Robert Virding

Principle Language Expert
at Erlang Solutions Ltd.

Erlang Solutions Ltd.

Implementing languages on the Erlang VM

- What languages?
- Basic Tools
- 3 case studies + interesting techniques
  - LFE (Lisp Flavoured Erlang)
  - Erlog (Prolog)
  - Luerl (Lua)

What languages?

- Anything written in another language
  - Config files
  - DSLs
  - Other “languages”
  - ...

Basic tools

- leex - lexical scanner generator
- yecc - parser generator
- syntax tools - for building erlang code
- XML parsers (xmlerl
- compiler (of course)

leex

- lexical scanner generator
- based on lex/flex (but simpler)
- uses regular expressions to define tokens
- generates scanning functions
  - direct use
    `string/2`
  - for the i/o system
    `token/2, tokens/2`

leex - example

```
Definitions.

U = [A-Z]
L = [a-z]
D = [0-9]

Rules.

{L}({U}|{L}|{D}|_)* :
        {token,{atom,TokenLine,list_to_atom(TokenChars)}}.
({U}|_)({U}|{L}|{D}|_)* :
        {token,{var,TokenLine,list_to_atom(TokenChars)}}.
%[^\n]* :   skip_token.
```

leex - usage

- direct usage

```
{ok,Tokens,EndLine} = my_scan:string(Chars, StartLine)
```

- i/o system

```
{ok,Tokens,EndLine} =
    io:request(Io, {get_until,'',my_scan,tokens,[StartLine]})
```

yecc

- LALR(1) parser generator
- based on yacc
- generating parsing functions

  `parse/1`

yecc - example

```
Nonterminals E T F.
Terminals '+' '*' '(' ')' number.
Rootsymbol E.

E -> E '+' T : {'+','$1','$2'}.
E -> E '-' T : {'-','$1','$2'}.
E -> T : '$1'.
T -> T '*' F : {'*','$1','$2'}.
T -> T '/' F : {'/','$1','$2'}.
F -> '(' E ')' : '$2'.
F -> number : '$1'
```

yecc - reduce-reduce errors

- would like to write

```
expr -> pattern '=' expr : ... .
```

- but have to write

```
expr -> expr '=' expr : ... .
```

  - and then check the pattern

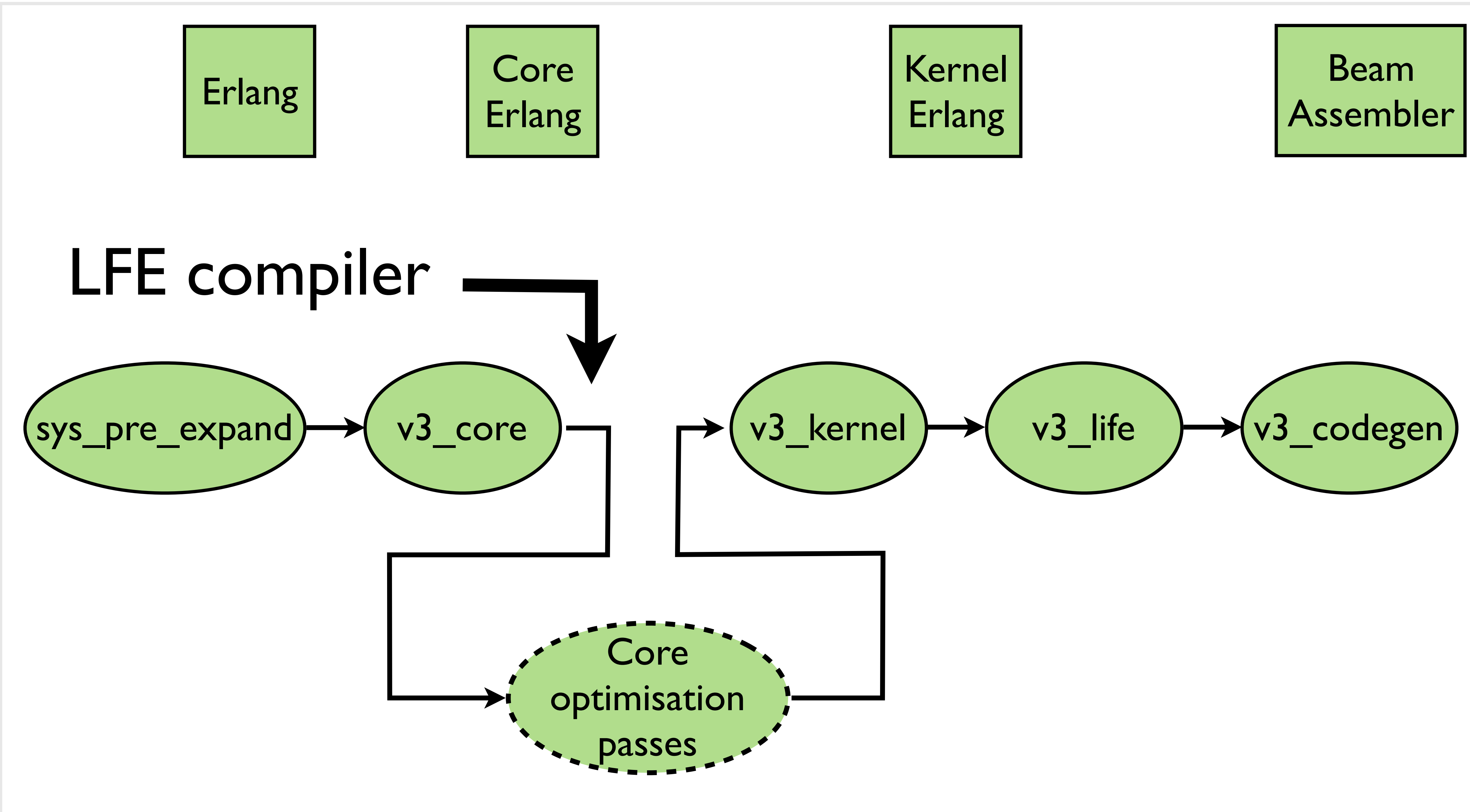
Other tools

- syntax tools
  - library for creating and working with erlang code
- other parsing systems
  - PEG parsers

Erlang compiler

- can work on files and erlang abstract code
- can generate `.beam` files or binaries
- has Core, a nice intermediate language
  - simple and regular
  - easier to compile to

Erlang compiler



Core erlang

- simple functional language
- "normal" lexical scoping
- has just the basics
  - no records
  - no list comprehensions
- supports pattern matching (yeah!)
- most optimisations done on core
- dialyzer speaks Core

Core erlang - example

```
fac(N) when N > 0 ->
    N * fac(N-1);
fac(0) -> 0.

'fac'/1 =
    fun (_cor0) ->
        case _cor0 of
          <N> when call 'erlang':'>'(N, 0) ->
              let <_cor1> = call 'erlang':'-'(N, 1)
              in  let <_cor2> = apply 'fac'/1 (_cor1)
                  in call 'erlang':'*' (N, _cor2)
          <0> when 'true' -> 0
          ( <_cor3> when 'true' ->
                ( primop 'match_fail' ({'function_clause',_cor3})
                  -| [{'function_name',{'fac',1}}] )
            -| ['compiler_generated'] )
        end
```

LFE (Lisp Flavoured Erlang)

The goal was:

- provide lots of lisp goodies
  - real homoiconicity and macros (yeah!)
- seamlessly interact with vanilla Erlang/OTP
  - be able to freely mix vanilla code and LFE code
- small core language
- same speed as vanilla Erlang

LFE (Lisp Flavoured Erlang)

➡ modify the language to fit with Erlang/OTP

- no mutable data
- only have standard Erlang data types
  - Erlang style records
- Erlang style modules and functions
  - no functions with variable number of arguments
- macros are only compile-time
- Lisp-2 (instead of Lisp-1)

- is as fast as vanilla Erlang

LFE - core

compile to Core Erlang

- LFE core primitives ARE Core Erlang

```
(case expr clause ... )
(if test true-expr false-expr)
(receive clause ... (after timeout ... ))
(catch ... )
(try expr (case ... ) (catch ... ) (after ... ))
(lambda ... )
(match-lambda clause ... )
(let ... )
(let-function ... ), (letrec-function ... )
(cons h t), (list ... ), (tuple ... ) (binary ... )
(funcall var arg ... )
(call mod func arg ... )
(define-function name lambda | match-lambda)
```

Erlog

- standard prolog, at least a strict subset
- completely different semantics to Erlang
  - backtracking
  - logical variables
  - unification

Erlog

- good mapping between Erlog <-> Erlang data structures
  - atoms <-> atoms
  - number <-> numbers
  - lists <-> lists
  - structures <-> tuples

    `foo(bar, 42)` <-> `{foo,bar,42}`

Erlog

- except for logical variables
  - they are both “global” and “mutable”
  - they can be unified `X = Y`
  - changes seen everywhere
  - no corresponding Erlang data structure
    - we are cunning and use `{Index}`

➡ must be kept in a global table
  - dict or array
  - keep copies on stack so automatic reseting on backtracking

Backtracking

- Erlang does not have backtracking
  - Once a choice has been made it has been made
  - Once a function has been evaluated it is done
- It is impossible to go back and try an alternative
  - At least in the language
- This is a very nice feature, but we must explicitly do it ourselves!

Backtracking

```
top_call(Data) ->
    Succeed = fun (D) -> {succeed,D} end,          %We have succeeded
    find_solution(Data, Succeed).

a(Data, Next) ->
    NewData = do_stuff(Data),                      %Our stuff
    OurNext = fun (D) -> even_more_stuff(D, Next) end,
    more_stuff_a(NewData, OurNext).

b(Data, Next) ->
    NewData = do_stuff(Data),
    %% Set up our choice point.
    cp([fun () -> more_stuff_a(NewData, Next) end,
        fun () -> more_stuff_b(NewData, Next) end,
        fun () -> more_stuff_c(NewData, Next) end]).
```

Backtracking - choice points

```
cp([G|Gs]) ->
    case G() of
        {succeed,Value} -> {succeed,Value};
        fail -> cp(Gs)
    end;
cp([]) -> fail.
```

Luerl

- implement standard Lua 5.2
- simple, rather neat little *imperative* language
- dynamic language
- lexically scoped
- mutable variables/environments

Luerl - Lua datatypes

- nil
- booleans
- numbers (floating point)
- strings
- *mutable* key-value tables
  - which it uses as tables/arrays/lists/kitchen sink
  - updates are visible everywhere

Luerl - table store

➡ need to manage global data

- global table store
    - orddict/array/process dictionary/ETS tables
- environments
    - orddict (for now)
- tables
    - array + orddict

```
-record(luerl, {tabs,free,next,          %Table structure
                meta=[],                 %Data type metatables
                env,                     %Environment
                tag}).                   %Unique tag
```

Luerl - table store

```
get_table_key(#tref{}=Tref, Key, St) when is_number(Key) ->
    case ?IS_INTEGER(Key, I) of
        true -> get_table_int_key(Tref, Key, I, St);
        false -> get_table_key_key(Tref, Key, St)
    end;
get_table_key(#tref{}=Tref, Key, St) ->
    get_table_key_key(Tref, Key, St);
get_table_key(Tab, Key, St) ->                  %Just find the metamethod
    case getmetamethod(Tab, <<"__index">>, St) of
        nil -> lua_error({illegal_index,Tab,Key});
        Meth when element(1, Meth) =:= function ->
            {Vs,St1} = functioncall(Meth, [Tab,Key], St),
            {first_value(Vs),St1};              %Only one value
        Meth ->                                 %Recurse down the metatable
            get_table_key(Meth, Key, St)
    end.
```

Luerl - table store

```
get_table_key_key(#tref{i=N}=T, Key, #luerl{tabs=Ts}=St) ->
    #table{t=Tab,m=Meta} = ?GET_TABLE(N, Ts),   %Get the table.
    case orddict:find(Key, Tab) of
        {ok,Val} -> {Val,St};
        error ->
            %% Key not present so try metamethod
            get_table_metamethod(T, Meta, Key, Ts, St)
    end.
get_table_int_key(#tref{i=N}=T, Key, I, #luerl{tabs=Ts}=St) ->
    #table{a=A,m=Meta} = ?GET_TABLE(N, Ts),     %Get the table.
    case array:get(I, A) of
        undefined ->
            %% Key not present so try metamethod
            get_table_metamethod(T, Meta, Key, Ts, St);
        Val -> {Val,St}
    end.
```

Luerl - table store

```
get_table_metamethod(T, Meta, Key, Ts, St) ->
    case getmetamethod_tab(Meta, <<"__index">>, Ts) of
        nil -> {nil,St};
        Meth when element(1, Meth) =:= function ->
            {Vs,St1} = functioncall(Meth, [T,Key], St),
            {first_value(Vs),St1};      %Only one value
        Meth ->                         %Recurse down the metatable
            get_table_key(Meth, Key, St)
    end.
```

Luerl - table store

```
set_table_key_key(#tref{i=N}, Key, Val, #luerl{tabs=Ts0}=St) ->
    #table{t=Tab0,m=Meta}=T = ?GET_TABLE(N, Ts0),       %Get the table
    case orddict:find(Key, Tab0) of
        {ok,_} ->
            Tab1 = orddict:store(Key, Val, Tab0),
            Ts1 = ?SET_TABLE(N, T#table{t=Tab1}, Ts0),
            St#luerl{tabs=Ts1};
        error ->
            case getmetamethod_tab(Meta, <<"__newindex">>, Ts0) of
                nil ->
                    Tab1 = orddict:store(Key, Val, Tab0),
                    Ts1 = ?SET_TABLE(N, T#table{t=Tab1}, Ts0),
                    St#luerl{tabs=Ts1};
                Meth when element(1, Meth) =:= function ->
                    functioncall(Meth, [Key,Val], St);
                Meth -> set_table_key(Meth, Key, Val, St)
            end
    end.
```