Create Your Own Language

How to implement a language on top of Erlang Virtual Machine (BEAM)

Hamidreza Soleimani
Backend Developer / Architect @ BisPhone

Tehran Linux User Group
August 6, 2015

SQL Java Haskell PHP

Lisp C Prolog SQL C# Objectiv-C

Why should we create a new language?

Elixir Rust

Scala Ruby Go

Erlang XML Perl

C++ Python SQL Javascript

There are lots of Languages!

- By category:
  Programming, Query, Domain Specific, etc
- By Paradigm:
  Imperative, Declarative (Logic, Functional), Structured (Object Oriented, Modular), etc
- By Implementation:
  Compiled, Interpreted, Mixed, etc
- By Type System:
  Static, Dynamic, Strong, Weak, etc

So why should we create it?

- Reason 1: Implementing a new idea
- Reason 2: Solving a new problem
- Reason 3: Mastering language concepts
- Reason 4: Just for fun!

How can we create a new language?

1. Designing

1.1. Identifying the problem

1.2. Planning the target platform/machine

1.3. Determining language category, paradigm, types, etc

2. Implementing Frontend

2.1. Lexical Scanning (Token Generating)
(example: lex, flex, leex, etc)

2.2. Syntax & Semantics Parsing (AST Generating)
(example: yacc, bison, yecc)

2.3. Preprocessing

2.4. Lint Analyzing

2.5. Generating Intermediate Language
(example: GCC IR, LLVM IR, BEAM Bytecode, Java Bytecode)

3. Implementing Backend

3.1. Analyzing Intermediate Language

3.2. Optimizing

3.3. Generating Machine Code

3.4. Evaluating and Executing

Lets look inside Erlang compiler

What is Erlang?

"Erlang is a general-purpose, functional, concurrent, garbage-collected programming language and runtime system, with eager evaluation, single assignment, and dynamic typing. It was originally designed by Ericsson to support distributed, fault-tolerant, soft real-time, highly available, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system."

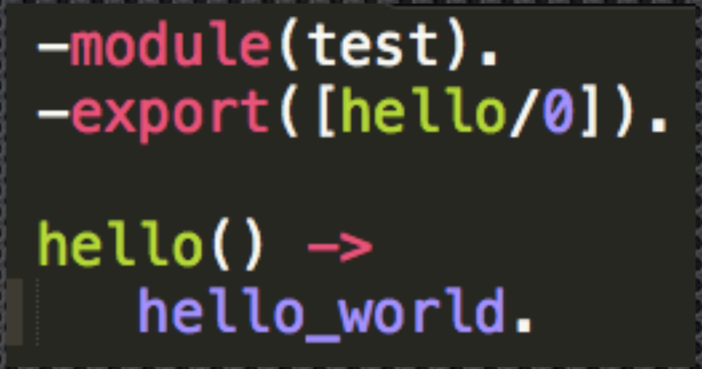
– Holy Wikipedia

Code Generation Steps

- Erlang Source Code
- Parse Transformed & Preprocessed Code (erlc -P)
- Source Transformed Code (erlc -E)
- Abstract Syntax Tree (erlc +dabstr)
- Expanded Abstract Syntax Tree (erlc +dexp)
- Core Erlang (erlc +to_core)
- Assembler Code (erlc -S)
- BEAM Bytecode (erlc)

Source Code

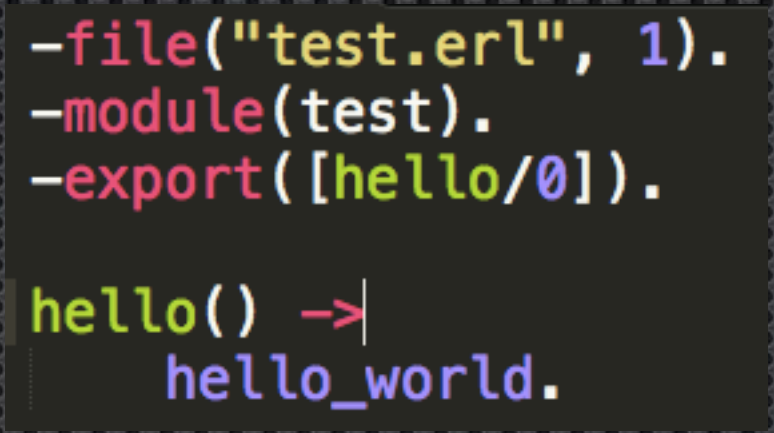
$ vim test.erl



```
-module(test).
-export([hello/0]).

hello() ->
    hello_world.
```

Parse Transformed Code

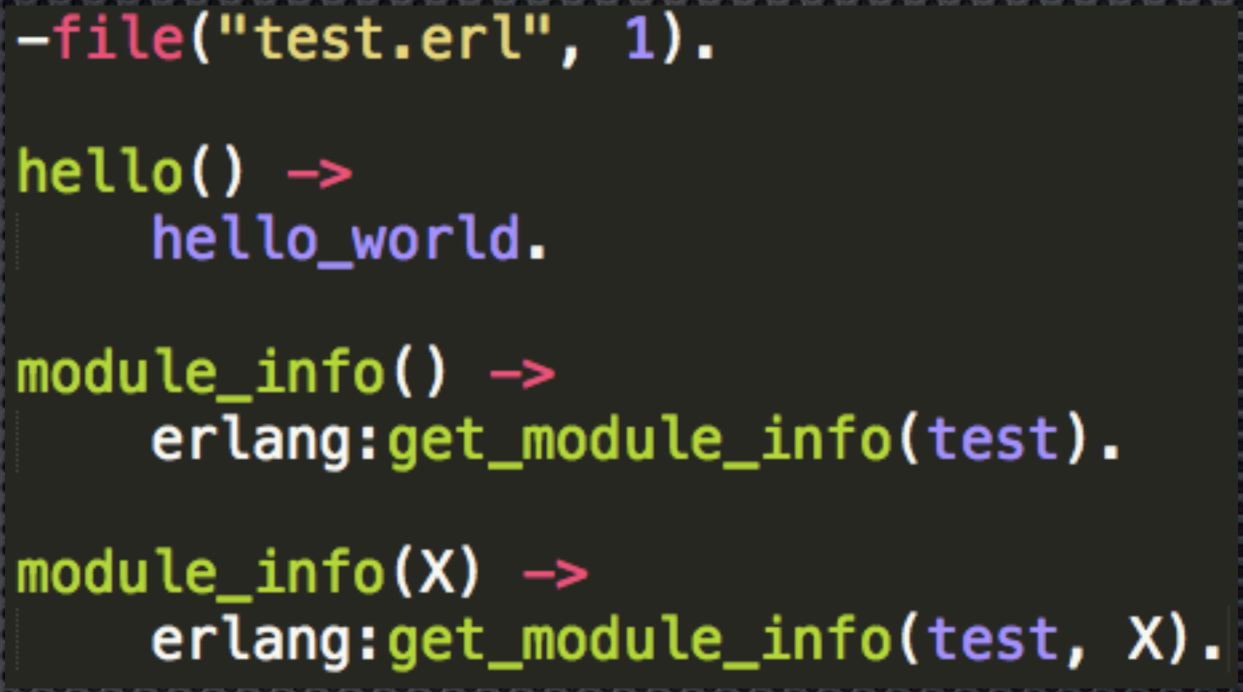
$ erlc -P test.erl



```erlang
-file("test.erl", 1).
-module(test).
-export([hello/0]).

hello() ->
    hello_world.
```

Source Transformed Code

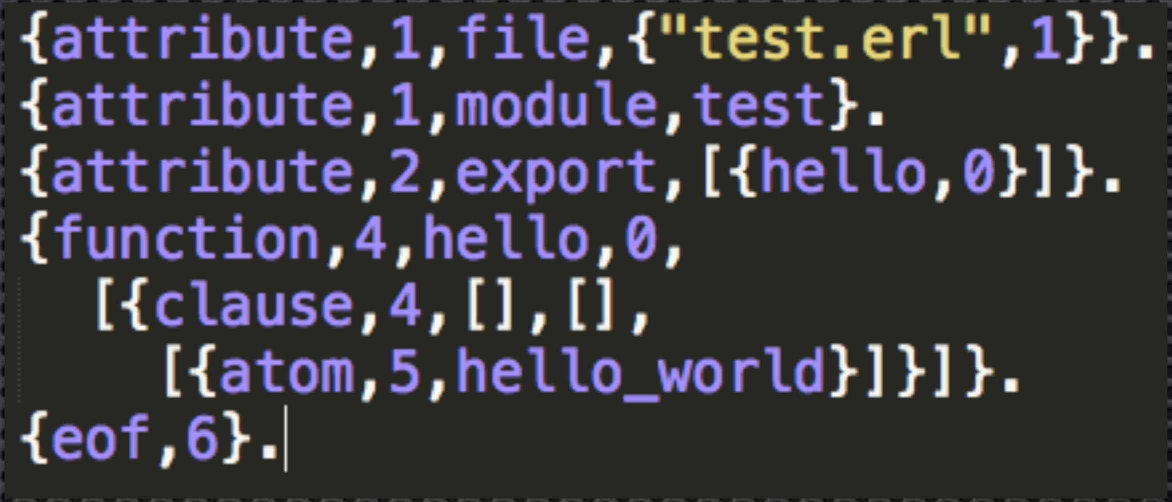
$ erlc -E test.erl



```
-file("test.erl", 1).

hello() ->
    hello_world.

module_info() ->
    erlang:get_module_info(test).

module_info(X) ->
    erlang:get_module_info(test, X).
```

AST Code

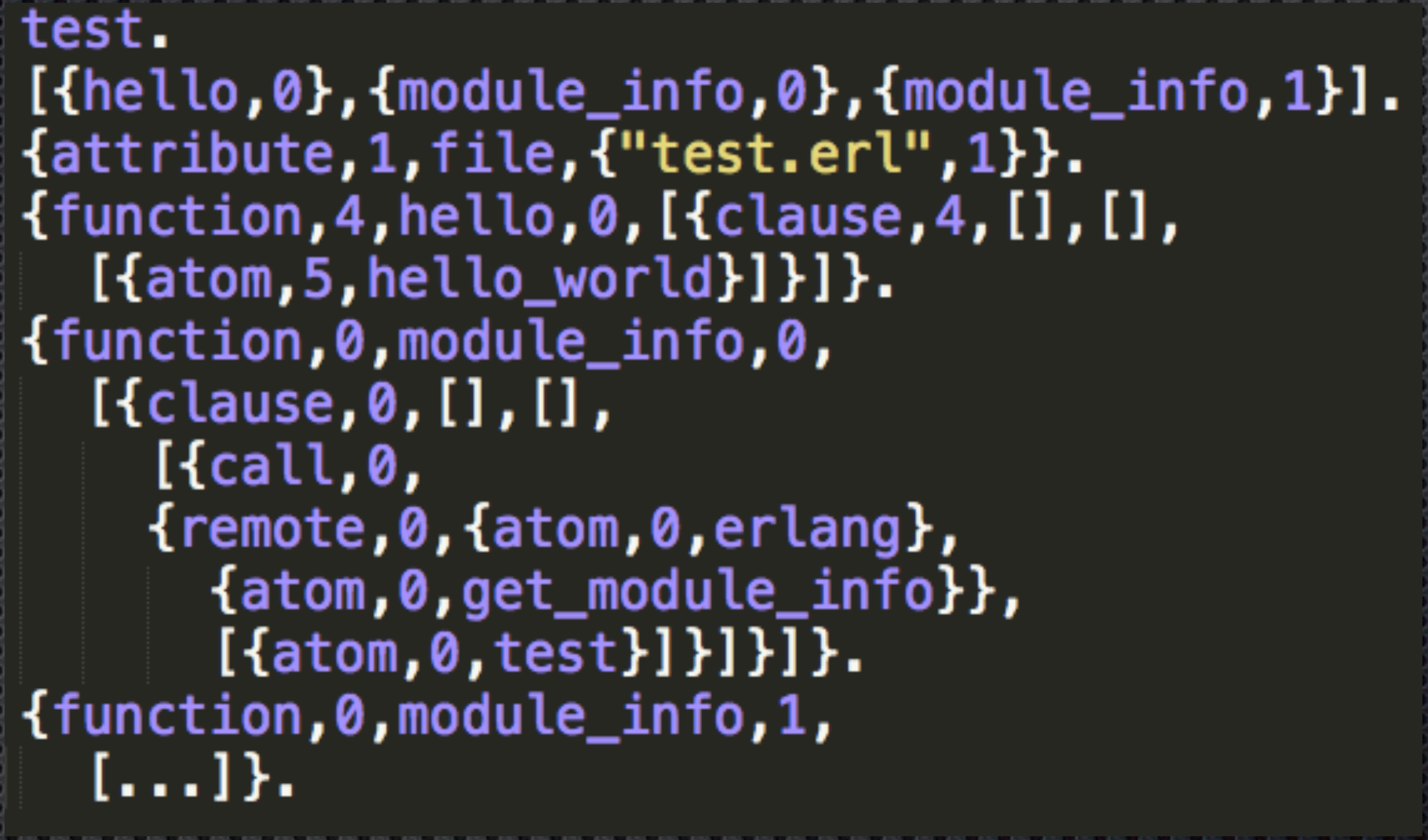
$ erlc +dabstr test.erl



```
{attribute,1,file,{"test.erl",1}}.
{attribute,1,module,test}.
{attribute,2,export,[{hello,0}]}.
{function,4,hello,0,
  [{clause,4,[],[],
    [{atom,5,hello_world}]}]}.
{eof,6}.
```

Expanded AST Code

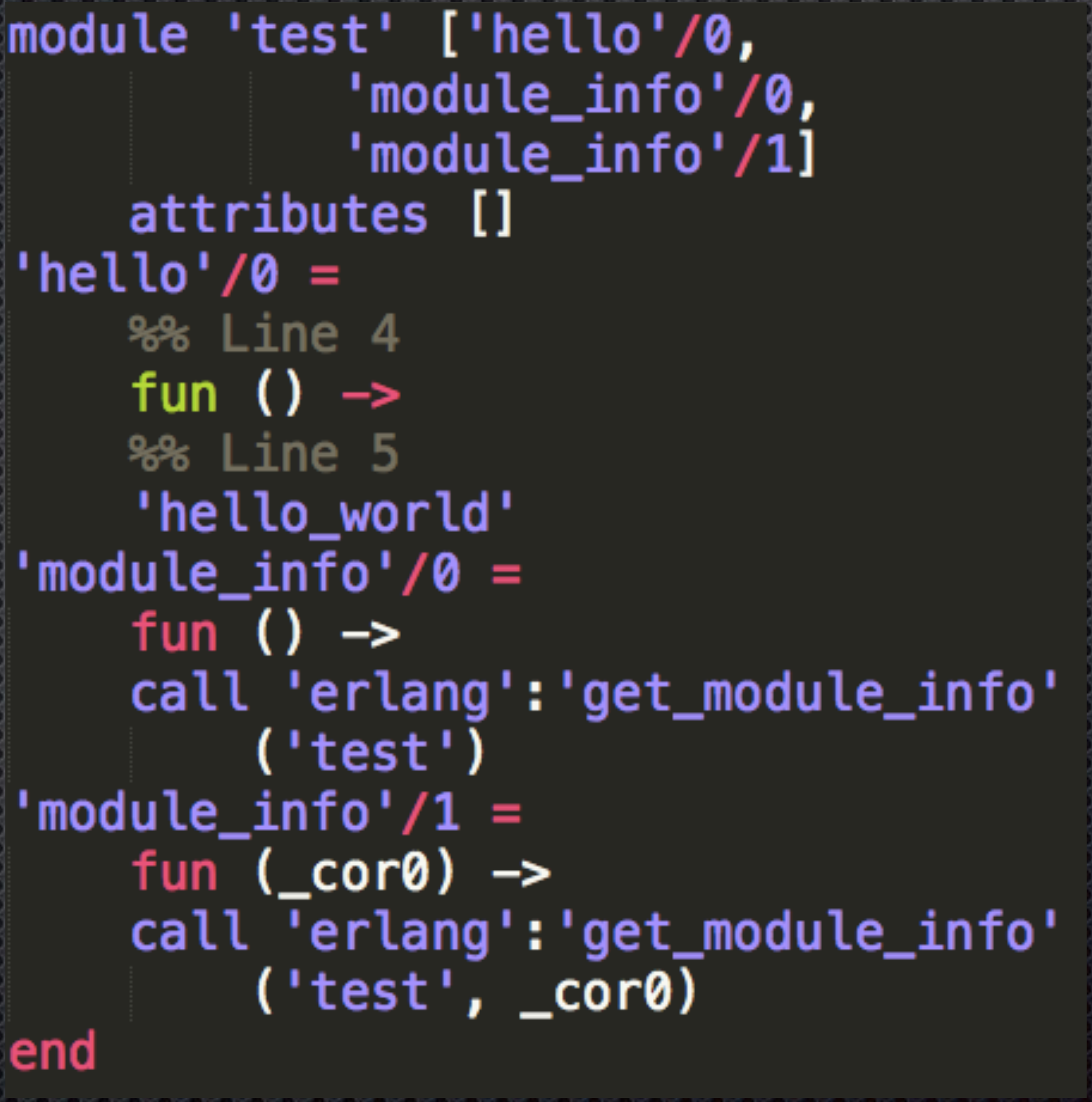
$ erlc +dexp test.erl



```
test.
[{hello,0},{module_info,0},{module_info,1}].
{attribute,1,file,{"test.erl",1}}.
{function,4,hello,0,[{clause,4,[],[],
  [{atom,5,hello_world}]}]}.
{function,0,module_info,0,
  [{clause,0,[],[],
    [{call,0,
    {remote,0,{atom,0,erlang},
      {atom,0,get_module_info}},
      [{atom,0,test}]}]}]}.
{function,0,module_info,1,
  [...]}.
```

Core Erlang Code

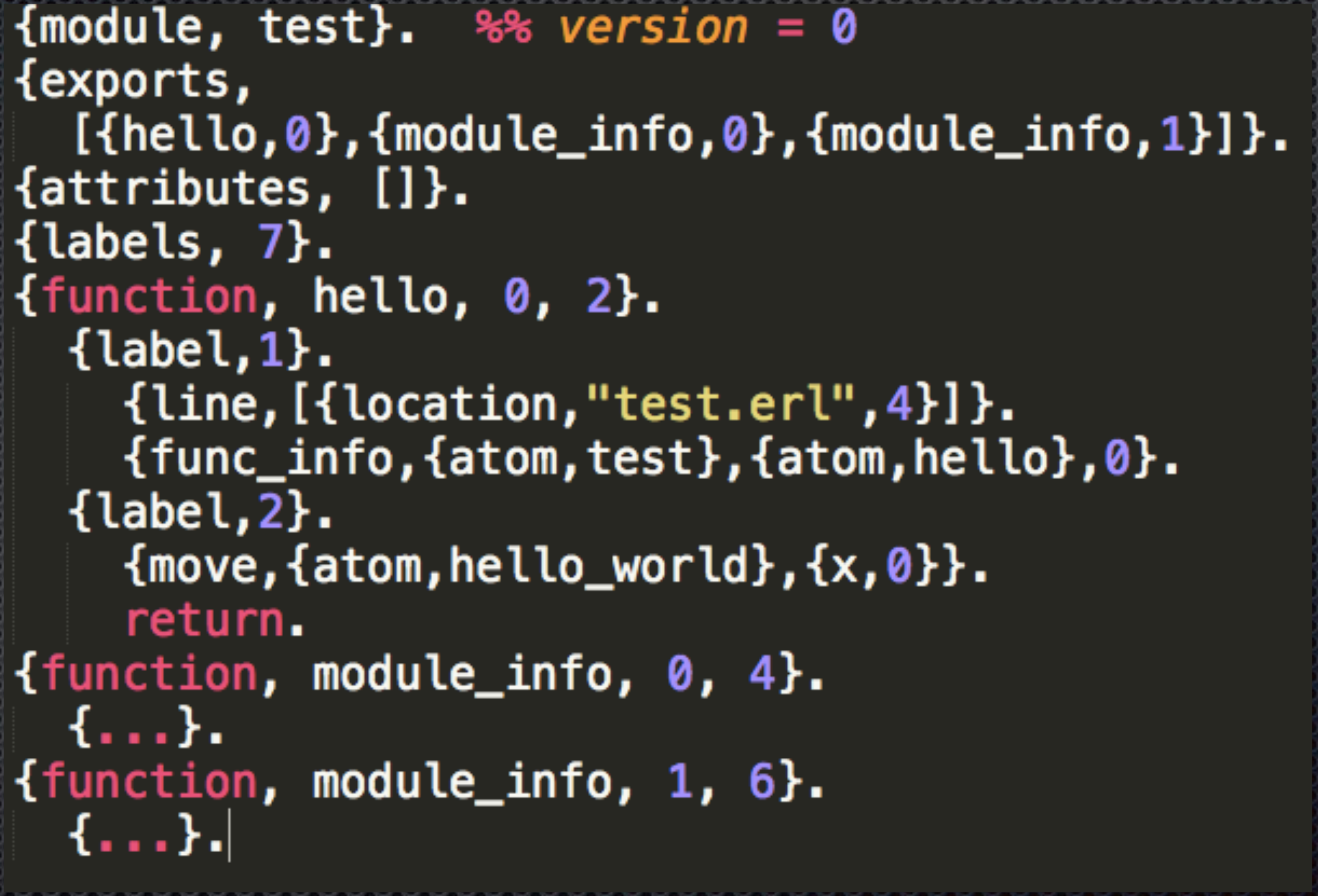
$ erlc +to_core test.erl



```
module 'test' ['hello'/0,
               'module_info'/0,
               'module_info'/1]
    attributes []
'hello'/0 =
    %% Line 4
    fun () ->
    %% Line 5
    'hello_world'
'module_info'/0 =
    fun () ->
    call 'erlang':'get_module_info'
        ('test')
'module_info'/1 =
    fun (_cor0) ->
    call 'erlang':'get_module_info'
        ('test', _cor0)
end
```

Assembler Code

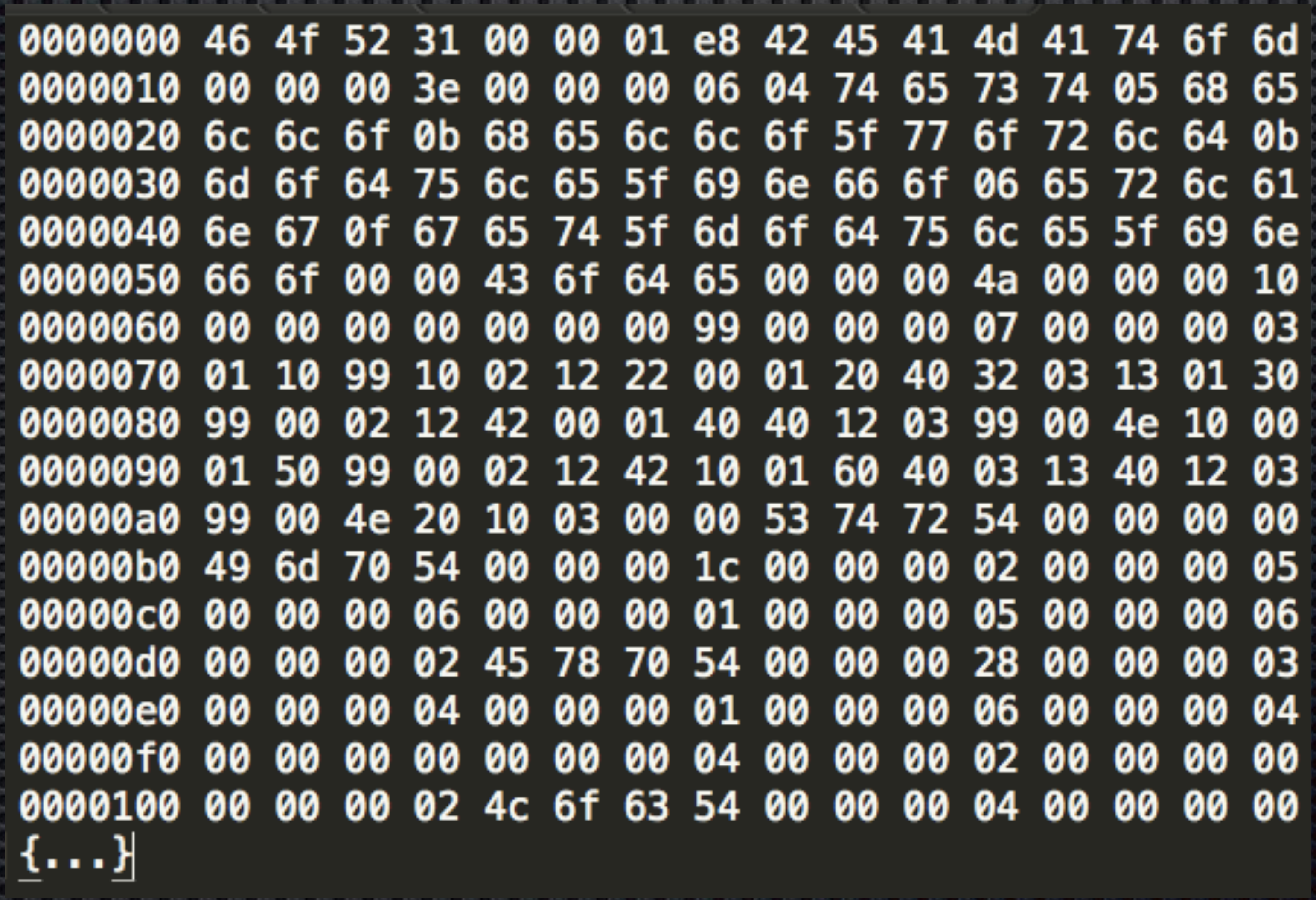
$ erlc -S test.erl



```
{module, test}.  %% version = 0
{exports,
  [{hello,0},{module_info,0},{module_info,1}]}.
{attributes, []}.
{labels, 7}.
{function, hello, 0, 2}.
  {label,1}.
    {line,[{location,"test.erl",4}]}.
    {func_info,{atom,test},{atom,hello},0}.
  {label,2}.
    {move,{atom,hello_world},{x,0}}.
    return.
{function, module_info, 0, 4}.
  {...}.
{function, module_info, 1, 6}.
  {...}.
```

BEAM Byte Code

$ erlc test.erl



```
0000000 46 4f 52 31 00 00 01 e8 42 45 41 4d 41 74 6f 6d
0000010 00 00 00 3e 00 00 00 06 04 74 65 73 74 05 68 65
0000020 6c 6c 6f 0b 68 65 6c 6c 6f 5f 77 6f 72 6c 64 0b
0000030 6d 6f 64 75 6c 65 5f 69 6e 66 6f 06 65 72 6c 61
0000040 6e 67 0f 67 65 74 5f 6d 6f 64 75 6c 65 5f 69 6e
0000050 66 6f 00 00 43 6f 64 65 00 00 00 4a 00 00 00 10
0000060 00 00 00 00 00 00 00 99 00 00 00 07 00 00 00 03
0000070 01 10 99 10 02 12 22 00 01 20 40 32 03 13 01 30
0000080 99 00 02 12 42 00 01 40 40 12 03 99 00 4e 10 00
0000090 01 50 99 00 02 12 42 10 01 60 40 03 13 40 12 03
00000a0 99 00 4e 20 10 03 00 00 53 74 72 54 00 00 00 00
00000b0 49 6d 70 54 00 00 00 1c 00 00 00 02 00 00 00 05
00000c0 00 00 00 06 00 00 00 01 00 00 00 05 00 00 00 06
00000d0 00 00 00 02 45 78 70 54 00 00 00 28 00 00 00 03
00000e0 00 00 00 04 00 00 00 01 00 00 00 06 00 00 00 04
00000f0 00 00 00 00 00 00 00 04 00 00 00 02 00 00 00 00
0000100 00 00 00 02 4c 6f 63 54 00 00 00 04 00 00 00 00
{...}
```

Lets create a query language for Tnesia

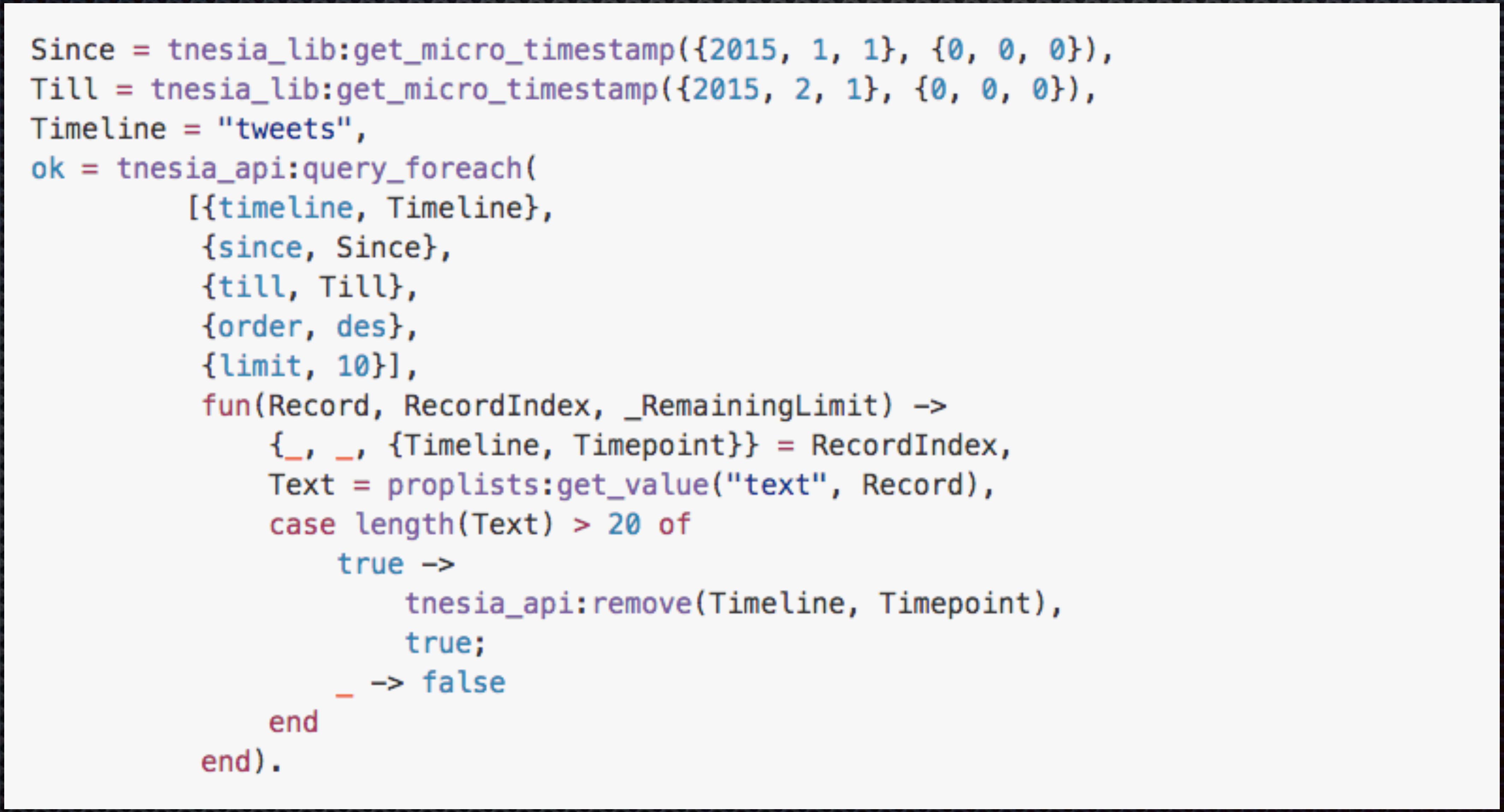
What is Tnesia?

“Tnesia is a time-series data storage which lets you run time-based queries on a large amount of data, without scanning the whole set of data, and in a key-value manner. It can be used embedded inside an Erlang application, or stand-alone with HTTP interface to outside which talks in a simple query language called TQL.”

– *https://github.com/bisphone/Tnesia*

The Problem

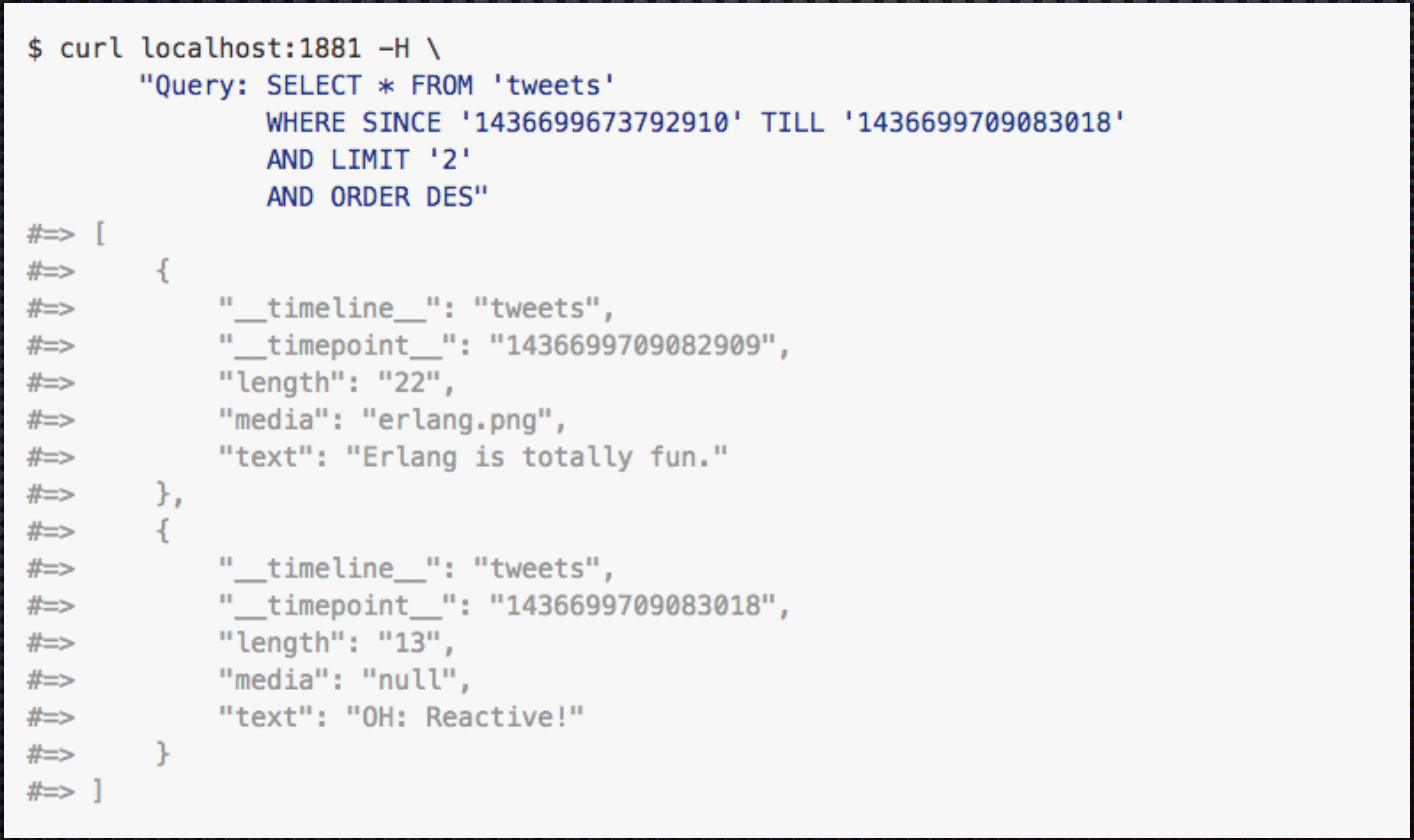
It seems that its Erlang API looks strange to non-Erlang developer!



```erlang
Since = tnesia_lib:get_micro_timestamp({2015, 1, 1}, {0, 0, 0}),
Till = tnesia_lib:get_micro_timestamp({2015, 2, 1}, {0, 0, 0}),
Timeline = "tweets",
ok = tnesia_api:query_foreach(
         [{timeline, Timeline},
          {since, Since},
          {till, Till},
          {order, des},
          {limit, 10}],
          fun(Record, RecordIndex, _RemainingLimit) ->
              {_, _, {Timeline, Timepoint}} = RecordIndex,
              Text = proplists:get_value("text", Record),
              case length(Text) > 20 of
                  true ->
                      tnesia_api:remove(Timeline, Timepoint),
                      true;
                  _ -> false
              end
          end).
```

The Solution

Create a Query Language (TQL) that can be used over HTTP.



```
$ curl localhost:1881 -H \
       "Query: SELECT * FROM 'tweets'
               WHERE SINCE '1436699673792910' TILL '1436699709083018'
               AND LIMIT '2'
               AND ORDER DES"
#=> [
#=>     {
#=>         "__timeline__": "tweets",
#=>         "__timepoint__": "1436699709082909",
#=>         "length": "22",
#=>         "media": "erlang.png",
#=>         "text": "Erlang is totally fun."
#=>     },
#=>     {
#=>         "__timeline__": "tweets",
#=>         "__timepoint__": "1436699709083018",
#=>         "length": "13",
#=>         "media": "null",
#=>         "text": "OH: Reactive!"
#=>     }
#=> ]
```

TQL Concepts

Syntax

```
SELECT all | record_keys()
FROM timeline()
[ WHERE
    [ [ AND ] conditions() ]
    [ [ AND ] SINCE datetime() TILL datetime() ]
    [ [ AND ] ORDER order() ]
    [ [ AND ] LIMIT limit() ] ]
```

```
INSERT INTO timeline() record_keys()
RECORDS record_values()
```

```
DELETE FROM timeline()
WHEN record_time()
```

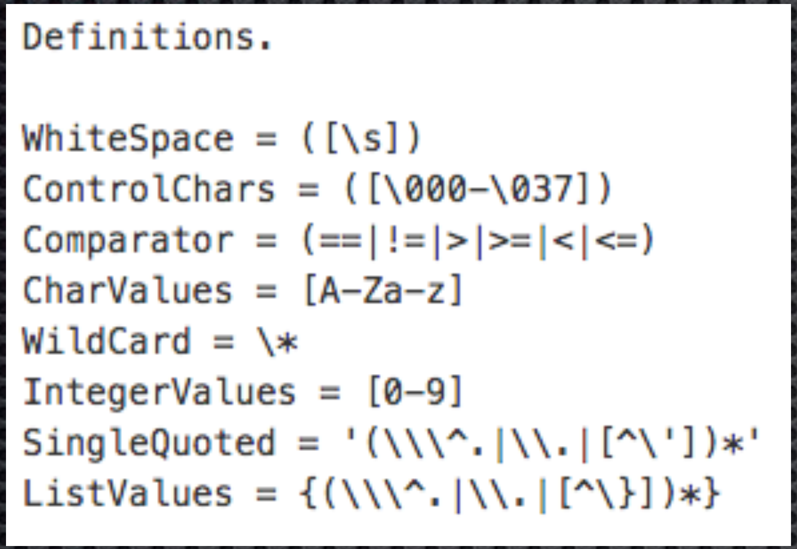
Types

```
timeline() :: 'string()'
record_keys() :: {'string()', ...}
record_values() :: {'string()', ...}
record_time() :: 'string()'
datetime() :: 'integer()'
order() :: 'asc' | 'des'
limit() :: 'integer()'
conditions() :: condition() AND condition()
condition() :: 'string()' comparator() 'string()'
comparator() :: == | != | > | >= | < | <=
```

Lexical Scanning

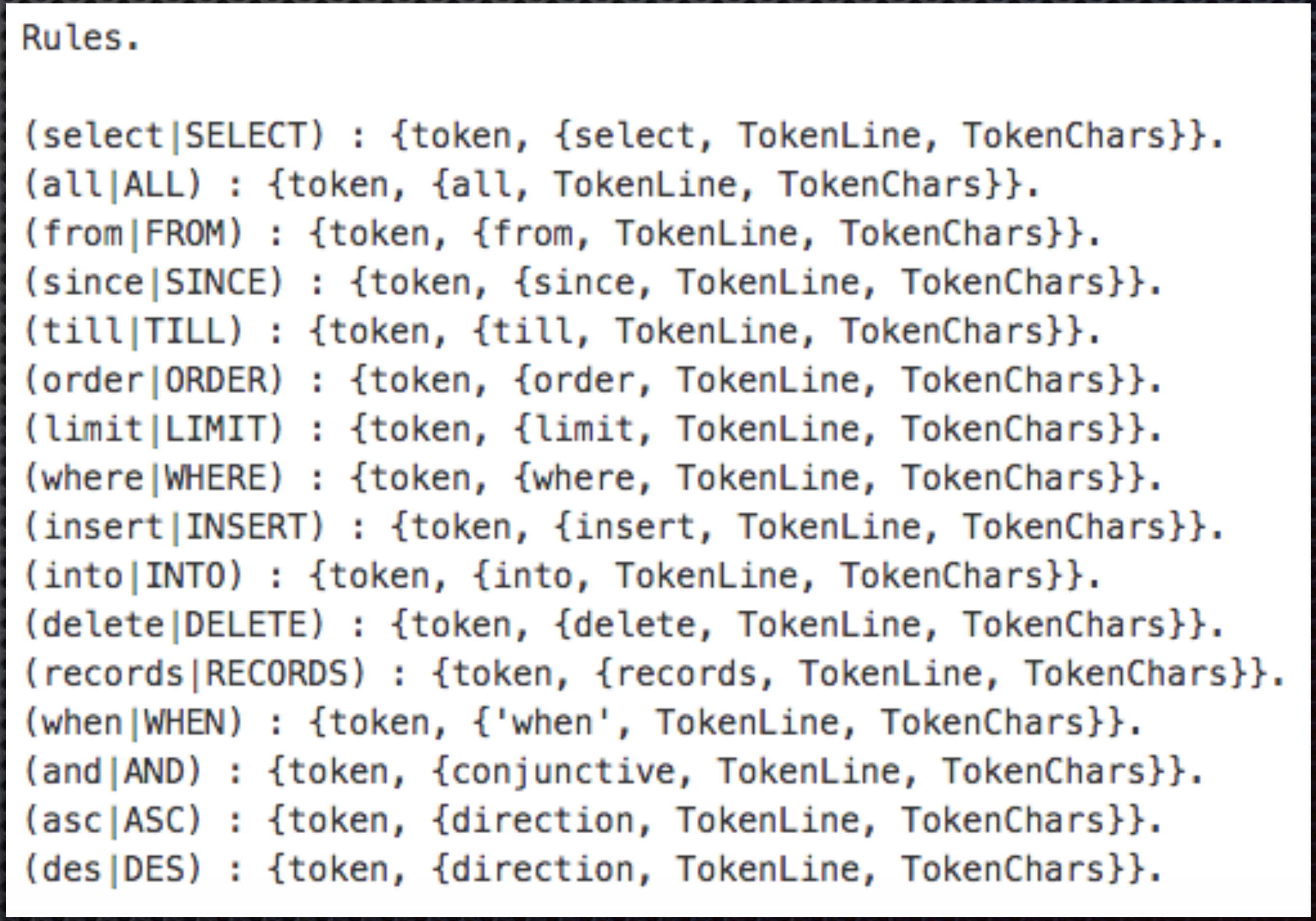
Using leex which is a Erlang lexer

Definitions



```
Definitions.

WhiteSpace = ([\s])
ControlChars = ([\000-\037])
Comparator = (==|!=|>|>=|<|<=)
CharValues = [A-Za-z]
WildCard = \*
IntegerValues = [0-9]
SingleQuoted = '(\\\^.|\\.|[^\'])*'
ListValues = {(\\\^.|\\.|[^\}])*}
```

Rules

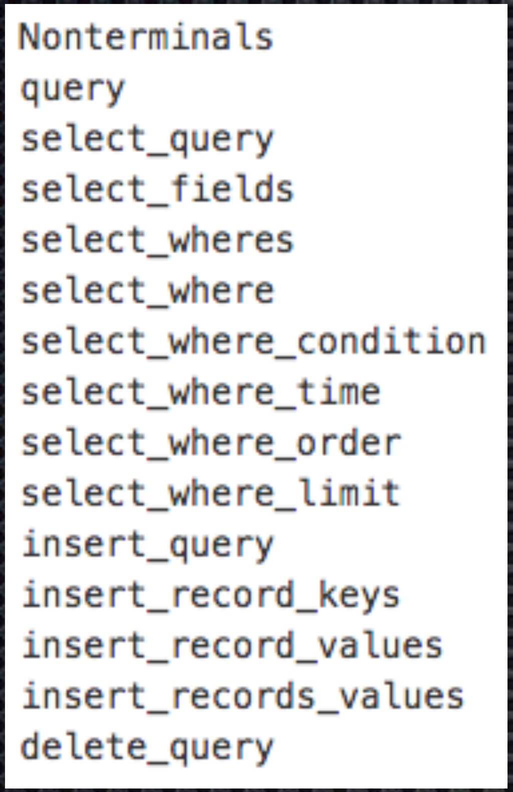


```
Rules.

(select|SELECT) : {token, {select, TokenLine, TokenChars}}.
(all|ALL) : {token, {all, TokenLine, TokenChars}}.
(from|FROM) : {token, {from, TokenLine, TokenChars}}.
(since|SINCE) : {token, {since, TokenLine, TokenChars}}.
(till|TILL) : {token, {till, TokenLine, TokenChars}}.
(order|ORDER) : {token, {order, TokenLine, TokenChars}}.
(limit|LIMIT) : {token, {limit, TokenLine, TokenChars}}.
(where|WHERE) : {token, {where, TokenLine, TokenChars}}.
(insert|INSERT) : {token, {insert, TokenLine, TokenChars}}.
(into|INTO) : {token, {into, TokenLine, TokenChars}}.
(delete|DELETE) : {token, {delete, TokenLine, TokenChars}}.
(records|RECORDS) : {token, {records, TokenLine, TokenChars}}.
(when|WHEN) : {token, {'when', TokenLine, TokenChars}}.
(and|AND) : {token, {conjunctive, TokenLine, TokenChars}}.
(asc|ASC) : {token, {direction, TokenLine, TokenChars}}.
(des|DES) : {token, {direction, TokenLine, TokenChars}}.
```

Parsing

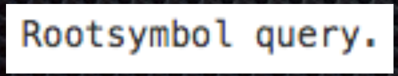
Using yecc which is an Erlang parser generator

Non-terminals



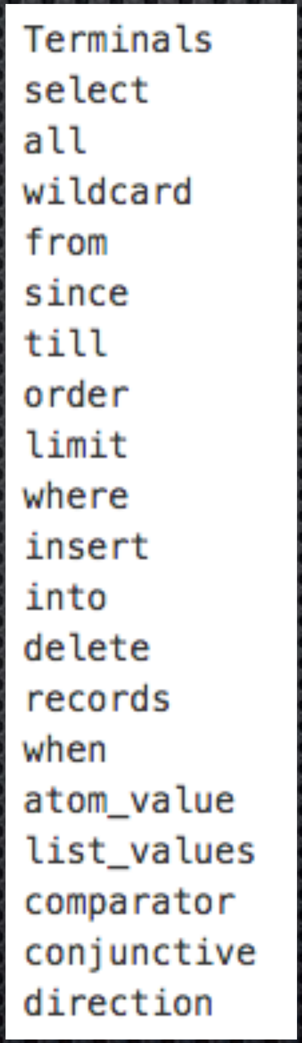
```
Nonterminals
query
select_query
select_fields
select_wheres
select_where
select_where_condition
select_where_time
select_where_order
select_where_limit
insert_query
insert_record_keys
insert_record_values
insert_records_values
delete_query
```

Root-symbol



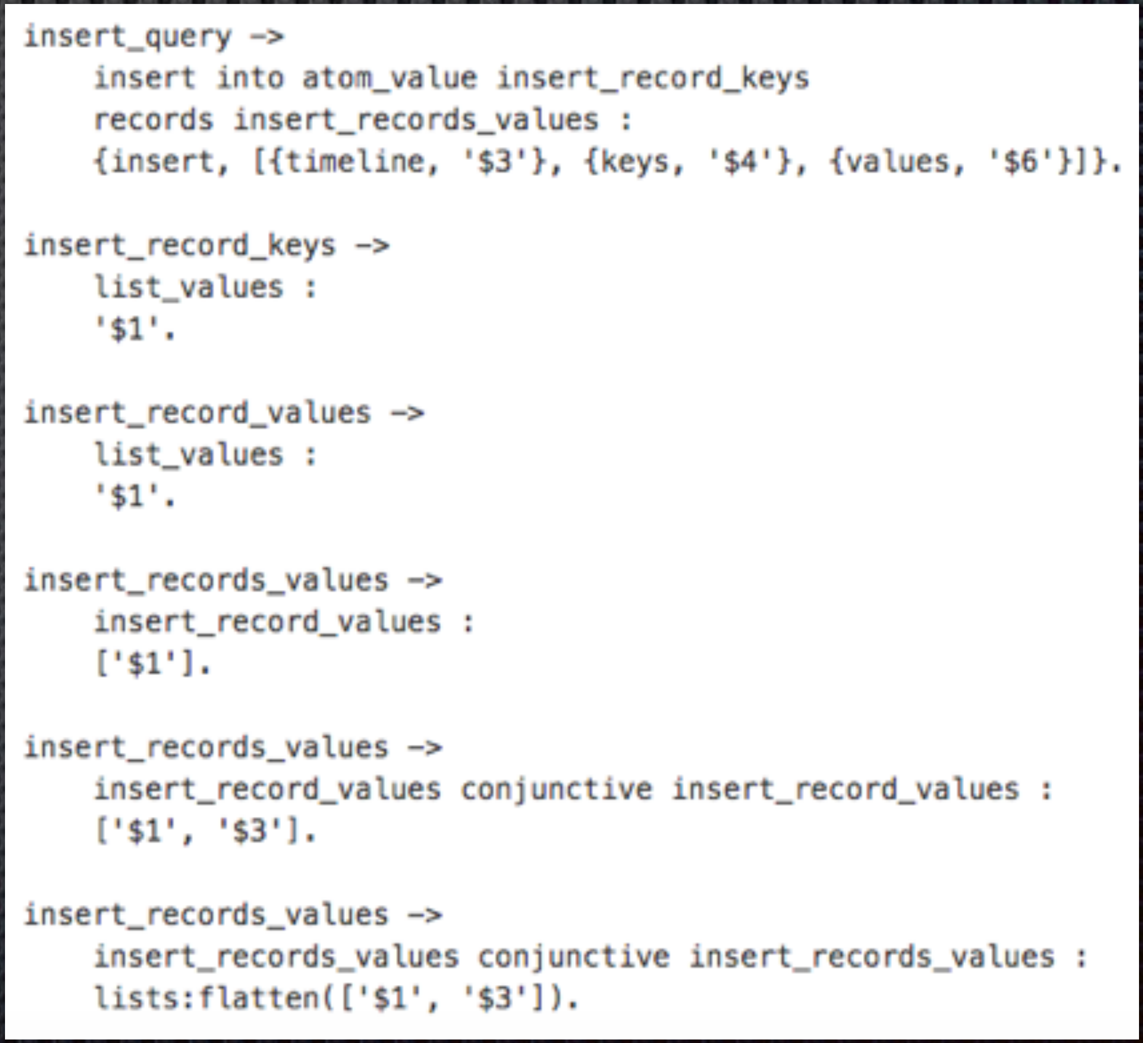
```
Rootsymbol query.
```

Terminals



```
Terminals
select
all
wildcard
from
since
till
order
limit
where
insert
into
delete
records
when
atom_value
list_values
comparator
conjunctive
direction
```

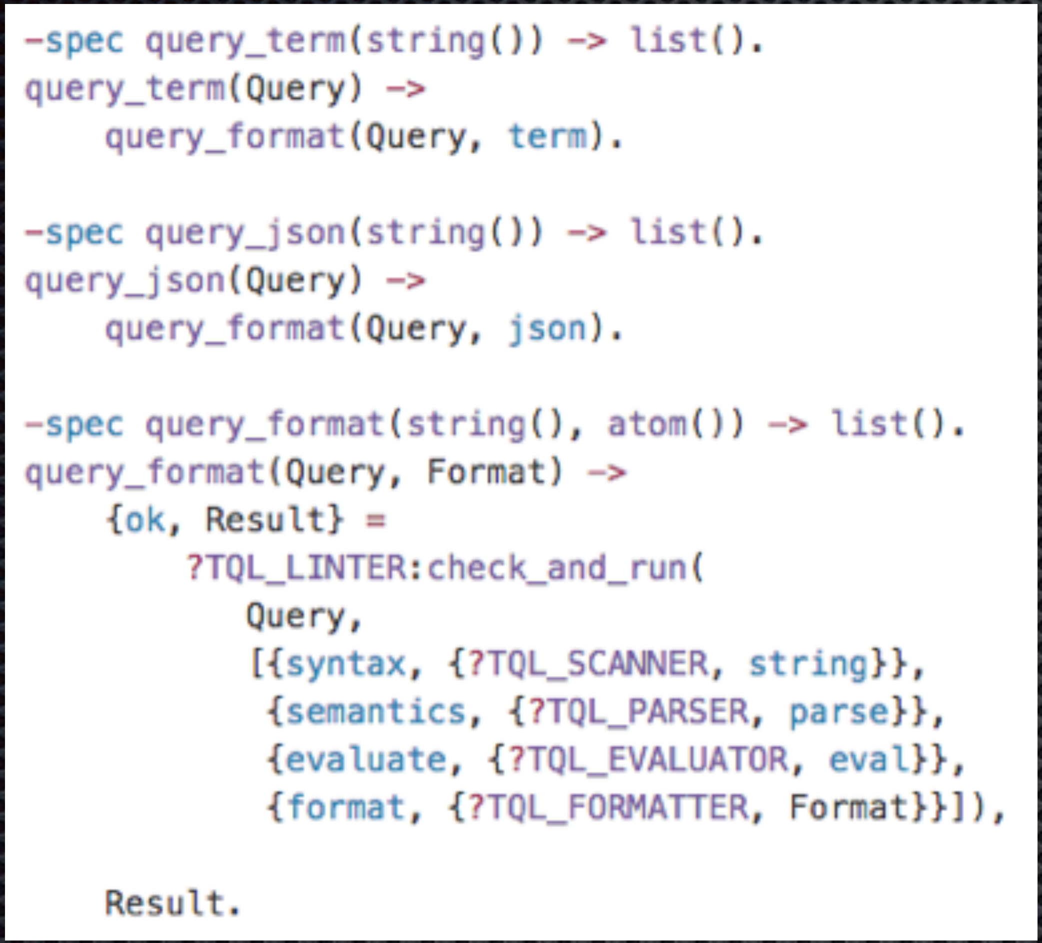
Rules



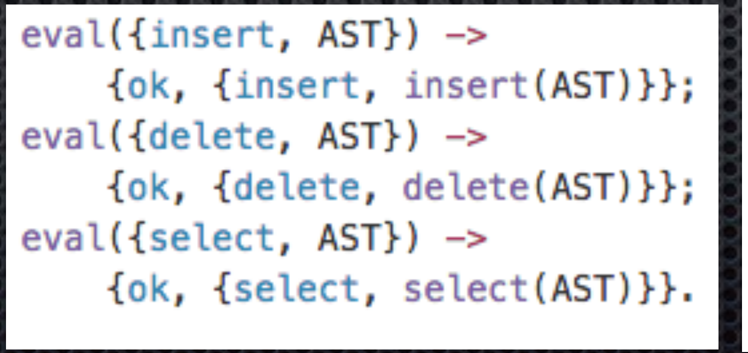
```
insert_query ->
    insert into atom_value insert_record_keys
    records insert_records_values :
    {insert, [{timeline, '$3'}, {keys, '$4'}, {values, '$6'}]}.

insert_record_keys ->
    list_values :
    '$1'.

insert_record_values ->
    list_values :
    '$1'.

insert_records_values ->
    insert_record_values :
    ['$1'].

insert_records_values ->
    insert_record_values conjunctive insert_record_values :
    ['$1', '$3'].

insert_records_values ->
    insert_records_values conjunctive insert_records_values :
    lists:flatten(['$1', '$3']).
```

Evaluating

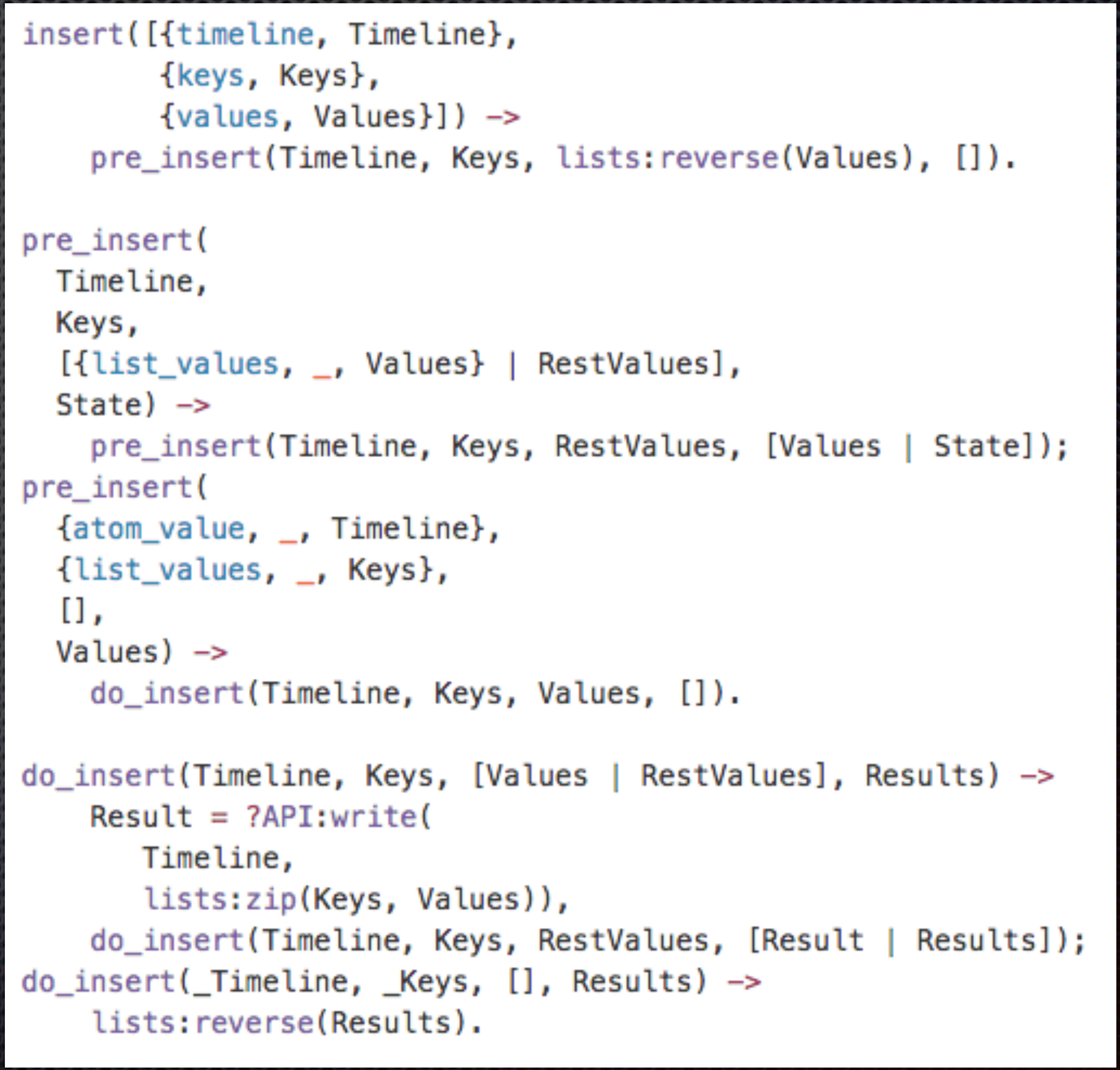
Evaluating AST in Erlang without any intermediate code generation



```erlang
-spec query_term(string()) -> list().
query_term(Query) ->
    query_format(Query, term).

-spec query_json(string()) -> list().
query_json(Query) ->
    query_format(Query, json).

-spec query_format(string(), atom()) -> list().
query_format(Query, Format) ->
    {ok, Result} =
        ?TQL_LINTER:check_and_run(
           Query,
           [{syntax, {?TQL_SCANNER, string}},
            {semantics, {?TQL_PARSER, parse}},
            {evaluate, {?TQL_EVALUATOR, eval}},
            {format, {?TQL_FORMATTER, Format}}]),

    Result.
```



```erlang
eval({insert, AST}) ->
    {ok, {insert, insert(AST)}};
eval({delete, AST}) ->
    {ok, {delete, delete(AST)}};
eval({select, AST}) ->
    {ok, {select, select(AST)}}.
```



```erlang
insert([{timeline, Timeline},
        {keys, Keys},
        {values, Values}]) ->
    pre_insert(Timeline, Keys, lists:reverse(Values), []).

pre_insert(
  Timeline,
  Keys,
  [{list_values, _, Values} | RestValues],
  State) ->
    pre_insert(Timeline, Keys, RestValues, [Values | State]);
pre_insert(
  {atom_value, _, Timeline},
  {list_values, _, Keys},
  [],
  Values) ->
    do_insert(Timeline, Keys, Values, []).

do_insert(Timeline, Keys, [Values | RestValues], Results) ->
    Result = ?API:write(
        Timeline,
        lists:zip(Keys, Values)),
    do_insert(Timeline, Keys, RestValues, [Result | Results]);
do_insert(_Timeline, _Keys, [], Results) ->
    lists:reverse(Results).
```

Question || Comment

- https://hamidreza-s.github.com