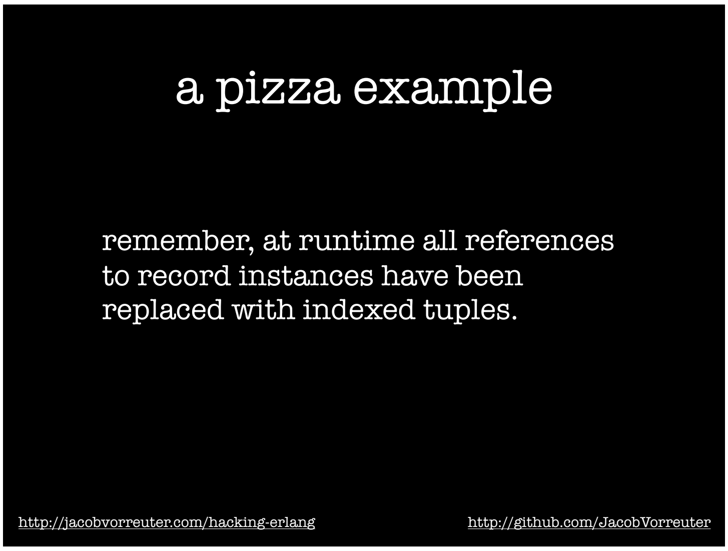
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

```
-module(example2).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    case Rec of
        Pizza when is_record(Pizza, pizza) ->
            [{size, Pizza#pizza.size},
             {toppings, Pizza#pizza.toppings},
             {price, Pizza#pizza.price}];
        _ ->
            exit(wtf_do_i_do_with_this)
    end.
```

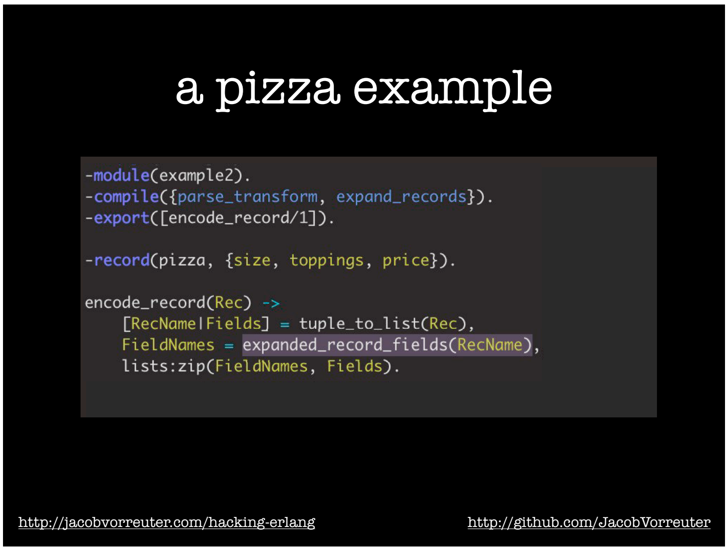
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

remember, at runtime all references to record instances have been replaced with indexed tuples.

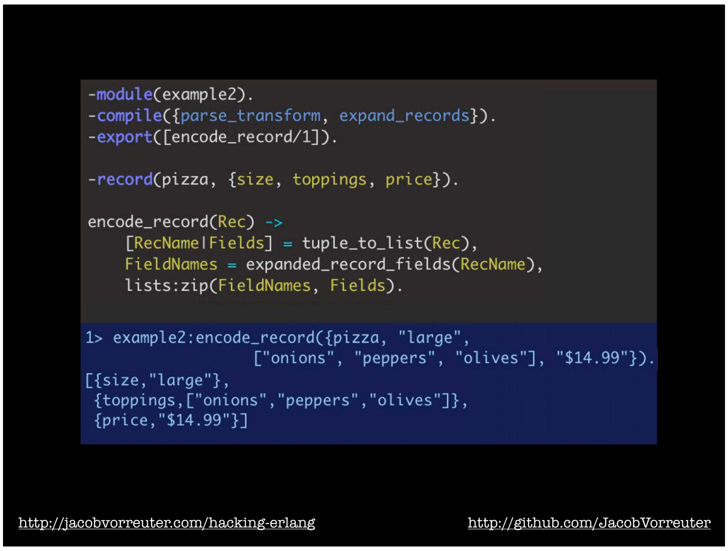
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

```
-module(example2).
-compile({parse_transform, expand_records}).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    [RecName|Fields] = tuple_to_list(Rec),
    FieldNames = expanded_record_fields(RecName),
    lists:zip(FieldNames, Fields).
```

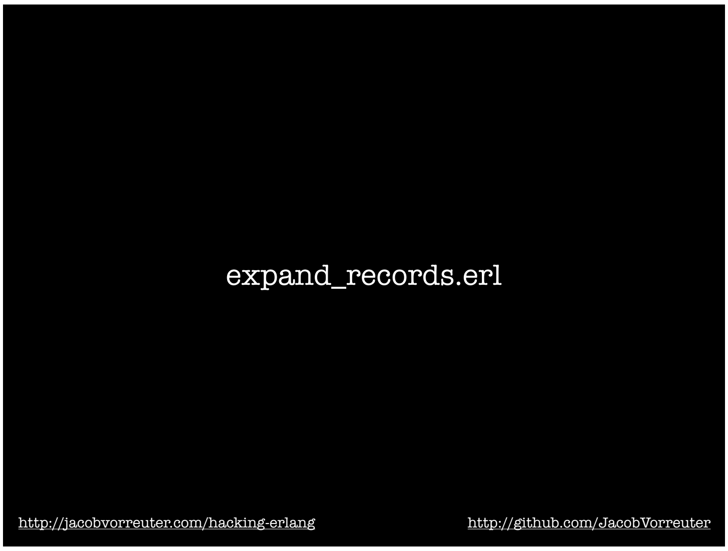
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



```
-module(example2).
-compile({parse_transform, expand_records}).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    [RecName|Fields] = tuple_to_list(Rec),
    FieldNames = expanded_record_fields(RecName),
    lists:zip(FieldNames, Fields).
```

```
1> example2:encode_record({pizza, "large",
                   ["onions", "peppers", "olives"], "$14.99"}).
[{size,"large"},
 {toppings,["onions","peppers","olives"]},
 {price,"$14.99"}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



expand_records.erl

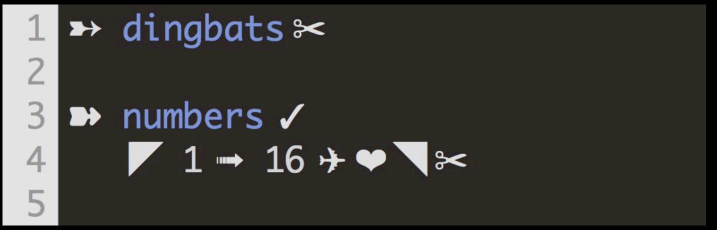
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

intermission

Act II

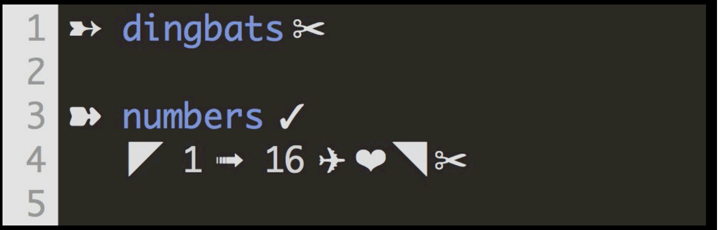
compiling custom syntax

Compiling Custom Syntax

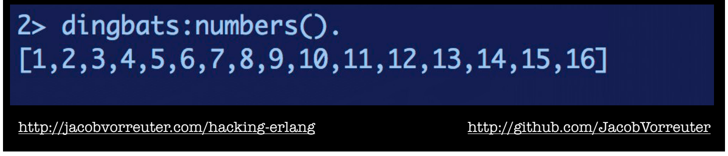


```
➻ dingbats ✂

➽ numbers ✓
  ◤ 1 ➡ 16 ✈ ❤ ◥ ✂
```

Compiling Custom Syntax



```
➻ dingbats ✂

➽ numbers ✓
  ◤ 1 ➡ 16 ✈ ❤ ◥ ✂
```



```
2> dingbats:numbers().
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

Compiling Custom Syntax

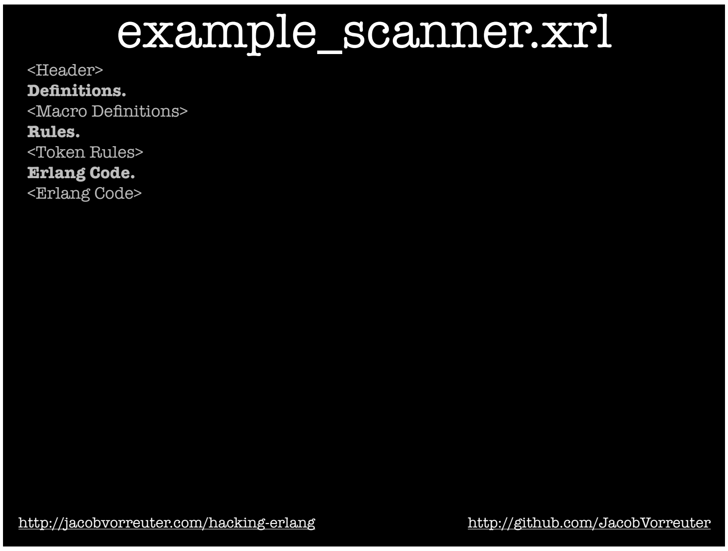
leex - A regular expression based lexical analyzer generator for Erlang, similar to lex or flex.

yecc - An LALR-1 parser generator for Erlang, similar to yacc.

leex

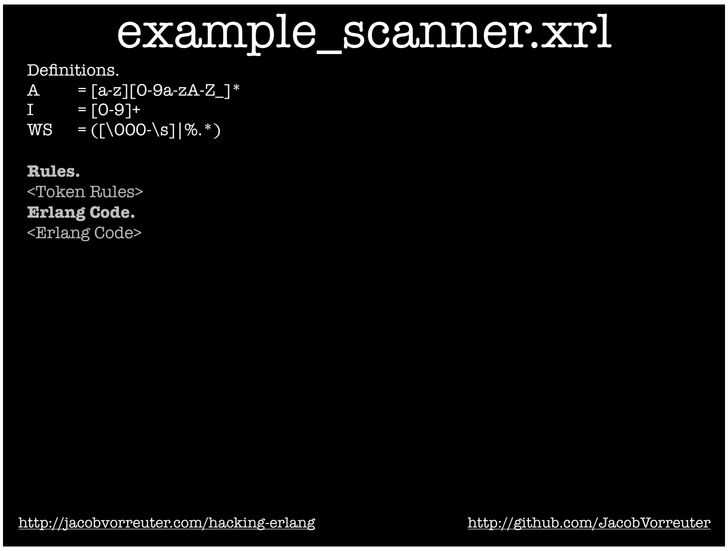
The leex module takes a definition file with the extension .xrl as input and generates the source code for a lexical analyzer as output.

<Header>
Definitions.
<Macro Definitions>
Rules.
<Token Rules>
Erlang Code.
<Erlang Code>



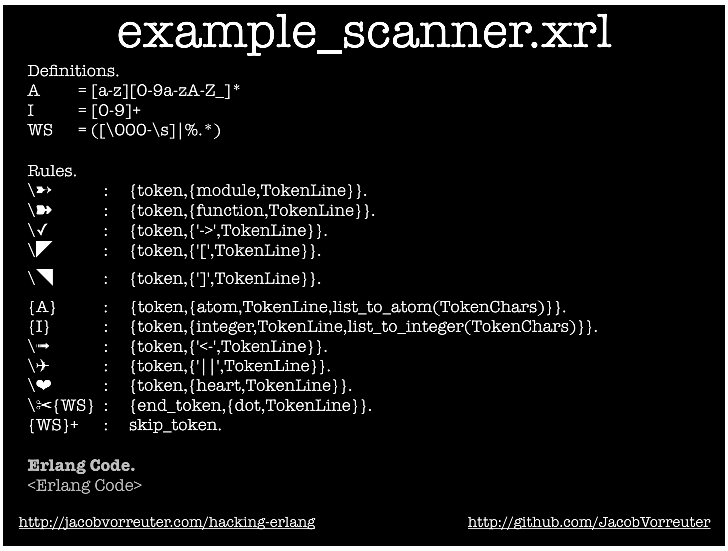
example_scanner.xrl

```
<Header>
Definitions.
<Macro Definitions>
Rules.
<Token Rules>
Erlang Code.
<Erlang Code>
```



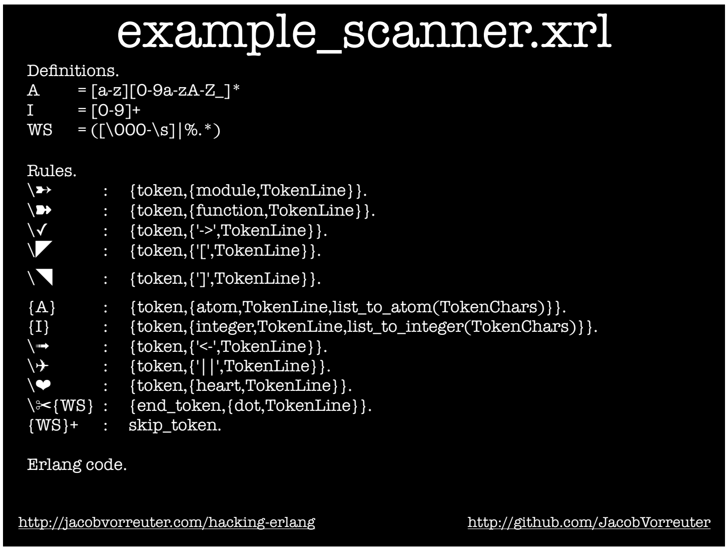
example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
<Token Rules>
Erlang Code.
<Erlang Code>
```



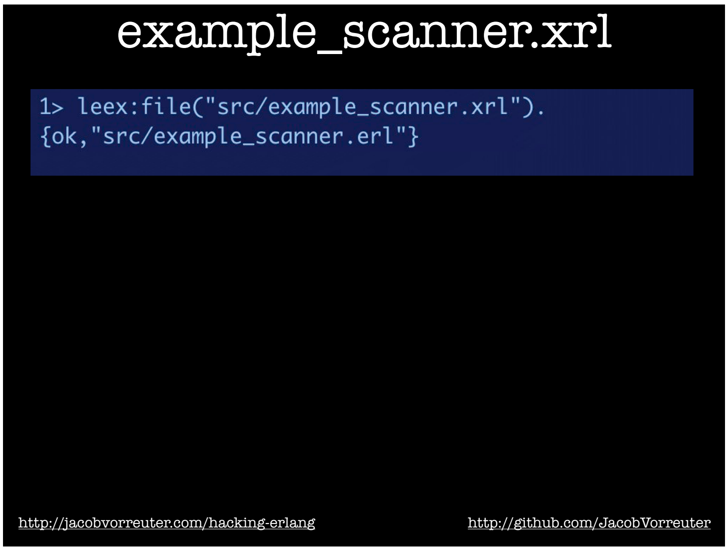
example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
\➳         :    {token,{module,TokenLine}}.
\➽         :    {token,{function,TokenLine}}.
\✓         :    {token,{'->',TokenLine}}.
\◤         :    {token,{'[',TokenLine}}.
\◥         :    {token,{']',TokenLine}}.
{A}        :    {token,{atom,TokenLine,list_to_atom(TokenChars)}}.
{I}        :    {token,{integer,TokenLine,list_to_integer(TokenChars)}}.
\➠         :    {token,{'<-',TokenLine}}.
\✈         :    {token,{'||',TokenLine}}.
\❤         :    {token,{heart,TokenLine}}.
\✄{WS}     :    {end_token,{dot,TokenLine}}.
{WS}+      :    skip_token.

Erlang Code.
<Erlang Code>
```



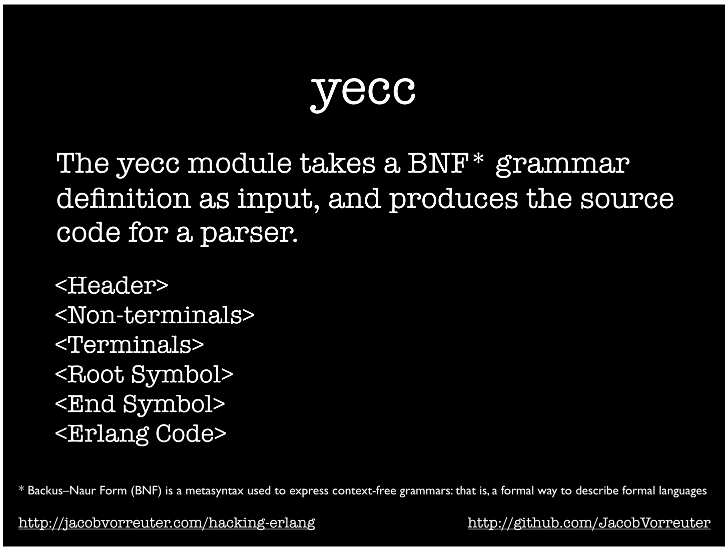
example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
\➳         :    {token,{module,TokenLine}}.
\➽         :    {token,{function,TokenLine}}.
\✓         :    {token,{'->',TokenLine}}.
\◤         :    {token,{'[',TokenLine}}.
\◥         :    {token,{']',TokenLine}}.
{A}        :    {token,{atom,TokenLine,list_to_atom(TokenChars)}}.
{I}        :    {token,{integer,TokenLine,list_to_integer(TokenChars)}}.
\➠         :    {token,{'<-',TokenLine}}.
\✈         :    {token,{'||',TokenLine}}.
\❤         :    {token,{heart,TokenLine}}.
\✄{WS}     :    {end_token,{dot,TokenLine}}.
{WS}+      :    skip_token.

Erlang code.
```



example_scanner.xrl

```
1> leex:file("src/example_scanner.xrl").
{ok,"src/example_scanner.erl"}
```

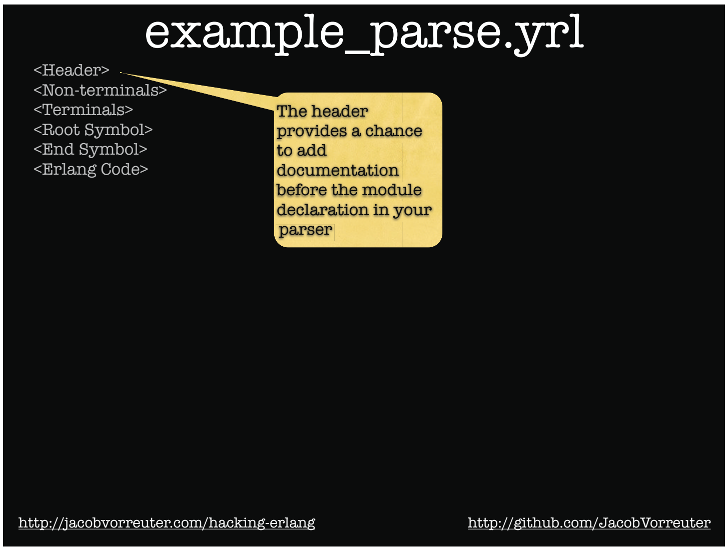


yecc

The yecc module takes a BNF* grammar definition as input, and produces the source code for a parser.

```
<Header>
<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>
```

* Backus–Naur Form (BNF) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages

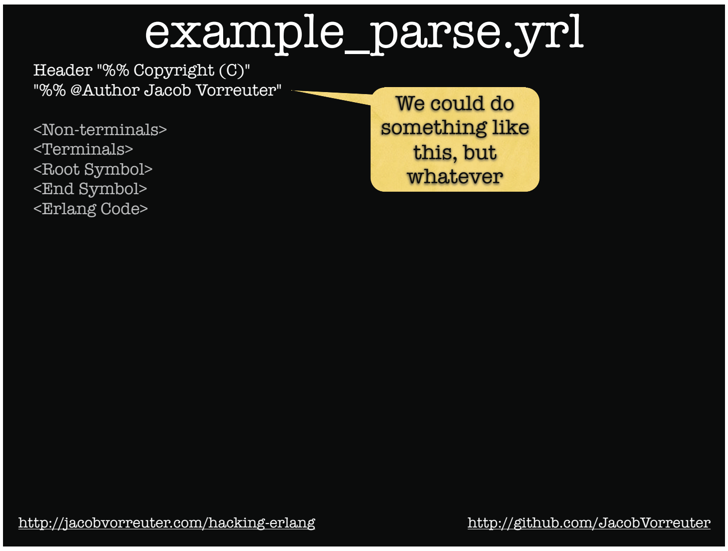


example_parse.yrl

<Header>
<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

The header provides a chance to add documentation before the module declaration in your parser

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

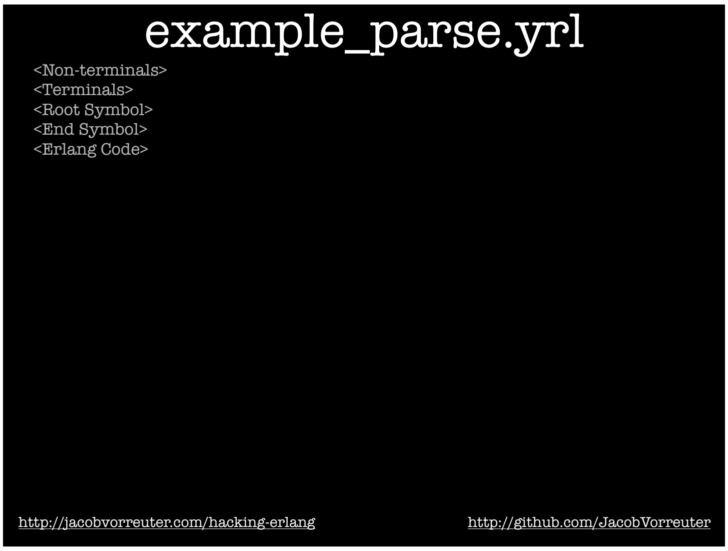


example_parse.yrl

```
Header "%% Copyright (C)"
"%% @Author Jacob Vorreuter"

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>
```

We could do something like this, but whatever

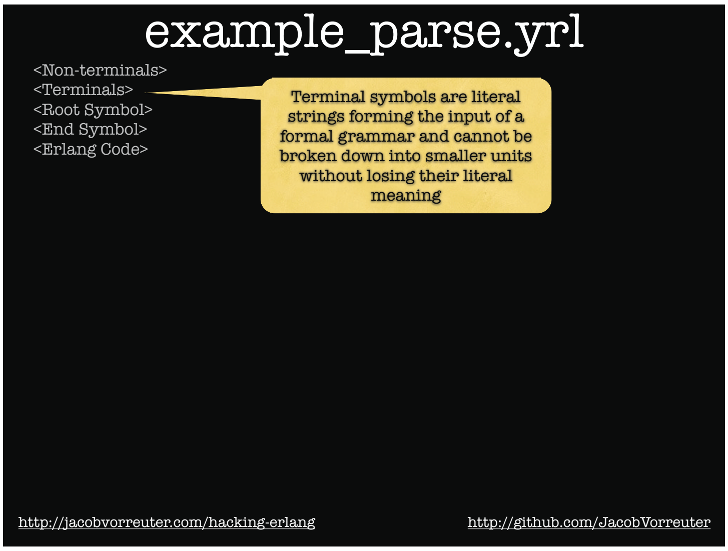
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

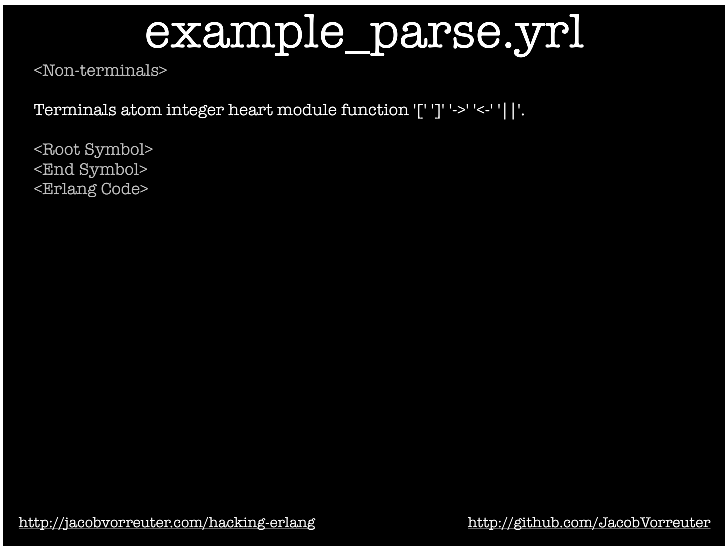


example_parse.yrl

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

Terminal symbols are literal strings forming the input of a formal grammar and cannot be broken down into smaller units without losing their literal meaning

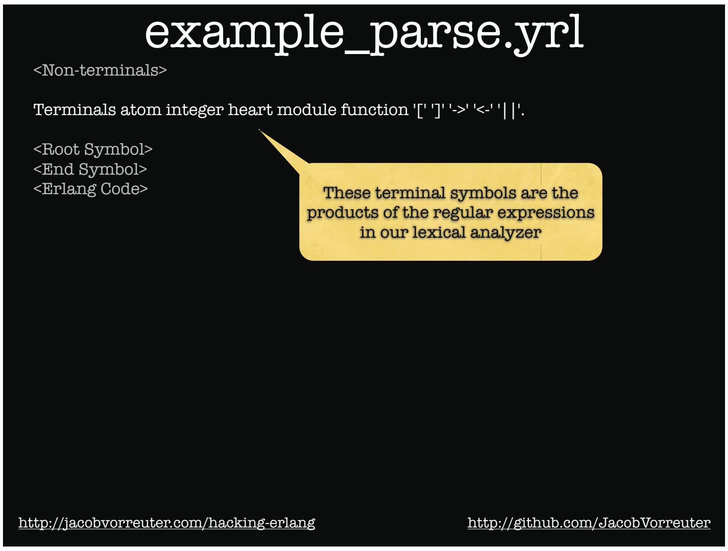
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

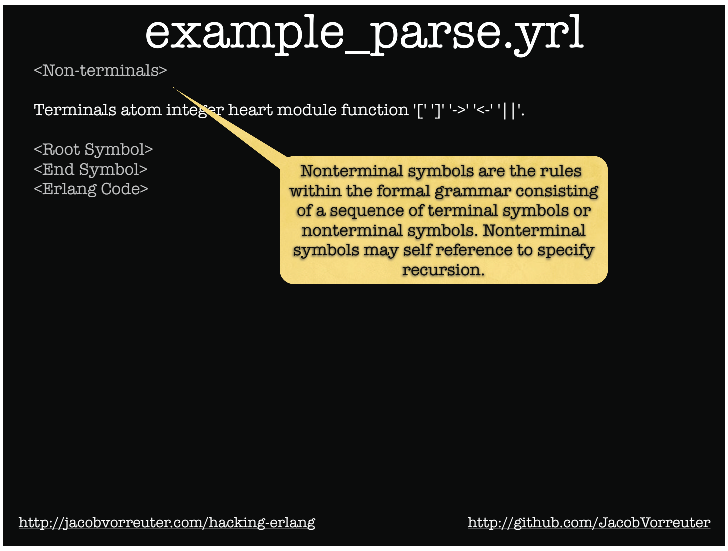


example_parse.yrl

```
<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>
```

These terminal symbols are the products of the regular expressions in our lexical analyzer

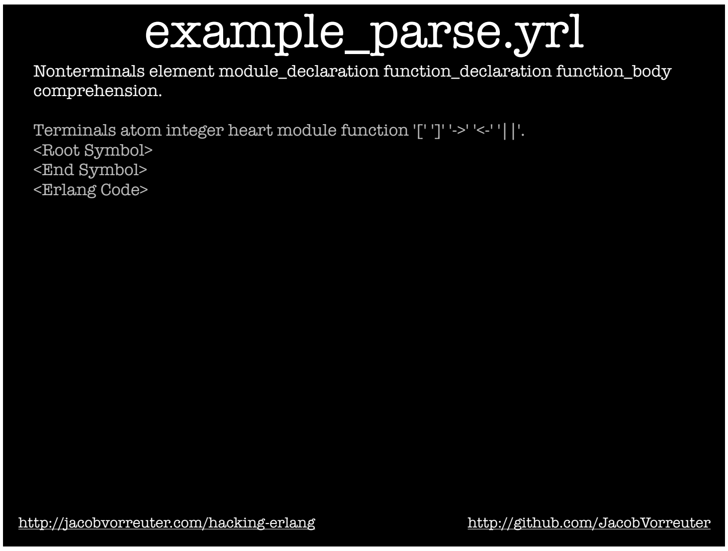
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

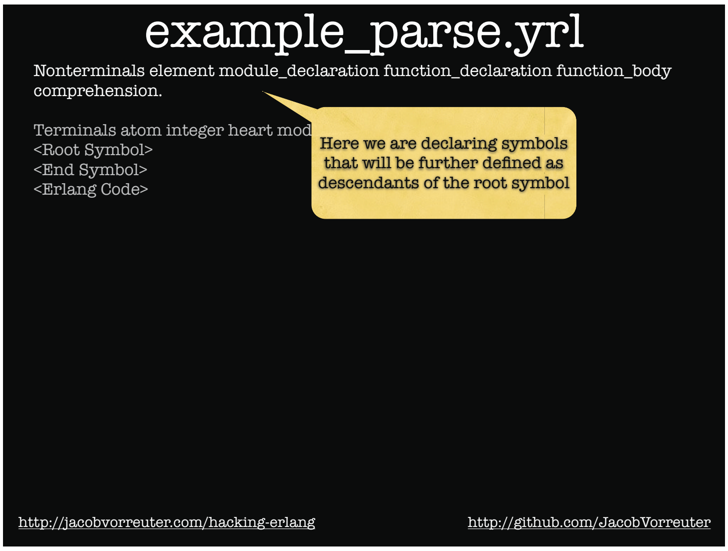
```
<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>
```

Nonterminal symbols are the rules within the formal grammar consisting of a sequence of terminal symbols or nonterminal symbols. Nonterminal symbols may self reference to specify recursion.



example_parse.yrl

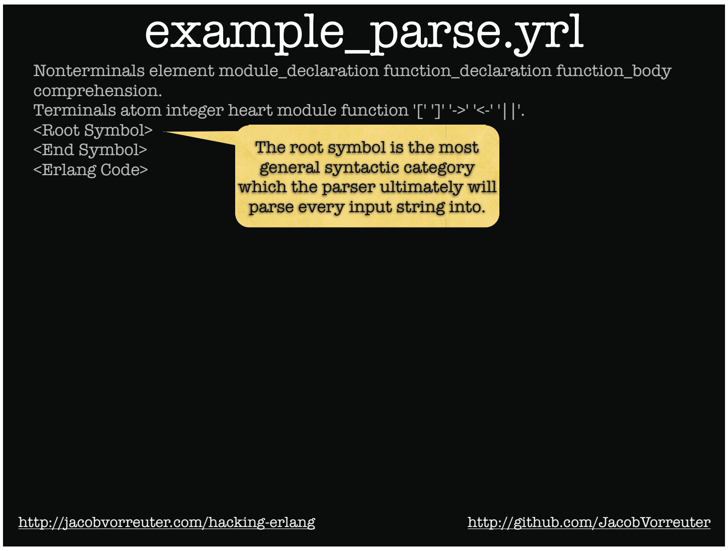
```
Nonterminals element module_declaration function_declaration function_body
comprehension.

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
<Root Symbol>
<End Symbol>
<Erlang Code>
```



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.

Terminals atom integer heart mod
<Root Symbol>
<End Symbol>
<Erlang Code>
```

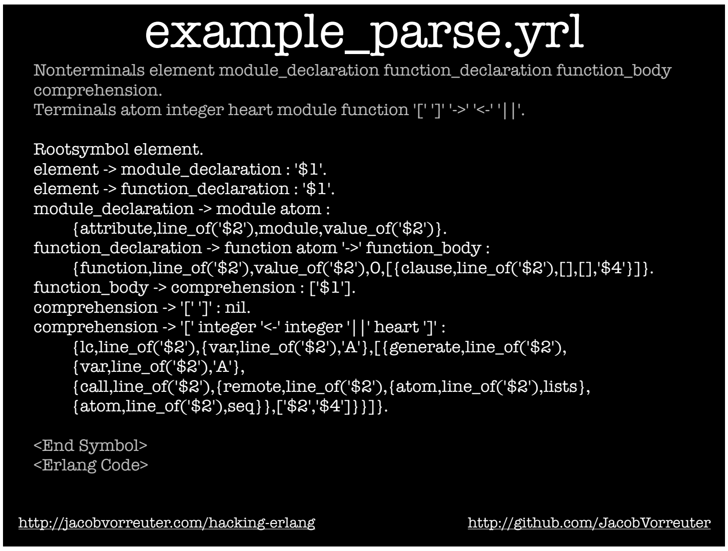
Here we are declaring symbols that will be further defined as descendants of the root symbol



example_parse.yrl

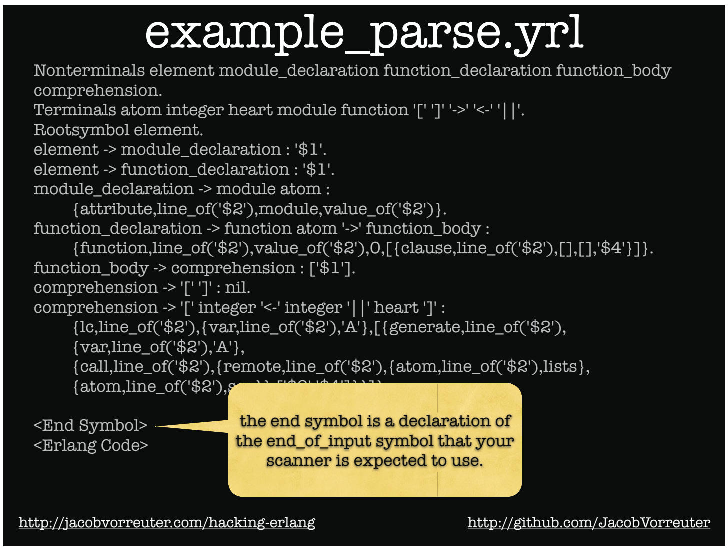
```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
<Root Symbol>
<End Symbol>
<Erlang Code>
```

The root symbol is the most general syntactic category which the parser ultimately will parse every input string into.



example_parse.yrl

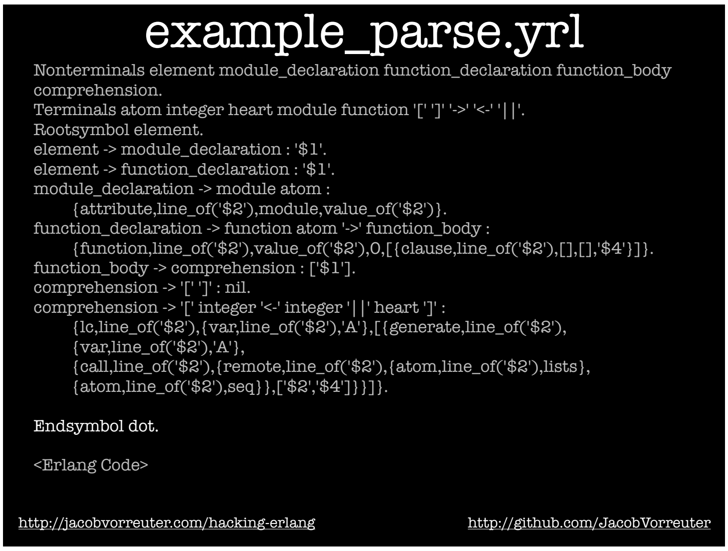
```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.

<End Symbol>
<Erlang Code>
```



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),s

<End Symbol>
<Erlang Code>
```

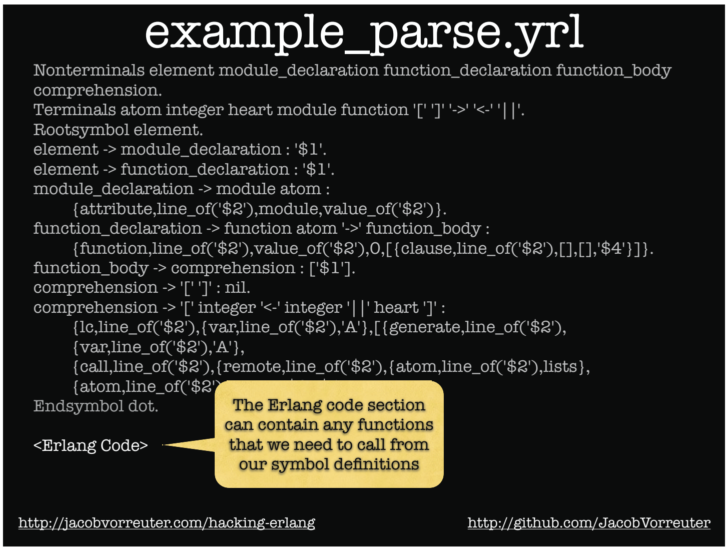
the end symbol is a declaration of the end_of_input symbol that your scanner is expected to use.



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.

Endsymbol dot.

<Erlang Code>
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

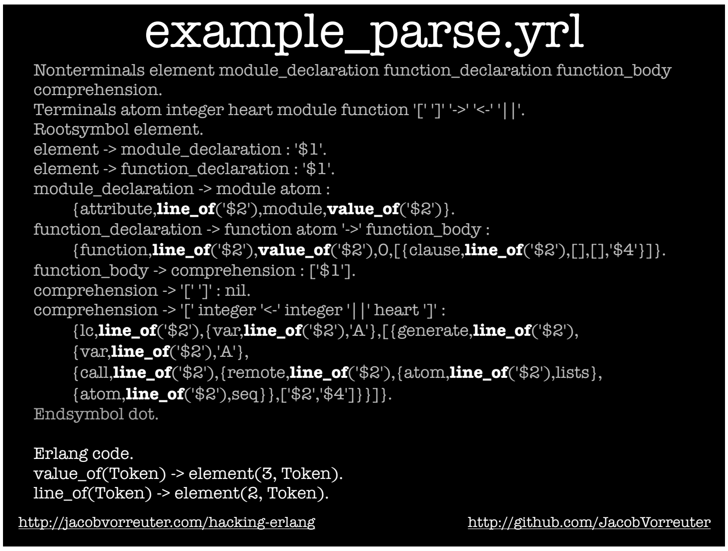


example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'
Endsymbol dot.

<Erlang Code>
```

The Erlang code section can contain any functions that we need to call from our symbol definitions

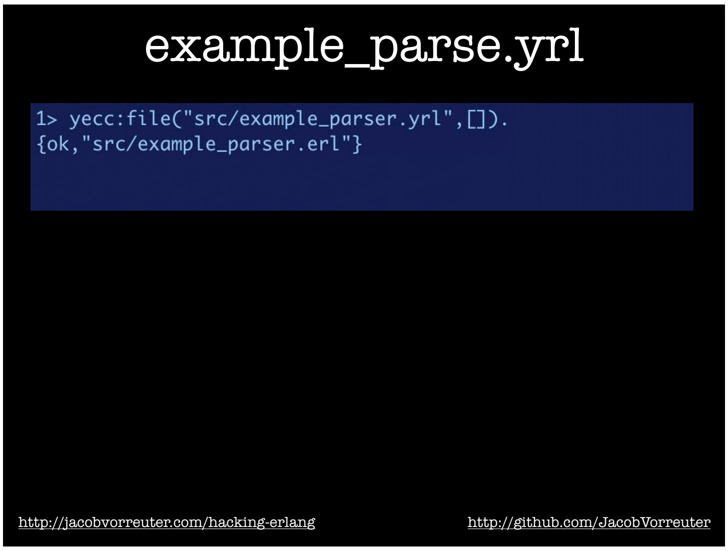
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.
Endsymbol dot.

Erlang code.
value_of(Token) -> element(3, Token).
line_of(Token) -> element(2, Token).
```

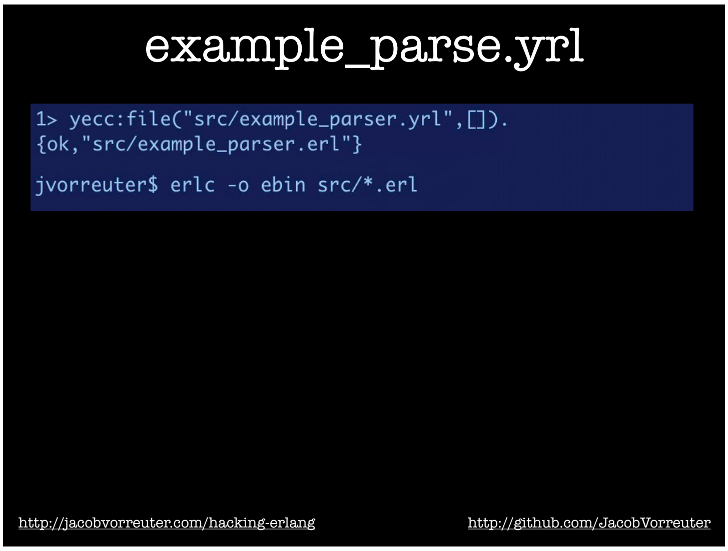
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> yecc:file("src/example_parser.yrl",[]).
{ok,"src/example_parser.erl"}
```

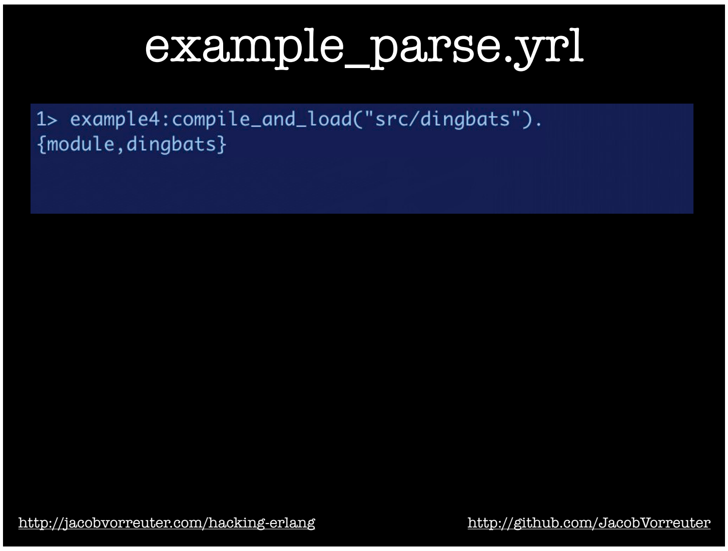
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> yecc:file("src/example_parser.yrl",[]).
{ok,"src/example_parser.erl"}

jvorreuter$ erlc -o ebin src/*.erl
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> example4:compile_and_load("src/dingbats").
{module,dingbats}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

example_parse.yrl

```
1> example4:compile_and_load("src/dingbats").
{module,dingbats}
2> dingbats:numbers().
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

example4.erl

```
-module(example4).
-export([compile_and_load/1]).

compile_and_load(Path) ->
    {ok, Bin} = file:read_file(Path),
    [Form|Forms] = scan_parse([], binary_to_list(Bin), 0, []),
    Forms1  = [Form,{attribute,1,compile,export_all}|Forms],
    {ok, Mod, Bin1} = compile:forms(Forms1, []),
    code:load_binary(Mod, [], Bin1).

scan_parse(Cont, Str, StartLoc, Acc) ->
    case example_scanner:tokens(Cont, Str, StartLoc) of
        {done, {ok, Tokens, EndLoc}, LeftOverChars} ->
            {ok, Form} = example_parser:parse(Tokens),
            scan_parse([], LeftOverChars, EndLoc, [Form|Acc]);
        _ ->
            lists:reverse(Acc)
    end.
```

custom syntax in the wild...

- Lisp Flavored Erlang
- Prolog Interpreter for Erlang
- Erlang implementation of the Django Template Language

END