The Meaning of LFE

Zeeshan Lakhani

Software Engineer at Basho Technologies,Inc | Founder/Organizer Papers We Love
@zeeshanlakhani

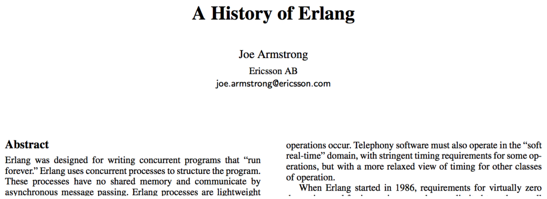
5-23-2015 (LambdaConf)

The Talk Within the Talk

Cheers Robert Virding

- Virding joined the Erlang team in 1988... at the time of "interpreted erlang"
- Virding created lfe in ~2008, "announcement" to erlang mailing list

Hello Erlang . . . and Bogdan/Björn's Erlang Abstract Machine


A History of Erlang

Joe Armstrong
Ericsson AB
joe.armstrong@ericsson.com

Abstract

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight

operations occur. Telephony software must also operate in the "soft real-time" domain, with stringent timing requirements for some operations, but with a more relaxed view of timing for other classes of operation.

When Erlang started in 1986, requirements for virtually zero

- 1986. . . OTP in 1996[1]
- COPL (Concurrency Oriented Programming Language)[2]
- Resilient to bugs and failures[3]

[1]Licentiate Thesis at KTH `http://bit.ly/1INtB8a`
[2]'Thesis' [Joe Armstrong] `http://bit.ly/1HvcOUi`
[3]Lessons from Erlang [Slaski] `http://bit.ly/1c6GvQ5`

1. COPLs must support processes. A process can be thought of as a self-contained virtual machine.

2. Several processes operating on the same machine must be strongly isolated. A fault in one processe should not adversely effect another process, unless such interaction is explicitly programmed.

3. Each process must be identified by a unique unforgeable identifier. We will call this the Pid of the process.

4. There should be no shared state between processes. Processes interact by sending messages. If you know the Pid of a process then you can send a message to the process.

5. Message passing is assumed to be unreliable with no guarantee of delivery.

6. It should be possible for one process to detect failure in another process. We should also know the reason for failure.

SMP (Symmetrical Multi Processor) in 2005/6

Beam without SMP - 1 scheduler on main process thread. Beam with SMP - 1 to many schedulers (based on cores), run in 1 thread each. Shared data structures protected w/ locks.[a]

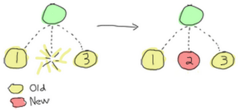
[a]Some facts about Erlang and SMP [Lundin] `http://bit.ly/1PyeraJ`

- Pattern Matching... Message arrives into mailbox (1 per process), on 'receive' try to match first item in mailbox

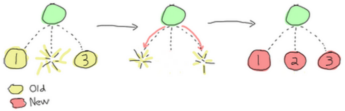
```
sync_index(Pid, IndexName, Timeout) ->
    process_flag(trap_exit, true),
    {ok, Ring} = riak_core_ring_manager:get_my_ring(),
    Nodes = riak_core_ring:all_members(Ring),
    WaitPid = spawn_link(?MODULE,
                         wait_for_index,
                         [self(), IndexName, Nodes]),
    receive
        {_From, ok} ->
            Pid ! ok;
        {'EXIT', _Pid, _Reason} ->
            sync_index(Pid, IndexName, Timeout)
    after Timeout ->
        exit(WaitPid, kill),
        %% Check if initFailure occurred
        {ok, _, S} = yz_solr:core(status,
                                  [{wt,json},
                                  {core, IndexName}]),
```

Supervision Trees (restart strategies)[4]

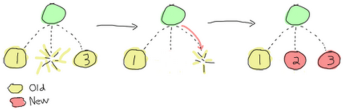
one for one



one for all



rest for one



[4]Learn You Some Erlang for Great Good [Hebert] `http://bit.ly/1SjbjOW`

Hello Lisp

- Recursive Functions of Symbolic Expressions and

Their Computation by Machine, Part I - April 1960[5]

- The whole language always available[6]
- The Lambda Papers (1975 - 80) - Steele & Sussman (Scheme)

[5]The John McCarthy Paper `http://stanford.io/1FA4PWs`
[6]What Made Lisp Different [Paul Graham] `http://bit.ly/1GsueSG`

'()[7]

- Is it true that this is an S-expression? xyz
- Is it true that this is an S-expression? (how are you doing so far)

YES. YES.

[7] The Little Schemer [Friedman, Felleisen]

Monad Hello Scheme

- A Schemer's View of Monads by Foltzer, Friedman

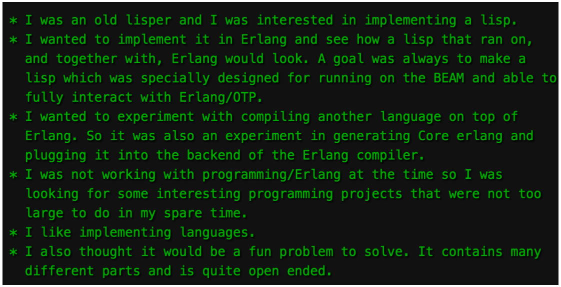
*The notion of abstract syntax is due to McCarthy <1963>, who designed the abstract syntax for Lisp <McCarthy et. al 1962>. The abstract syntax was intended to be used writing programs until designers could get around to create a concrete syntax with human-readable punctuation (instead of *L*ots of *I*rritating *S*illy *P*arentheses), but programmers soon got used to programming directly in abstract syntax.*[8]

[8] Appel's Modern Compiler Implementation in *

Hello LFE[10]

- code <=> data | homoiconicity | etc...

There were a number of reasons why Virding started with LFE:[9]



```
* I was an old lisper and I was interested in implementing a lisp.
* I wanted to implement it in Erlang and see how a lisp that ran on,
  and together with, Erlang would look. A goal was always to make a
  lisp which was specially designed for running on the BEAM and able to
  fully interact with Erlang/OTP.
* I wanted to experiment with compiling another language on top of
  Erlang. So it was also an experiment in generating Core erlang and
  plugging it into the backend of the Erlang compiler.
* I was not working with programming/Erlang at the time so I was
  looking for some interesting programming projects that were not too
  large to do in my spare time.
* I like implementing languages.
* I also thought it would be a fun problem to solve. It contains many
  different parts and is quite open ended.
```

[9] Secret History of LFE `http://bit.ly/1R6FKq9`
[10] lfe examples `http://bit.ly/1PYNNCJ`

```
(defun print-result ()
  (receive
    ((tuple pid msg)
      (io:format "Received message: '~p'~n" (list msg))
      (io:format "Sending message to process ~p ...~n" (list pid))
      (! pid (tuple msg))
      (print-result))))

(defun send-message (calling-pid msg)
  (let ([spawned-pid (spawn 'lambdaconf-proj 'print-result ())])
    (! spawned-pid (tuple calling-pid msg))))

> (lambdaconf-proj:send-message (self) 'hello)
#(<0.26.0> hello)
Received message: 'hello'
> Sending message to process <0.26.0> ...
(lambdaconf-proj:send-message (self) 'world)
#(<0.26.0> world)
Received message: 'world'
> Sending message to process <0.26.0> ...
(c:flush)
Shell got {hello}
Shell got {world}
ok
```

Binary Pattern Matching

```
> (defun pmbin ()
   (let (((binary (r (size 5)) (g (size 6)) (b (size 5)))
          #b(23 180)))
     (: io format '"~p ~p ~p~n" (list r g b))))
2 61 20
ok
```

do (from lisp) Iteration Primitive

As long as the condition is false, do executes the body repeatedly;

```
(defun do-run (x y)
  (do ((n x (+ n 1))
       (m y (- m 1))
       (c 0 (+ c 1)))
      ((begin
         ;; no real reason,
         ;; but use let b/c non static vals
         (print `(let ([c^ ,c]) c^))
         (> n m))
       c)))
 ;; (let ((c^ 94)) c^)->94
```

Joe's Fav[11]

Erlang

```
factorial_server() ->
    receive
        {From, N} ->
            From ! factorial(N),
            factorial_server()
    end.
```

LFE

```
(defun factorial-server ()
  (receive
    ((tuple from n)
     (! from (factorial n))
     (factorial-server))))
```

[11] My favorite Erlang program [Joe Armstrong] http://bit.ly/1FzV2zQ

GC Per Process (stack and a heap)[13]

12

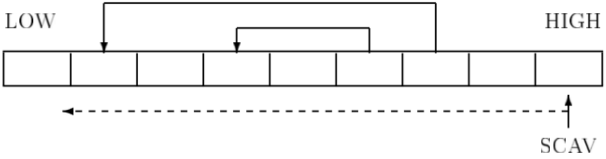


Fig. 1. Heap organisation.

- lower the address, greater the age
- history list - keep trace of age of objects to reclaim unmarked bits

[12][Armstrong, Virding] One Pass Real-Time Generational Mark-Sleep Garbage Collection

[13]A History of the Erlang VM [Virding] `http://bit.ly/1F34FTH`

Generational GC

probably (old) paper on generational mark-sweep based on the supposition that most objects only live a very short time while a small portion live much longer. More efficient: reclaim newly allocated objects more often than old objects. Hist list collector gets swept more at the beginning of the list. Erlang. . . no destructive operations that can create forward pointers.

Heap binaries (up to 64 bytes in size)

Store on each processes's heap

Binaries > 64 bytes

These are allocated in a separate heap outside the process scope. Reference counted binaries.

Interop

- Easy
- Elixir Interop (mostly) Works Too[14]

[14] The State of LFE [McGreggor] `http://bit.ly/1FCCzV5`

```
(defmodule some_props
  (export all)
  (import (from foo (hello2 1))))

(include-lib "eqc/include/eqc.hrl")
(include-lib "eqc/include/eqc_statem.hrl")

(defmacro NUM_TESTS () 100)

(defun prop_reverse ()
  (FORALL L
          (: eqc_gen list (: eqc_gen int))
          (== L (lists:reverse (lists:reverse L)))))



(defun reverse_helper (NumTests)
  (hello2 NumTests)
  (: eqc quickcheck
    (: eqc numtests NumTests (prop_reverse))))
```

```
LFE Shell V6.3.1 (abort with ^G)
> (c "src/some_props.lfe")
#(module some_props)
> (some_props:min_max_helper 100)
Tests: 100
Starting Quviq QuickCheck version 1.34.3
   (compiled at {{2015,5,11},{10,45,53}})
Licence for Basho reserved until {{2015,5,21},{3,56,34}}
xxxxxxxxxx.xx..xx.x.xx.x.x.x...x...x..x....xx.....x..x
...................x....x.x.....................(x10)...
(x1)xxxxxx
OK, passed 100 tests
true
```

S(Expression)peculative Discovery

REPL

Can define functions, variables (set, still single assignment), macros

One Lisp Here Two Lisp There

- Erlang's flat[15] namespace & convention
- Lisp-2 has distinct function & value namespaces.
- Lisp-2, the rules for evaluation in the functional position of a form are distinct from those for evaluation in the argument positions of the form. Common Lisp is a Lisp-2 dialect.[16]

```
> (defun xx (yy) yy)
xx
> (set xx 4)
4
> (xx 3)
3
> xx
4
```

[15] [Fred Hebert] http://bit.ly/1PYhdB6

[16] Technical Issues of Separation in Function Cells and Value Cells [Gabriel] http://bit.ly/1SgNtU6

Look at a Lisp-1 (clojure)

```
;; Give me some Clojure:
> (defn xx [yy] yy)
#'sandbox8948/xx
> (def xx 4)
#'sandbox8948/xx
> xx
4
> (xx 3)
java.lang.ClassCastException:
java.lang.Long cannot be cast to clojure.lang.IFn
```

Looking Back At Some Racket Homework

Racket

```
(define (sequence low high stride)
  (if (> low high)
      null
      (cons low (sequence
                  (+ low stride)
                  high stride))))
```

LFE

```
(defun sequence (low high stride)
  (if (> low high)
    '()
    (cons low (sequence
                (+ low stride)
                high stride))))
```

Got cond

```
(defun list-nth-mod (xs n)
  (cond [(< n 0) #(error "list-nth-mod: negative number")
        [(=:= xs '()) #(error "list-nth-mod: empty list")
        ['true (car (list-tail xs (rem n (length xs))))])
```

Return a stream for delayed computation

```
(defun cycle-lists (xs ys)
  (fletrec ((stream (lst1 n1 lst2 n2)
                    (cons
                     (cons (list-nth-mod lst1 n1)
                           (list-nth-mod lst2 n2))
                     (lambda ()
                       (stream lst1 (+ n1 1)
                               lst2 (+ n2 1))))))
    (lambda () (stream xs 0 ys 0))))
```

Test Homework

```
(is-equal '((1 . a) (2 . -))
          (: homework stream-for-n-steps
            (: homework cycle-lists '(1 2 3 4) '(a - c))
(is-equal '((1 . a) (2 . -) (3 . c) (4 . a))
          (: homework stream-for-n-steps
            (: homework cycle-lists '(1 2 3 4) '(a - c))
(is-equal '((1 . a) (2 . b) (3 . a) (1 . b))
          (: homework stream-for-n-steps
            (: homework cycle-lists '(1 2 3) '(a b)) 4)))
```

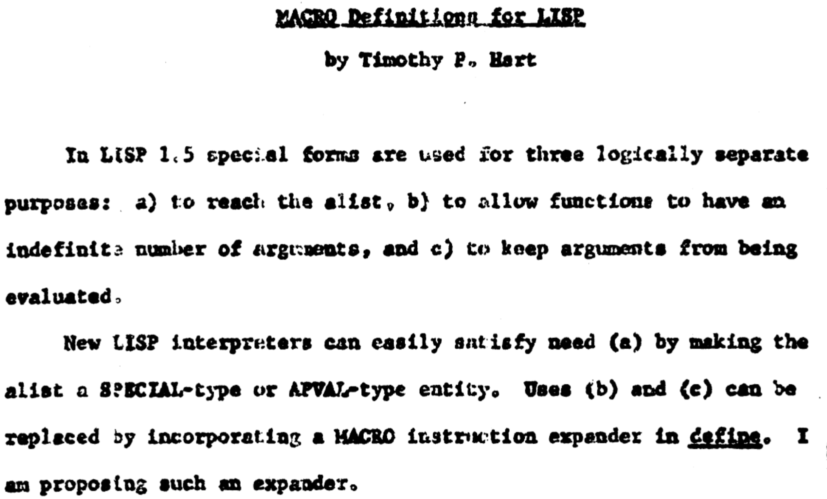
Macrology[18]

17

- hard to keep DRY
- Boilerplate
- Code Generation
- DSLs (Domain Specific Languages)
- DSP (Domain Specific Programming)

[17] An Introduction to Lisp Macros [David Nolen] `http://bit.ly/1KldAXx`
[18] [Fogus](`http://bit.ly/1cR1uXN`)

Introduction for Lisp - 1963[20]



MACRO Definitions for LISP

by Timothy P. Hart

In LISP 1.5 special forms are used for three logically separate purposes: a) to reach the alist, b) to allow functions to have an indefinite number of arguments, and c) to keep arguments from being evaluated.

New LISP interpreters can easily satisfy need (a) by making the alist a SPECIAL-type or APVAL-type entity. Uses (b) and (c) can be replaced by incorporating a MACRO instruction expander in define. I am proposing such an expander.

19

[19]Macro Definition for LISP [Hart] http://bit.ly/1AntnBE
[20]The Evolution of Lisp [Steele, Gabriel] http://bit.ly/1K5KVlX

Syntax Macros[21]

- Lisp adjunct to compiler
- Unlike simple token substitution macros such in CPP (the C preprocessor).
- Syntax Macros (like those in Lisp) operate on Abstract Syntax Trees (ASTs) and operate during parsing.
- Macros produce ASTs the replace the code of the macro invocation in downstream compiler operations and declare the type of AST they return.

[21]Programmable Sytax Macros [Weise Crew] `http://bit.ly/1EZo8Uv`

Backquote Macro

[22]

- ‘- switch to template mode
- ’- protect symbols
- , - unquote/substitute
- ,@ - unquote splice - ‘(tuple 4 5 6 ,@a) => (tuple 4 5 6 1 2 3)

match-lambda - pattern match over lambdas

```
;;; quick destructure - lfefriday
(defun destruct ()
  (lists:append
   (lists:map
    (match-lambda ((`#(,item ,count))
                   (lists:duplicate count item)))
    '(#(a 1) #(b 2) #(C 3) #(_d_ 4)))))
```

[22] Clojure's Backtick [Brandon Bloom] http://bit.ly/1BdslTT

Thread Macros[23]

```
> (->> '(1 2 3 4 5) cdr (lists:map (lambda (x) (+ x 1))))
(3 4 5 6)
```

[23] Akin to clj-threading http://bit.ly/1LdvkkA

```
(define-syntax ->
  (syntax-rules
    ([x]
     x)
    ([x (s ss ...)]
     (s x ss ...))
    ([x y]
     (y x))
    ([x y z ...]
     (-> (-> x y) z ...))))

(define-syntax ->>
  (syntax-rules
    ([x]
     x)
    ([x (ss ...)]
     (ss ... x))
    ([x y]
     (y x))
    ([x y z ...]
     (->> (->> x y) z ...))))
```

```
(deftest single-thread
  (is-equal 'x (-> 'x))
  (is-equal (list 'x) (-> 'x (list)))
  (is-equal (list 'x 'y) (-> 'x (list 'y)))
  (is-equal (list 'x 'y 'z) (-> 'x (list 'y 'z)))
  (is-equal 'z (-> '(x z y) cdr car))
  (is-equal (-> 1 (- 2 3)) -4))

(deftest double-thread
  (is-equal 'x (->> 'x))
  (is-equal (list 'x) (->> 'x (list)))
  (is-equal (list 'y 'x) (->> 'x (list 'y)))
  (is-equal (list 'y 'z 'x) (->> 'x (list 'y 'z)))
  (is-equal 'y (->> '(x y z) cdr car))
  (is-equal (->> 1 (- 2 3)) -2))
```

Expansion

```
> (macroexpand-all '(-> 0 (+ 1) (+ 2) (+ 3)
                         (cons '())) $ENV)

(cons (call 'erlang '+ (call 'erlang '+
        (call 'erlang '+ 0 1) 2) 3) '())
-> (6)

> (macroexpand-all '(->> 0 (+ 1) (+ 2) (+ 3)
                         (cons '())) $ENV)

(cons '() (call 'erlang '+ 3 (call 'erlang '+ 2
             (call 'erlang '+ 1 0))))
-> (() . 6)
```

24

[24] Clojure macroexpand example http://bit.ly/1Le5OM7

UnHygienic[26]

Programming languages with hygienic macros automatically rename variables to prevent subtle but common bugs arising from unintentional variable capture— the experience of the practical programmer is that hygienic macros "just work."[25]

- hygiene prevent collisions of symbol definitions
- gensym - symbol w/ unique name - would help
- lfe macros - own evaluation semantics

[25] A Theory of Hygienic Macros [Herman, Wand] `http://bit.ly/1EleRFz`
[26] `http://bit.ly/1KlKuXZ`

LFE's Intermediate Representation (IR) -> Core Erlang

[27]

- IR for common interface
- Erlang does it too!
- Elixir has a straight to Beam compiler target
- Core Erlang scopes nested as-in ordinary lambda calculus, unlike Erlang's scoping rules
- Transitions from Core Erlang to code for the register-based BEAM VM

```
c("some_props.erl", to_core).
%% from core to register-based BEAM
c("some_props.erl", 'S'). %% disassembled BEAM code
```

[27] A Peek Inside the Erlang Compiler `http://bit.ly/1BdXpTr`

Expressions[28]

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{var} \;\mid\; \mathit{fname} \;\mid\; \mathit{lit} \;\mid\; \mathit{fun} \\
 & \mid & \mathtt{[}\, \mathit{exprs}_1 \;\mathtt{|}\; \mathit{exprs}_2 \,\mathtt{]} \\
 & \mid & \mathtt{\{}\, \mathit{exprs}_1 \mathtt{,}\, \ldots \mathtt{,}\, \mathit{exprs}_n \,\mathtt{\}} \\
 & \mid & \mathtt{let}\ \mathit{vars}\ \mathtt{=}\ \mathit{exprs}_1\ \mathtt{in}\ \mathit{exprs}_2 \\
 & \mid & \mathtt{do}\ \mathit{exprs}_1\ \mathit{exprs}_2 \\
 & \mid & \mathtt{letrec}\ \mathit{fname}_1\ \mathtt{=}\ \mathit{fun}_1 \\
 & & \cdots \mathit{fname}_n\ \mathtt{=}\ \mathit{fun}_n\ \mathtt{in}\ \mathit{exprs} \\
 & \mid & \mathtt{apply}\ \mathit{exprs}_0\mathtt{(}\mathit{exprs}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathit{exprs}_n\mathtt{)} \\
 & \mid & \mathtt{call}\ \mathit{exprs}_{n+1}\mathtt{:}\mathit{exprs}_{n+2}\mathtt{(}\mathit{exprs}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathit{exprs}_n\mathtt{)} \\
 & \mid & \mathtt{primop}\ \mathit{Atom}\mathtt{(}\mathit{exprs}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathit{exprs}_n\mathtt{)} \\
 & \mid & \mathtt{try}\ \mathit{exprs}_1\ \mathtt{catch}\ \mathtt{(}\mathit{var}_1\mathtt{,}\ \mathit{var}_2\mathtt{)}\ \mathtt{->}\ \mathit{exprs}_2 \\
 & \mid & \mathtt{case}\ \mathit{exprs}\ \mathtt{of}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \mathtt{end} \\
 & \mid & \mathtt{receive}\ \mathit{clause}_1 \cdots\ \mathit{clause}_n \\
 & & \quad \mathtt{after}\ \mathit{exprs}_1\ \mathtt{->}\ \mathit{exprs}_2 \\
\mathit{vars} & ::= & \mathit{var} \;\mid\; \mathtt{<}\mathit{var}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathit{var}_n\mathtt{>}
\end{array}
$$

[28] An introduction to Core Erlang [Carlsson] `http://bit.ly/1ElLfbj`

Eg. Erlang

```erlang
f(X) ->
    case X of
        {foo, A} -> B = g(A);
        {bar, A} -> B = h(A)
    end,
    {A, B}.
```

Eg. To Core Erlang

```
'f'/1 =
    %% Line 15
    fun (_cor0) ->
    let <_cor5,A,B> =
        %% Line 16
        case _cor0 of
          %% Line 17
          <{'foo',A}> when 'true' ->
          let <B> =
              apply 'g'/1
              (A)
          in  <B,A,B>
          %% Line 18
          <{'bar',A}> when 'true' ->
          let <B> =
              apply 'h'/1
              (A)
          in  <B,A,B>
          ( <_cor3> when 'true' ->
            primop 'match_fail'
            ({'case_clause',_cor3})
        -| ['compiler_generated'] )
        end
    in  %% Line 20
        {A,B}
```

LL(1) Parser Generator

- (Spell to finish the generator, currently handwritten generator)[29]
- LL(1) parser
  - Top-Down (predictive parser)
  - [L scan the input from l_r, L create leftmost derivation 1 *1 input symbol of lookahead*]
  - First and Follow Sets for Products
  - Uses stack to store productions it must return[30]

[29]Spell [Virding] `http://bit.ly/1Fr05kI`

[30]TopDown Parsing Stanford Handout [Johnson] `http://stanford.io/1AnY7Cq`

lfe LL(1) "tedious" table (missing some cols for presentation) [31]

	(	)	[	]	.	'`,,@	#(#B(#M(
f	f->s		f->s			f->s	f->s
s	s->(l)		s-> [s]			s->' s	s->(p)
l	l->s t	l->e	l->s t	l->e		l->s t	l->s t
t	t->s t	t->e	t->s t	t->e	t->. s	t->s t	t->s t
p	p->s p	p->	p->s p	p->		p->s p	p->s p

[31] lfe_parse.erl `https://bitly.com/shorten/`

LL(1) "Simpler" Example [32]

Grammar:

$$S \rightarrow (S)/\epsilon$$

- Can generate all nested balanced parenthesis (()) $.
- terminals are leaf notes in a parse tree, cannot be broken down further... e.g. a char or digit in some cases
- nonterminals are non-leaf nodes in the parse tree
- Table of Productions...

	(	)	$ (bottom of stack, special terminal)
S	S->(S)	S->ϵ	S->ϵ

- Hit epsilon (push nothing to the stack, leave loop)

[32] Video `http://bit.ly/1ejqEzj`

The Goings-on

- lfe stdlib, dialyzer-dev branch, success-typing[33]
- lfetool[34]
- JVM options - Erjang/jife | lfe/OTP starts-up Clojure (multinode)[35]
- There's always Joxa (Lisp-1, In-System Macros)[36]
- Elixir's :+1: -> And hygienic macros (late resolution), protocols[37]

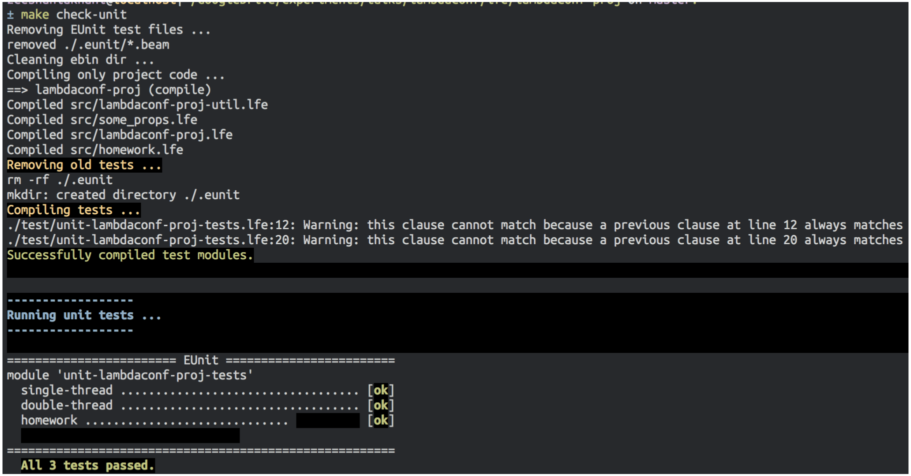
[33]gh: `http://bit.ly/1JBkZjZ`, Practical Type Inference Based on Success Typings `http://bit.ly/1JBl1bM`

[34]lfetool - lfe project template `http://bit.ly/1KmHfiR`

[35][McGreggor] `http://bit.ly/1EYTmLt`

[36][@bltroutwine] `http://bit.ly/1FxJJrR`, [@ericbmerritt] `http://bit.ly/1JBjmmC`

[37]Elixir macros-hygiene `http://bit.ly/1FxX975`



```
± make check-unit
Removing EUnit test files ...
removed ./.eunit/*.beam
Cleaning ebin dir ...
Compiling only project code ...
==> lambdaconf-proj (compile)
Compiled src/lambdaconf-proj-util.lfe
Compiled src/some_props.lfe
Compiled src/lambdaconf-proj.lfe
Compiled src/homework.lfe
Removing old tests ...
rm -rf ./.eunit
mkdir: created directory ./.eunit
Compiling tests ...
./test/unit-lambdaconf-proj-tests.lfe:12: Warning: this clause cannot match because a previous clause at line 12 always matches
./test/unit-lambdaconf-proj-tests.lfe:20: Warning: this clause cannot match because a previous clause at line 20 always matches
Successfully compiled test modules.


------------------
Running unit tests ...
------------------

======================== EUnit ========================
module 'unit-lambdaconf-proj-tests'
  single-thread .................................. [ok]
  double-thread .................................. [ok]
  homework ............................            [ok]

=======================================================
  All 3 tests passed.
```

More lfe

- Duncan McGreggor
- lfe Friday
- @ErlangLisp on Twitter
- #erlang-lisp on Freenode IRC
- lisp-flavoured-erlang@googlegroups.com
- `http://lfe.github.io/`