LFE (programming language)

Lisp Flavored Erlang (**LFE**) is a functional, concurrent, garbage collected, general-purpose programming language and Lisp dialect built on Core Erlang and the Erlang virtual machine (BEAM). LFE builds on Erlang to provide a Lisp syntax for writing distributed, fault-tolerant, soft real-time, non-stop applications. LFE also extends Erlang to support metaprogramming with Lisp macros and an improved developer experience with a feature-rich read–eval–print loop (REPL).[1] LFE is actively supported on all recent releases of Erlang; the oldest version of Erlang supported is R14.

Contents

LFE

Paradigm	Multi-paradigm: concurrent, functional
Family	Erlang, Lisp
Designed by	Robert Virding
Developer	Robert Virding
First appeared	2008
Stable release	1.3 / 4 June 2017
Typing discipline	dynamic, strong
Implementation language	Erlang
Platform	IA-32, x86-64
OS	Cross-platform
License	Apache 2.0
Filename extensions	.lfe .hrl
Website	lfe.io (http://lfe.io)
Influenced by	
Erlang, Common Lisp, Maclisp, Scheme, Elixir, Clojure, Hy	
Influenced	
Joxa, Concurrent Schemer	

History

Initial release

Initial work on LFE began in 2007, when Robert Virding started creating a prototype of Lisp running on Erlang.[2] This work was focused primarily on parsing and exploring what an implementation might look like. No version control system was being used at the time, so tracking exact initial dates is somewhat problematic.[2]

Virding announced the first release of LFE on the *Erlang Questions* mail list in March 2008.[3] This release of LFE was very limited: it did not handle recursive `letrec`s, `binary`s, `receive`, or `try`; it also did not support a Lisp shell.[4]

Initial development of LFE was done with version R12B-0 of Erlang[5] on a Dell XPS laptop.[4]

Motives

Robert Virding has stated that there were several reasons why he started the LFE programming language:[2]

- He had prior experience programming in Lisp.
- Given his prior experience, he was interested in implementing his own Lisp.
- In particular, he wanted to implement a Lisp in Erlang: not only was he curious to see how it would run on and integrate with Erlang, he wanted to see what it would *look* like.
- Since helping to create the Erlang programming language, he had had the goal of making a Lisp which was specifically designed to run on the BEAM and able to fully interact with Erlang/OTP.
- He wanted to experiment with compiling another language on Erlang. As such, he saw LFE as a means to explore this by generating Core Erlang and plugging it into the backend of the Erlang compiler.

Features

- A language targeting Erlang virtual machine (BEAM)
- Seamless Erlang integration: zero-penalty Erlang function calls (and vice versa)
- Metaprogramming via Lisp macros and the homoiconicity of a Lisp
- Common Lisp-style documentation via both source code comments and docstrings
- Shared-nothing architecture concurrent programming via message passing (Actor model)
- Emphasis on recursion and higher-order functions instead of side-effect-based looping
- A full read–eval–print loop (REPL) for interactive development and testing (unlike Erlang's shell, the LFE REPL supports function and macro definitions)
- Pattern matching
- Hot loading of code
- A Lisp-2 separation of namespaces for variables and functions
- Java inter-operation via JInterface and Erjang
- Scripting abilities with both `lfe` and `lfescript`

Syntax and semantics

Symbolic expressions (S-expressions)

Like Lisp, LFE is an expression-oriented language. Unlike non-homoiconic programming languages, Lisps make no or little syntactic distinction between *expressions* and *statements*: all code and data are written as expressions. LFE brought homoiconicity to the Erlang VM.

Lists

In LFE, the list data type is written with its elements separated by whitespace, and surrounded by parentheses. For example, `(list 1 2 'foo)` is a list whose elements are the integers `1` and `2`, and the atom `foo`. These values are implicitly typed: they are respectively two integers and a Lisp-specific data type called a *symbolic atom*, and need not be declared as such.

As seen in the example above, LFE expressions are written as lists, using prefix notation. The first element in the list is the name of a *form*, i.e., a function, operator, or macro. The remainder of the list are the arguments.

Operators

The LFE-Erlang operators are used in the same way. The expression

```
(* (+ 1 2 3 4 5 6) 2)
```

evaluates to 42. Unlike functions in Erlang and LFE, arithmetic operators in Lisp are variadic (or *n-ary*), able to take any number of arguments.

Lambda expressions and function definition

LFE has *lambda*, just like Common Lisp. It also, however, has *lambda-match* to account for Erlang's pattern-matching abilities in anonymous function calls.

Erlang idioms in LFE

This section does not represent a complete comparison between Erlang and LFE, but should give a taste.

Pattern matching

Erlang:

```
1> {Len,Status,Msg} = {8,ok,"Trillian"}.
{8,ok,"Trillian"}
2> Msg.
"Trillian"
```

LFE:

```
> (set (tuple len status msg) #(8 ok "Trillian"))
#(8 ok "Trillian")
> msg
"Trillian"
```

List comprehensions

Erlang:

```
1> [trunc(math:pow(3,X)) || X <- [0,1,2,3]].
[1,3,9,27]
```

LFE:

```
> (list-comp
    ((<- x '(0 1 2 3)))
    (trunc (math:pow 3 x)))
(1 3 9 27)
```

Or idiomatic functional style:

```
> (lists:map
    (lambda (x) (trunc (math:pow 3 x)))
    '(0 1 2 3))
(1 3 9 27)
```

Guards

Erlang:

```
right_number(X) when X == 42; X == 276709 ->
true;
right_number(_) ->
false.
```

LFE:

```
(defun right-number?
  ((x) (when (orelse (== x 42) (== x 276709)))
    'true)
  ((_) 'false))
```

cons'ing in function heads

Erlang:

```
sum(L) -> sum(L,0).
sum([], Total) -> Total;
sum([H|T], Total) -> sum(T, H+Total).
```

LFE:

```
(defun sum (l) (sum l 0))
(defun sum
  (('() total) total)
  (((cons h t) total) (sum t (+ h total))))
```

or using a ``cons`` literal instead of the constructor form:

```
      (defun sum (l) (sum l 0))
      (defun sum
        ((`() total) total)
        ((`(,h . ,t) total) (sum t (+ h total))))
```

Matching records in function heads

Erlang:

```
handle_info(ping, #state {remote_pid = undefined} = State) ->
    gen_server:cast(self(), ping),
    {noreply, State};
handle_info(ping, State) ->
    {noreply, State};
```

LFE:

```
(defun handle_info
  (('ping (= (match-state remote-pid 'undefined) state))
    (gen_server:cast (self) 'ping)
    `#(noreply ,state))
  (('ping state)
    `#(noreply ,state)))
```

Receiving messages

Erlang:

```
      universal_server() ->
          receive
              {become, Func} ->
                  Func()
          end.
```

LFE:

```
      (defun universal-server ()
        (receive
          ((tuple 'become func)
           (funcall func))))
```

or:

```
      (defun universal-server ()
        (receive
          (`#(become ,func)
            (funcall func))))
```

Examples

Erlang interoperability

Calls to Erlang functions take the form *(<module>:<function> <arg1> ... <argn>)*:

```
(io:format "Hello, World!")
```

Functional paradigm

Using recursion to define the Ackermann function:

```
(defun ackermann
  ((0 n) (+ n 1))
  ((m 0) (ackermann (- m 1) 1))
  ((m n) (ackermann (- m 1) (ackermann m (- n 1)))))
```

Composing functions:

```
(defun compose (f g)
  (lambda (x)
   (funcall f
     (funcall g x))))

(defun check ()
  (let* ((sin-asin (compose #'sin/1 #'asin/1))
         (expected (sin (asin 0.5)))
         (compose-result (funcall sin-asin 0.5)))
    (io:format "Expected answer: ~p~n" (list expected))
    (io:format "Answer with compose: ~p~n" (list compose-result))))
```

Concurrency

Message-passing with Erlang's light-weight "processes":

```
(defmodule messenger-back
 (export (print-result 0) (send-message 2)))

(defun print-result ()
  (receive
    ((tuple pid msg)
      (io:format "Received message: '~s'~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 'messenger-back 'print-result ())))
    (! spawned-pid (tuple calling-pid msg))))
```

Multiple simultaneous HTTP requests:

```
(defun parse-args (flag)
  "Given one or more command-line arguments, extract the passed values.

  For example, if the following was passed via the command line:

    $ erl -my-flag my-value-1 -my-flag my-value-2

  One could then extract it in an LFE program by calling this function:

    (let ((args (parse-args 'my-flag)))
      ...
      )
  In this example, the value assigned to the arg variable would be a list
  containing the values my-value-1 and my-value-2."
  (let ((`#(ok ,data) (init:get_argument flag)))
    (lists:merge data)))

(defun get-pages ()
```

```
  "With no argument, assume 'url parameter was passed via command line."
  (let ((urls (parse-args 'url)))
    (get-pages urls)))

(defun get-pages (urls)
  "Start inets and make (potentially many) HTTP requests."
  (inets:start)
  (plists:map
    (lambda (x)
      (get-page x)) urls))

(defun get-page (url)
  "Make a single HTTP request."
  (let* ((method 'get)
         (headers '())
         (request-data `#(,url ,headers))
         (http-options ())
         (request-options '(#(sync false))))
    (httpc:request method request-data http-options request-options)
    (receive
      (`#(http #(,request-id #(error ,reason)))
       (io:format "Error: ~p~n" `(,reason)))
      (`#(http #(,request-id ,result))
       (io:format "Result: ~p~n" `(,result))))))
```

References

1. Virding, Robert. "Lisp Flavored Erlang" (http://www.erlang-factory.com/upload/presentations/61/Robertvirding-LispFlavouredErlang.pdf) (PDF). *Erlang Factory*. Retrieved 2014-01-17.
2. "LFE History on the Lisp Flavored Erlang mail list" (https://groups.google.com/d/msg/lisp-flavoured-erlang/XA5HeLbQQDk/TUHabZCHXB0J). Retrieved 2014-05-28.
3. "LFE announcement on Erlang Questions mail list" (https://groups.google.com/d/msg/erlang-questions/rf43UofUvvQ/2-L1AecnkUoJ). Retrieved 2014-01-17.
4. Armstrong, Joe; Virding, Robert (2013-12-30). "Hardware used in the development of Erlang and LFE" (http://www.erlang.org/course/history.html) (Email exchange). Interviewed by Duncan McGreggor. Retrieved 2014-01-17.
5. "Follow-up to LFE announcement on Erlang Questions mail list" (https://groups.google.com/d/msg/erlang-questions/rf43UofUvvQ/dthHortHwGoJ). Retrieved 2014-01-17.

External links

- Official website (http://lfe.io)
- LFE (https://github.com/rvirding/lfe) on GitHub
- LFE Quick Start (https://lfe.gitbooks.io/quick-start/content/index.html)
- LFE User Guide (https://lfe.gitbooks.io/reference-guide/)
- LFE on Rosetta Code (http://rosettacode.org/wiki/Category:LFE)