Quick Start

1 Getting Set Up

1.1 Dependencies

First and foremost, you will need Erlang installed. On Mac OS X, this is as easy as executing `brew install erlang` or on Ubuntu `apt-get install erlang`. You can also install Erlang from the various pre-built packages provided on the official Erlang download page or from the Erlang Solutions page (supports many more package types).

If you will be using `rebar` to build LFE, you'll need to get that too.

You will need to download git or install it using your favorite package manager.

Next, you will need to download LFE itself:

```
$ git clone git://github.com/rvirding/lfe.git
```

1.2 Building

With the dependencies installed, we're now ready to build LFE.

1.2.1 Using `make`

```
$ cd ./lfe
$ erlc -o src src/lfe_scan.xrl
$ make
```

Ordinarily, only the `make` command would be necessary. However, there's currently an issue with the `Makefile` that temporarily requires that first step.

1.2.2 Using `rebar`

Alternatively, one may use `rebar` to build LFE:

```
$ cd ./lfe
$ rebar compile
```

1.3 Installing

On non-development systems, or any system where you don't want to run LFE from a git checkout, installing system-wide is the preferred way to use LFE. If you set your `$ERL_LIBS` environment variable, LFE will install there. Here's how one might do this on a Mac OS X system with Erlang installed by Homebrew:

```
$ export ERL_LIBS=/usr/local/lib/erlang/lib
$ make install
```

You can then check that everything is where you expect:

```
$ ls -1 $ERL_LIBS|grep lfe
lfe-0.7
```

Now you can use LFE from anywhere:

```
$ erl
Erlang R15B03 (erts-5.9.3.1) [source] [64-bit] [smp:8:8] [async-threads:0]
[hipe] [kernel-poll:false] [dtrace]

Eshell V5.9.3.1  (abort with ^G)
1> lfe_shell:start().
LFE Shell V5.9.3.1 (abort with ^G)
<0.33.0>
>
```

2 Executing Code

Once you've got LFE built, you want to play with it, right? Let's take a look at the REPL (interactive shell) first.

2.1 The REPL

To start the REPL, simply run the `lfe` script and tell `erl` (which is being run in the script) which additional code paths it should look for (in this case, the compiled LFE code):

```
$ ./bin/lfe -pa ./ebin
```

This assumes that you are still in the `lfe` directory where you build LFE. Running that command will dump you into the LFE REPL, looking something like this:

```
Erlang R15B03 (erts-5.9.3.1) [source] [64-bit] [smp:8:8] [async-threads:0] ...

LFE Shell V5.9.3.1 (abort with ^G)
>
```

Note that you can also start the the LFE shell manually from an existing Erlang shell (as we did earlier in this guide) or you can pass parameters to `erl` to start it up:

```
$ erl -pa ebin -s lfe_boot -noshell -noinput
```

Now try doing some basic operations:

```
> (+ 1 2 3)
6
> (cons 1 2)
(1 . 2)
> (cons (list 1 2) (list 3 4))
((1 2) 3 4)
>
```

Next, let's operate on some variables:

```
> (let ((x 123456789)) x)
123456789
> (let ((x 123456789)) (* x x))
15241578750190521
> (let ((x 123456789)) (* x x x x))
232305722798259244150093798251441
>
```

Looking good!

2.2 Running Scripts

Okay, so now that you can run things in the REPL, you want to run them as a script, yes? Here's how.

Let's create a temporary subdirectory to play in without fear of messing up our LFE directory:

```
$ mkdir tmp
$ cd tmp
```

Then, in that directory, let's create the following file and save it as `hello.lfe`:

```
(defmodule hello
  (export (start 0)))

(defun start ()
  (: io format '"Lfe says 'Hello, World!'~n"))
```

To compile that script and then run it, we can do this:

```
$ ../bin/lfec hello.lfe
$ ../bin/lfe -s hello start -s erlang halt
```

Or, if we want to use Erlang directly, we could do this:

```
$ erl -pa ../ebin -s lfe_comp file hello -s erlang halt
$ erl -pa ../ebin -s hello start -s erlang halt
```

Or, we could compile it and run it in the same command:

```
$ erl -pa ../ebin -s lfe_comp file hello -s hello start -s erlang halt
```

Note that this is the command line equivalent of the following:

```
$ erl -pa ../ebin
1> lfe_comp:file(hello).
{ok,hello}
2> hello:start().
Lfe says 'Hello, World!'
ok
3>
```

2.3 Running Scripts from the REPL

You can also use your new `hello.lfe` script in the REPL itself. There are two ways you can do this, using `slurp` or compiling the file. If you use `slurp`, all the functions are pulled into the shell namespace, and you won't have to reference the module name. Again, assuming that you are in `lfe/tmp`:

```
$ ../bin/lfe -pa ../ebin
> (slurp '"hello.lfe")
#(ok hello)
> (start)
Lfe says 'Hello, World!'
ok
>
```

If you choose to compile your module instead, you will use it like so:

```
$ ../bin/lfe -pa ../ebin
> (c '"hello")
#(module hello)
> (: hello start)
Lfe says 'Hello, World!'
ok
>
```

Note that in the second example, you need to reference the module.

For more information on the LFE shell, be sure to see the "REPL" section of the User Guide Introduction.

3 Using Libraries

3.1 OTP Modules

Taking advantage of the Erlang stdlib is straightforward. All you need to do is prepend the call with a `:` and adjust to use the LFE syntax vs. the Erlang syntax.

In fact, we've already seen an example of this in Section 2 when we called `(: io format ...)` as part of a "Hello World."

Here's an example `base64` usage from the Erlang `stdlib`:

```
> (: base64 encode_to_string '"Space is big. Really big.")
"U3BhY2UgaXMgYmlnLiBSZWFsbHkgYmlnLg=="
> (: base64 decode_to_string '"QW5kIHNvIHRoZSBVbml2ZXJzZSBlbmRlZC4=")
"And so the Universe ended."
```

3.2 Processes in LFE

One of the first things that people want to do with LFE is examine the message passing syntax so that they can compare it with vanilla Erlang. Here's a small example of what this looks like in LFE:

```
(defun print-result ()
  (receive
```

```
    (msg
      (: io format '"Received message: '~s'~n" (list msg))
      (print-result))))
```

If that was saved in a module called `messenger`, then one could utilize it thusly:

```
> (set pid (spawn 'messenger 'print-result ()))
<0.34.0>
> (! pid '"Ford is missing.")
"Ford is missing."
Received message: 'Ford is missing.'
```

For more information on working with processes in LFE, be sure to view the tutorial.

Related to this, you can find details and discussion around OTP and creating your own servers in the OTP Servers tutorial.

3.3 Third-Party Libraries

Finally, accessing code that is written in third-party libraries is exactly the same. Simply use the modules they have provided. If you started the LFE REPL pointing to your third-party libraries with a `-pa` (path) option, then all you have to do is this:

```
> (: your-module some-function '"some parameter")
```

That's it!

4 Next Steps

This has been a quick overview of what LFE has to offer, so you might enjoy reading the User Guide next. You can see all our docs at a glance by visiting the Docs page.

If at any time you would like to provide feedback about the documentation on this site, feel free to create an issue. Be sure to give a full description so that we can best help you!