1 Introduction

1.1 Lisp Flavored Erlang

1.1.1 About

Lisp Flavored Erlang or LFE is a Lisp syntax front-end to the Erlang compiler. LFE is a Lisp-2, like Common Lisp, and comes with a REPL (shell). LFE coexists seamlessly with vanilla Erlang and OTP. As such, code written in LFE can freely be used together with modules written in vanilla Erlang and applications in Erlang/OTP.

1.1.2 Background

This work started life as a beautification of what Robert Virding had already created when he originally documented LFE. There are a few exemplar open source projects which have produced extraordinary documentation: both highly informative as well as being exceedingly easy on the eyes. We wanted LFE to aspire to those standards. In addition to helping with project adoption, creating an attractive and well-documented online resource for LFE makes it much nicer for the folks who already use the project.

The Github Pages feature provided us a means whereby an appealing open source project site could be created easily. These efforts were rewarded almost immediately by visitors and users who began spreading the word, further catalyzing our commitment to producing an improved user experience.

While attempting to add more verbose descriptions and enhance the prose around the original docs, there arose a strong desire to improve the organization of the topics covered as well. In this effort, we turned to the excellent Erlang books that have been published to date, and began drawing inspiration from these. It soon became clear that what was really needed was an LFE version of some combination of those wonderful efforts. With that, the LFE User Guide was fully set upon its course.

1.1.3 Motivation for the Uninitiated

If you have ever found yourself greatly admiring the Erlang language but thirsting for an alternative to the standard syntax, and you do not fear the elegance of parentheses (for a more civilized age), you might want to spend some time writing code in LFE. It could be just what you're looking for.

LFE has borrowings from Common Lisp and Scheme, so should provide a familiar face for those who have spent time hacking on projects powered by SBCL, Allegro, LispWorks, Chicken Scheme, Gambit, or Racket.

Similarly, those who have come to Lisp via the Java VM-powered Clojure will find much to love in the Erlang VM-powered LFE. LFE was released just one year after Clojure, but has 100% compatibility with the features in Erlang that inspired Clojure, some of which the Clojure community is still working on. You can get those without waiting when you use

LFE!

1.2 Getting Started

The user guide assumes the following background knowledge:

- basic familiarity with Lisp or Lisp dialects
- a passing knowledge of Erlang
- a working installation of Erlang and LFE

For those that would like additional information on any of these, we recommend the resources below.

Online:

- The LFE Quick Start
- Learn You Some Erlang for Great Good (.mobi)
- Practical Common Lisp
- On Lisp
- Structure and Interpretation of Computer Programs

Books:

- Programming Erlang: Software for a Concurrent World
- Erlang Programming: A Concurrent Approach to Software Development
- Introducing Erlang
- Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp

The LFE Quick Start is an important resource, as it covers dependencies, building LFE, installation, using the REPL, running scripts, and using modules/libraries (OTP and third-party).

1.3 More About LFE

1.3.1 What LFE Isn't

Just to clear the air and set some expectations, here's what you're *not* going to find in LFE:

- An implementation of Scheme
- An implementation of Common Lisp
- An implementation of Clojure

As such, you will not find the following:

- A Scheme-like single namespace
- CL packages or munged names faking packages
- Access to Java libraries

1.3.2 What LFE Is!

Here's what you *can* expect of LFE:

- A proper Lisp-2, based on the features and limitations of the Erlang VM
- Compatibility with vanilla Erlang and OTP
- It runs on the standard Erlang VM

Furthermore, as a result of Erlang's influence (and LFE's compatibility with it), the following hold:

- there is no global data
- data is not mutable
- only the standard Erlang data types are used
- you get pattern matching and guards
- you have access to Erlang functions and modules
- LFE has a compiler/interpreter
- functions with declared arity and fixed number of arguments
- Lisp macros

1.4 What to Expect from this Guide

The intent of this guide is to follow the same general pattern that the best Erlang books do, covering the topics listed in the User Guide table of contents from an LFE perspective.

Some of the Guide's sections will be covered in dedicated tutorials or other in-depth documents; in those cases, we provide links to that material. If your favorite topic is not covered above, let us know! We'll try to find a place for it :-)

1.5 The LFE REPL

1.5.1 Using the REPL

We covered basic REPL usage in the quick start. That's the best place to go for an introduction to using the LFE REPL. Regardless (and for your convenience), we also provide some information about the REPL in the document you are currently reading :-)

1.5.1.1 Starting the REPL

If you *don't* have LFE installed system-wide, you need to tell it (Erlang, really) where the LFE `.beam` files are. Here are the three ways to start up LFE in this case:

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

or:

```
$ erl -user lfe_boot -pa /path/to/lfe/ebin
```

or:

```
$ erl -pa /path/to/lfe/ebin
```

followed by this from the Erlang shell:

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

>
```

If you do have LFE installed system-wide, then starting the shell can be done in the ways listed below.

Using the `lfe` command. Be sure to change directory to where you have saved (or cloned) the LFE source code. Then:

```
$ ./bin/lfe
```

You can also start the LFE REPL by passing options directly to `erl`. Again, assuming that you have LFE installed system-wide, from any directory you may do this:

```
$ erl -user lfe_boot
```

Also, if you happen to be running an Erlang shell already, you can start the LFE REPL with the following:

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

>
```

1.5.1.2 Running Commands

Once you're in the REPL, it's just a matter of entering code:

```
> (+ 1.5 3 4 5 6 7)
28
>
```

Note that you can't define modules, macros, functions or records from the REPL; you'll have to put those in a module file and compile or `slurp` the file from the REPL. You can, however, use `lambda` from the REPL:

```
> (set exp
    (lambda (x y)
      (trunc (: math pow x y))))
#Fun<lfe_eval.15.53503600>
>
```

Then, using the `lambda` you have just defined is as easy as this:

```
> (funcall exp 2 6)
64
>
```

Or, if you want to get nuts:

```
> (: lists map
    (lambda (z)
      (funcall exp (car z) (cadr z)))
    (list (list 1.5) (list 3 4) (list 5 6)))
(1 81 15625)
>
```

1.5.1.3 Quitting the REPL

Just as there are multiple ways in which you can start the REPL, there are a couple ways you can leave it. You can jump into the JCL from the LFE prompt by hitting ^g and then entering q:

```
> ^g
User switch command
 --> q
$
```

or you can call the Erlang shell's quit function:

```
> (: c q)
ok
>
$
```

1.5.2 Special Functions

There are some functions specially defined in LFE for use from the REPL. These are listed below with information about their use.

- `(c File [Options])` - Compile and load an LFE file. Assumes default extension `.lfe`.
- `(l Module ...)` - Load modules.
- `(m Module ...)` - Print out module information, if no modules are given then print information about all modules.
- `(ec File [Options])` - Compile and load an Erlang file.
- `(slurp File)` - Slurp in a source LFE file and makes all functions and macros defined in the file available in the shell. Only one file can be slurped at a time and slurping a new file removes all data about the previous one.
- `(unslurp)` - Remove all function and macro definitions except the default ones.
- `(set Pattern Expr)` - Evaluate Expr and match the result with Pattern binding variables in it. These variables can then be used in the shell and also rebound in another set.
- `(: c Command Arg ...)` - All the commands in the Erlang shell's Command Interface Module can be reached in this way.

1.5.3 Special Variables

LFE also provides some convenience variables similar to what Lisp has defined for +, ++, +++, *, **, ***, and -. Additionally, LFE also provides the `$ENV` variable.

- `+/++/+++` - The three previous expressions input.
- `*/**/***` - The values of the previous 3 expressions.
- `-` - The current expression input.
- `$ENV` - The current environment (accessible in the REPL as well as in macros).

These probably warrant some examples.

Let's say you had just entered the following in the REPL:

```
> (+ 1.5)
3
> (: c memory)
(#(total 10026672)
 #(processes 1656528)
 #(processes_used 1656528)
 #(system 8370144)
 #(atom 153321)
 #(atom_used 147399)
 #(binary 1338560)
 #(code 3255239)
 #(ets 290544))
> (set my-func (lambda () (: io format '"Hello, Zaphod!")))
#Fun<lfe_eval.21.53503600>
>
```

Then you can get the previous expressions you input with the following commands:

```
> +++
(+ 1.5)
> +++
(: c memory)
> +++
(set my-func (lambda () (: io format '"Hello, Zaphod!")))
> ++
+++
> +
++
>
```

Most of us will actually use the arrow keys, thanks to the `readline` library. However, the classic, pre-`readline` approach is still available, should you choose to use it.

Similarly, you can get the results returned by using the variabels from the second bullet item. If you're following along in the REPL, go ahead and re-enter the commands we typed above to reset the last three items in your command history. Then do the following:

```
> ***
3
> ***
(#(total 9976496)
 #(processes 1606688)
```

```
 #(processes_used 1606688)
 #(system 8369808)
 #(atom 153321)
 #(atom_used 147399)
 #(binary 1338096)
 #(code 3255239)
 #(ets 290544))
> ***
#Fun<lfe_eval.21.53503600>
> (funcall *)
Hello, Zaphod!
ok
>
```

There's another, called the "dash" varibale. It is bound to the actual expression that is currently being evaluated. Here's an example of this being used:

```
> (: io format '"Evaluating the expression '~p' ...~n" (list -))
Evaluating the expression '[':',io,format,
                            [quote,"Evaluating the expression '~p' ...~n"],
                            [list,'-']]' ...
ok
>
```

We've saved one of the more archane special variables to last: `$ENV`. When you first start up a shell, the `$ENV` variable holds pristine state data:

```
> $ENV
(#(variable *** ())
 #(variable ** ())
 #(variable * ())
 #(variable - ())
 #(variable +++ ())
 #(variable ++ ())
 #(variable + ()))
>
```

We can define a few variables and then check them out with another display of the environment:

```
> $ENV
(#(variable my-func #Fun<lfe_eval.10.53503600>)
 #(variable asnwer 42)
 #(variable *** 42)
 #(variable
...
```

If you `slurp` a file in the REPL, your environment will be updated with all the definitions in that file:

```
> (slurp '"examples/core-macros.lfe")
#(ok -no-mod-)
> $ENV
(#(function
   bq-expand
   2
   #(letrec
     (lambda (exp n)
```

```
...
```

There is, as you might have guessed, much more to that ellided output (for that particular example, nearly all the rest of it is macro definitions).

Making use of `$ENV` can be very helpful when debugging include files, loading Erlang header files, or when creating macros. Furthermore, when spending a great deal of time in the REPL prototyping code for a project, it can be quite useful to refresh one's memory as to what functions and variables are currently available in `$ENV`.

Looking at the output for `$ENV` can be a bit overwhelming, however. As you might imagine, there is an easy answer to this: filter it! The following makes use of the Erlang `lists` module as well as patterns in an anonymous function, both of which will be covered in more detail later:

```
> (set filter-env
    (lambda (env)
      (: lists foreach
        (match-lambda
          (((tuple 'function func-name arity _))
           (: io format '"function: ~p/~p~n" (list func-name arity)))
          (((tuple 'macro macro-name _))
           (: io format '"macro: ~p~n" (list macro-name)))
          (((tuple 'variable var-name value))
           (: io format '"variable: ~p~n" (list var-name)))
          ((_)))
        env)))
#Fun<lfe_eval.21.53503600>
```

Now, as one hacks away in the REPL, `slurp`ing away at various modules, getting a list of what's defined in the current environment is a piece of cake:

```
> (funcall filter-env $ENV)
variable: 'my-var-4'
variable: 'my-var-3'
variable: 'my-var-2'
variable: 'my-var-1'
variable: filter
function: 'bq-expand'/2
macro: backquote
macro: 'orelse'
macro: 'andalso'
macro: 'cond'
macro: 'flet*'
macro: 'let*'
macro: 'list*'
macro: '?'
macro: ':'
macro: '++'
macro: cddr
macro: cdar
macro: cadr
macro: caar
variable: '***'
variable: '**'
variable: '*'
variable: '-'
```

```
variable: '+++'
variable: '++'
variable: '+'
ok
>
```

1.5.4 Getting Out of Trouble

Every once in a while you may find that you do something which causes the REPL to crash, presenting you with something that looks like this:

```
>
  =ERROR REPORT==== 17-Feb-2013::15:39:33 ===
  ...
```

You don’t have to quit and restart the REPL, if you don’t want to! There are a couple of steps that you can take instead.

1.5.4.1 Interrupting a Shell Process

When you get an error as seen above, type `^g`. This will put you into JCL (Job Control Language) mode. At the JCL prompt, type `?` to see a list of options:

```
User switch command
 --> ?
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q        - quit erlang
  ? | h             - this message
```

Let’s see what’s running:

```
--> j
  1* {lfe_shell,start,[]}
```

Our shell process is still alive, though not responding. Let’s interrupt it and then connect to it again:

```
--> i 1
--> c 1
exception error: function_clause
 in (: lists sublist #(error interrupted) 1)
 in (lfe_scan string 4)
 in (lfe_io scan_and_parse 3)

>
```

Once we interrupted the job, our error messages were printed to the REPL and we were placed back at the LFE prompt.

1.5.4.2 Starting a New Shell

Sometimes, though, there is no shell process any more. Here's how to start up a new shell process if the one that you're using dies:

```
--> s lfe_shell
--> j
      2* {lfe_shell,start,[]}
--> c 2
LFE Shell V5.9.3.1 (abort with ^G)
>
```

And you're back up!

1.6 Loading Files

1.6.1 Loading Files in the REPL

There are several ways in which one may load files in LFE.

1.6.1.1 `slurp`

As mentioned in the section on LFE's REPL, `slurp`ing a file makes all functions and macros defined in the file available in the shell. One does not need to reference the module (and, in fact, attempting to do so will result in an error. Also, note that only one file can be slurped at a time; slurping a new one removes all data about the previous file.

```
> (slurp '"my-module.lfe")
#(ok my-module)
>
```

1.6.1.2 `c`

Compiling a module from the REPL is what you need if you wish to work with multiple modules simultaneously:

```
> (c '"my-module")
#(module my-module)
>
```

1.6.1.3 `ec`

You can also load Erlang files in LFE:

```
> (ec '"../lfe/src/lfe_macro.erl")
#(ok lfe_macro)
>
```

1.6.1.4 `l`

If a module is in your Erlang/LFE path, you can load that too:

```
> (l 'mochiweb)
```

```
(#(module mochiweb))
>
```

1.6.2 Loading Files in Modules

Code may be included wholesale into LFE modules by either using `include-file` or `include-lib`.

1.6.2.1 `include-file`

If you have records or data that you would like to be available to more than one module, you can put those in a dedicated file and pull them in to your modules. For example, let's say I had defined the following constants in the file `include/consts.lfe`:

```
(defmacro *base-cool-factor* _ `0.8)
(defmacro *earth-adjustment* _ `0.3)
```

Then, in the following two files I could easily use those constants by including them:

```
(defmodule zaphod
  (export all)

(include-file "include/consts.lfe")

(defun get-coolness ()
  (let ((zaphod-cool-factor 0.9))
    (* (*base-cool-factor*) zaphod-cool-factor)))

(defmodule arthur
  (export all)

(include-file "include/consts.lfe")

(defun get-coolness ()
  (let ((arthur-cool-factor 0.1))
    (* (*base-cool-factor*) (*earth-adjustment*) arthur-cool-factor)))
```

1.6.2.2 `include-lib`

TBD

2 Diving In

2.1 Numbers and Operators

2.1.1 Integers and Floats

Let's start with something simple :-) To follow along, fire up your LFE REPL. Numbers are simple in LFE, just like Erlang:

```
> 1
1
> 2
2
> 3
3
>
```

Of course, it might be more interesting to look at something like different bases:

```
> #b101010
42
> #o52
42
> #x2a
42
> #36r16
42
>
```

LFE supports representing binary (`#b`), octal (`#o`), decimal (`#d`), hexidecimal (`#x`), as well as aribtrary bases from 1 through 36 (`#XrY`).

With some help from calling an Erlang function, we can work the bases in reverse, too:

```
> (: erlang integer_to_list 123 2)
"1111011"
```

Note that the first argument is the number you want to convert and the second is the base you want to use (see here for more details).

2.1.2 Arithmatic Operators

But numbers by themselves aren't going to do us much good if we can't operate on them. The usual apply:

```
> (+ 1 2 3 4 5 6)
21
> (- 6 21)
15
> (/ 36 7)
5.142857142857143
> (+ #b101010 #o52 #x2a #36r16)
168
> (* 42 4)
168
> (: erlang integer_to_list (+ #b1001 #b100 #b10) 2)
"1111"
> (div 11 2)
5
> (rem 11 2)
1
> `(,(div 11 2) ,(rem 11 2))
(5 1)
>
```

2.1.3 Logical Operators

The usual suspects are used as follows:

```
> (< 1 2)
true
> (> 1 2)
false
> (>= 2 2)
true
> (=< 3 2)
false
> (>= 3 2)
true
> (== 1 1)
true
> (== 1 1.0)
true
> (/= 1 1)
false
> (/= 2 1)
true
>
```

Note the rather awkward different between “less than” and “greater than”: it’s easy to forget that the angle brackets go at different ends for each.

Then there are the operators which also check against type for exact (non-)equality:

```
> (=:= 1 1.0)
false
> (=:= 1.0 1.0)
true
> (=/= 1.0 1.0)
false
> (=/= 1 1.0)
true
>
```

2.1.4 Boolean Operators

How about some logic?

```
> (and 'true 'false)
false
> (and 'true 'true)
true
> (or 'true 'true)
true
> (or 'true 'false)
true
> (or 'false 'false)
false
> (not 'false)
true
> (not 'true)
false
```

```
> (xor 'true 'true)
false
> (xor 'false 'false)
false
> (xor 'true 'false)
true
>
```

There are also two boolean operators that you can use if you want to make a decision based on the truth value of the first term without having to compute the second term (useful if you have no need to do the second computation when the first term is false):

```
> (andalso 'false 1)
false
> (andalso 'true 1)
1
> (orelse 'true 1)
true
> (orelse 'false 1)
1
>
```

In the case of `andalso` if the first argument is `false` the second one will not be evaluated; `false` will be returned. In the case of `orelse` if the first argument is `true` then `true` will be returned without evaluating the second argument.

Contrast this to regular `or` and `and`:

```
> (and 'false 1)
exception error: badarg
  in (: erlang and false 1)

> (and 'true 1)
exception error: badarg
  in (: erlang and true 1)

> (or 'false 1)
exception error: badarg
  in (: erlang or false 1)

> (or 'true 1)
exception error: badarg
  in (: erlang or true 1)

>
```

2.1.5 Bitwise Operators

As one would expect, Erlang has the usual bitwise operators as well. Binary representation is used below for clarity of demonstration. Let's define a utility function that will save a little typing:

```
> (set dec-to-bin (lambda (x) (: erlang integer_to_list x 2)))
#Fun<lfe_eval.10.53503600>
>
```

With that defined so that we can use it, let's take a look at some of these operators:

```
> (funcall dec-to-bin (band #b10001 #b1001))
"1"
> (funcall dec-to-bin (bor #b10001 #b1001))
"11001"
> (funcall dec-to-bin (bxor #b10001 #b1001))
"11000"
> (funcall dec-to-bin (bnot #b10001))
"-10010"
> (funcall dec-to-bin (bnot (bnot #b10001)))
"10001"
> (funcall dec-to-bin (bsl #b10001 1))
"100010"
> (funcall dec-to-bin (bsr #b10001 1))
"1000"
>
```

2.2 Atoms and Strings

2.2.1 Atoms

Atoms are a data type in Erlang that is used to represent non-numerical constants. In LFE, the typographical limitations of Erlang don't apply, since they're always quoted in LFE ;-)

Atoms have a value: the same as their text:

```
> 'strag
strag
>
```

We saw this in the section on Boolean operators with the atoms of `true` and `false`. Since there are no Boolean types in Erlang or LFE, the atoms `true` and `false` are used instead.

Here are some more examples of atoms:

```
> 'Vogon
Vogon
> '_Gargle_Blaster
_Gargle_Blaster
> '+
+
> '*
*
> '|and now with hyperspace bypasses|
|and now with hyperspace bypasses|
>
```

Though very simple, atoms have a huge impact on our everyday use of Erlang and LFE, primarily in the area of pattern matching. Hold that thought, though; we're not quite ready for it yet!

Furthermore, atoms are stored differently in Erlang than strings. They take up less space and are more efficient to compare than strings.

2.2.2 Strings

Now we come to the oddball of Erlang: the string. In truth, there is no such thing. Strings in Erlang are just lists of integers:

```
> '"Don't Panic."
"Don't Panic."
> (list 68 111 110 39 116 32 80 97 110 105 99 46)
"Don't Panic."
>
```

Because Erlang (and thus LFE) strings consume 8 bytes per character on 32-bit systems and 16 bytes on 64-bit systems, they are not very efficient. As such, if you need to work with long strings in LFE, you probably want to use `(binary ...)`, but that's in the next section :-)

2.3 Binary and Bitstrings

2.3.1 Lists and `binary`

A full discussion of the binary type is a huge topic that probably deserves one or more dedicated tutorials, especially given the close connection with pattern matching and the efficient parsing of binary data. However, for now, we're just going to look at one particular area: working with strings as binary data.

In the previous section, we had mentioned using `(binary ...)` to more efficiently represent large strings. Here's an example (pretending, for now, that our example is using a very large string ;-)):

```
> (binary "There's a frood who really knows where his towel is.")
#B(84 104 101 114 101 39 115 32 97 32 102 114 111 111 100 32 119 104 111 ...)
```

Or you could use the Erlang function, if you wanted:

```
> (: erlang list_to_binary '"There's a frood who really knows...")
#B(84 104 101 114 101 39 115 32 97 32 102 114 111 111 100 32 119 104 111 ...)
101 97 108 108 121 32 107 110 111 ...)
```

Let's set a variable with this value in the shell, so we can work with it more easily:

```
(set data (binary "There's a frood who really knows where his towel is."))
#B(84 104 101 114 101 39 115 32 97 32 102 114 111 111 100 32 119 104 111 ...)
```

2.3.2 Binary Functions in OTP

Let's convert it back to a list using a function from the Erlang stdlib `binary` module:

```
> (: unicode characters_to_list data)
"There's a frood who really knows where his towel is."
```

Note that the LFE `binary` function is quite different than the call to the `binary` module in the Erlang stdlib! The `binary` module has all sorts of nifty functions we can use (check out the docs). Here's an example of splitting our data:

```
> (: binary split data (binary " who really knows "))
(#B(84 104 101 114 101 39 115 32 97 32 102 114 111 111 100)
 #B(119 104 101 114 101 32 104 105 115 32 116 111 119 101 108 32 105 115 46))
```

The `split` gives us two pieces; here's how we can get the new string from that `split`:

```
> (: unicode characters_to_list
    (: binary split data (binary "who really knows ")))
"There's a frood where his towel is."
>
```

`binary split` creates a list of binaries, but since this is an `IoList` and `unicode characters_to_list` can handle those without us having to flatten them, our work is done! We get our result: the new string that we created by splitting on `"who really knows "`.

2.3.3 Bit-Packing (and Unpacking)

For this section, let's use the 16-bit color example that is given in Joe Armstrong's Erlang book where 5 bits are allocated for the red channel, 6 for the green and 5 for the blue. In LFE, we can create a 16-bit memory area like so:

```
> (set red 2)
2
> (set green 61)
61
> (set blue 20)
20
> (binary
    (red (size 5))
    (green (size 6))
    (blue (size 5)))
#B(23 180)
>
```

All packed and ready!

We can use patterns to unpack binary data in a `let` expression into the variables `r`, `g`, and `b`, printing out the results within the `let`:

```
> (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
>
```

We're getting a little ahead of ourselves here, by throwing a pattern in the mix, but it's a good enough example to risk it :-)

2.3.4 So What's a Bitstring?

We've been looking at binaries in LFE, but what's a bitstring? The [Erlang docs] say it well: A bitstring is a sequence of zero or more bits, where the number of bits doesn't need to be divisible by 8. If the number of bits is divisible by 8, the bitstring is also a binary.

2.3.5 LFE's Exact Definition of Binary

Here's the full definition for the `binary` from in LFE:

```
(binary seg ... )
```

Where `seg` is:

```
byte
string
(val integer|float|binary|bitstring|bytes|bits
     (size n) (unit n)
     big-endian|little-endian|native-endian|little|native|big
     signed|unsigned)
```

This should help you puzzle through some of the more complex binary constructions you come accross ;-)

2.4 Variables

2.4.1 Variables in the REPL

Variables in LFE don't have the same syntactical limitations that vanilla Erlang has. Let's take a look at some examples in the REPL:

```
> (set &$% '"Mostly Harmless")
"Mostly Harmless"
> &$%
"Mostly Harmless"
```

Your variable does *not* have to start with a capital letter and not only can it contain special characters, it can *entirely consist* of them! We don't recommend this, however ;-)

Furthermore, LFE also does not share with Erlang the characteristic of not being able to change a variable once you've set it's value. In the REPL you can do this without issue:

```
> (set phrase '"Don't Panic")
"Don't Panic"
> phrase
"Don't Panic"
> (set phrase '"Mostly Harmless")
"Mostly Harmless"
> phrase
"Mostly Harmless"
>
```

In previous sections we've set variables and worked with those variables in the REPL (saving us some typing), so this should all seem a bit familiar.

As such, this should be fairly intuitive clear at this point:

```
> (set the-answer 42)
42
> (* the-answer 2)
```

```
84
> (* the-answer the-answer)
1764
> (* the-answer the-answer the-answer)
74088
>
```

Unlike Erlang, the LFE REPL doesn't have the `b()` and `f()` functions ("show bound variables" and "flush bound variables" respectively).

2.4.2 Variables in LFE Modules

Unlike Lisp, LFE doesn't support global variables, so (unless you create some dirty hacks!) you won't be doing things like this in your modules:

```
(defvar *sneaky-global-data* ...)
(defparameter *side-effect-special* ...)
(defconstant +my-constant+ ...)
```

(Not to mention that LFE doesn't even *define* `defvar`, `defparameter`, or `defconstant`.)

As such, you shouldn't run into variables that are defined at the module-level, only inside actual functions.

2.4.2 Variables in Functions

There are *all sorts* of ways one might set a variable in an LFE function. The snippets below illustrate some of these, though for demonstration purposes, they are executed in the REPL.

```
> (let ((x 2)
        (y 3))
    (list x y (* x y)))
(2 3 6)
>
```

Above we set two variables, and then withing the scope of the `let` with display some values, one of which is computed from the variables.

```
> (let* ((x 2)
         (y 3)
         (z (* x y)))
    (list x y z))
(2 3 6)
>
```

In this example, we make use of `let*`'s ability to use defined variables in subsequent variables assignments. Tying this with regular `let` will result in an error.

```
> (let (((tuple name place age) #("Ford Prefect" "Betelgeuse Seven" 234)))
    (list name place age))
("Ford Prefect" "Betelgeuse Seven" 234)
>
```

Here is an example of multiple-binding in LFE. We haven't covered patterns yet, but we will

– and this example is making use of patterns to assign data from the given record to the variables in the `tuple`.

Patterns may be used in several different LFE forms, each of which may do some varaible binding.

2.5 Pattern Matching

2.5.1 What Are Patterns?

Pattern matching in Erlang is huge, and it has a proportional impact on LFE and what one can do with this dialect of Lisp. Pattern matching in LFE can be used in function clauses, `let`, `case`, `receive` and in the macros `cond`, `lc`, and `bc`. From the REPL, pattern matching may be done in `set` as well.

Pattern matching in LFE happens when an expression matches a given pattern, e.g.:

```
(... (<pattern> <expression>) ...)
```

where the `<pattern>` might be something like this:

```
(binary (f float (size 32))
        (b bitstring))
```

or this:

```
(tuple 'ok value)
```

or this:

```
(list a b c)
```

or this:

```
(cons h t)
```

and the `<expression>` is any legal LFE expression. Ideally, it will return data that will be matched by the pattern.

If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs. All of this is best understood through the examples given below. Each example is preceeded by the general form of pattern as used in the given context. This should help keep things clear, even when the examples get convoluted.

2.5.2 Patterns in Forms

2.5.2.1 `let`

Pattern matching in `let` has the following general form:

```
(let ((<pattern> <expression>)
```

```
      (<pattern> <expression>) ... )
  ... )
```

Examples:

```
> (let (((tuple len status data) #(8 ok "Trillian")))
       (list len status data))
(8 ok "Trillian")
>
```

In this example, we have a pattern of `(tuple len status data)` and this is getting matched against our expression which is some data of the form `#(8 ok "Trillian")`. The pattern expects a tuple, and a tuple is what we gave it. With the pattern's variables bound inside the `let`, we can return a list of the variables.

If our pattern was written to expect a list and the expression was a tuple, we'd get a `badmatch` error:

```
> (let (((list len status data) #(8 ok "Trillian")))
       (list len status data))
exception error: #(badmatch #(8 ok "Trillian"))

>
```

Whatever our expression is going to be needs to be matched in the pattern. If we had a list integers in the expression, we would need a pattern like `(list i1 i2 i3 ...)`.

Here's a super-simplified version of a `let` with pattern matching:

```
> (let ((data '"Trillian"))
       (list data))
("Trillian")
>
```

Here our pattern was simply the variable `data` and our expression was the string "Trillian". This, of course, is easily recognized as a standard variable assignment within a `let`.

Patterns can nest, though, and with this you can start to get a sense of the power they hold. Let's look at a more complicated example:

```
> (let (((tuple lens status data)
         #((8 43) #(err "msg too short") "Trillian")))
       (list lens status data))
("\b+" #(err "msg too short") "Trillian")
>
```

As you can see, we've nested our expression: length is a two-valued list and status is a two-valued tuple. Our pattern, however, is still simple. But this is going to change: we want to extract our data into more variables, and we do this by mirroring the expression data structure in the pattern itself:

```
> (let (((tuple (list len-data len-total) (tuple status-code status-msg) data)
         #((8 43) #(err "msg too short") "Trillian")))
       (list len-data len-total status-code status-msg data))
(8 43 err "msg too short" "Trillian")
```

```
>
```

As you can see, our nested pattern extracted the data into the pattern's variables. If all we cared about was the status message, we could make this simpler by using the "I don't care" variable (the underscore):

```
> (let (((tuple (list _ _) (tuple _ status-msg) _)
         #((8 43) #(err "msg too short") "Trillian")))
       (list status-msg))
("msg too short")
```

Having seen these examples, you are probably gaining some insight into the power of pattern matching in Erlang and LFE. There's more, though :-) See below for equally potent uses.

2.5.2.2 `case`

Pattern matching in `case` has the following general form:

```
(case <expression>
  (<pattern> <expression> ... )
  (<pattern> <expression> ... )
  ...)
```

Keep in mind that `case` may also be used (optionally) inside the `try` form. For more information on `try`, see section 5.2.

Let's take a look at `case` in action:

```
> (set data #(6 warn "Arthur"))
#(6 warn "Arthur")
> (case data
    ((tuple len 'ok msg)
      (: io format '"~s seems good.~n" (list msg)))
    ((tuple len 'err msg)
      (: io format '"There's a problem with ~s.~n" (list msg)))
    ((tuple len 'warn msg)
      (: io format '"Be careful of ~s.~n" (list msg))))
Be careful of Arthur.
ok
>
```

The patterns we are using in this `case` example expect data of one particular format, differentiating by the second element of the provided tuple. With new data, we can exercise the other cases:

```
> (set data #(8 ok "Trillian"))
#(8 ok "Trillian")
```

We won't re-type the `case` example here; just hit the "up" arror until you get to the `case` entry and hit return:

```
> (case ...)
Trillian seems good.
ok
>
```

Similarly, we can test the remaining case:

```
> (set data #(6 err "Zaphod"))
#(6 err "Zaphod")
> (case ...)
There's a problem with Zaphod.
ok
>
```

2.5.2.3 `receive`

Pattern matching in `receive` has the following general form:

```
(receive
  (<pattern> ... )
  (<pattern> ... )
  ...
  (after timeout
    ... ))
```

There is a tutorial on working with Erlang's [light weight processes in LFE](), and several example usages of `receive` are given there. On the second page of that tutorial, we see that any message sent to `receive` is accepted and processed. In the example below, we replace the simple pattern of the whole data (i.e., `msg`) with a series of patterns that will print only if the message matches one of the provided patterns.

Save the following in a file named `rcv-pttrn.lfe`:

```
(defmodule rcv-pttrn
  (export (safety-check 0)))

(defun safety-check ()
  (receive
    ((tuple 'ok item)
      (: io format '"~s is safe to approach.~n" (list item))
      (safety-check))
    ((tuple 'warn item)
      (: io format '"Approach ~s with extreme caution.~n" (list item))
      (safety-check))
    ((tuple 'crit item)
      (: io format '"Withdraw from ~s immediately!~n" (list item))
      (safety-check))))
```

Next, start up the LFE REPL, compile the module above, and start our safety server:

```
> (c '"rcv-pttrn")
#(module rcv-pttrn)
> (set pid (spawn 'rcv-pttrn 'safety-check ()))
<0.34.0>
>
```

Now let's give our patterns a try by sending messages to the server process:

```
> (! pid #(ok "Earth"))
#(ok "Earth")
Earth is safe to approach.
```

```
> (! pid #(warn "Frogstar"))
#(warn "Frogstar")
Approach Frogstar with extreme caution.
> (! pid #(crit "Krikkit"))
#(crit "Krikkit")
Withdraw from Krikkit immediately!
>
```

As you can see, the `receive` patterns are working.

We can also see what happens when we send messages that don't match any of the defined patterns:

```
> (! pid #(noop "This won't be matched"))
#(noop "This won't be matched")
> (! pid '"Neither will this"))
"Neither will this"
>
```

Absolutely nothing, that's what. Well, nothing from the process we spawned, that is… just the REPL doing its thang.

2.5.2.4 `cond`

Pattern matching in `cond` has the following general form:

```
(cond (<test> ... )
      ((?= <pattern> <expr>) ... )
      ... )
```

Typically, a `cond` looks like this:

```
(cond ((== a 1) (: io format '"It's one!"))
      ((== a 2) (: io format '"It's two!")))
```

In other words, a series of tests with conditional results. LFE extends the basic form with support for pattern matching, as seen in the general form above.

Here's an example of how one can do pattern matching in LFE with `cond` (starting with the setting of some data):

```
> (set data #(8 ok "Trillian"))
#(8 ok "Trillian")
> (cond ((?= (tuple len 'ok msg) data)
         (: io format '"~s seems good.~n" (list msg)))
        ((?= (tuple len 'err msg) data)
         (: io format '"There's a problem with ~s.~n" (list msg)))
        ((?= (tuple len 'warn msg) data)
         (: io format '"Be careful of ~s.~n" (list msg))))
Trillian seems good.
ok
>
```

Note that this is a replacement of the `case` example above.

We can set the `data` variable differently to exercise the other code paths, and then enter the `cond` expression from above (elided below to save space):

```
> (set data #(6 warn "Arthur"))
#(6 warn "Arthur")
> (cond ... )
Be careful of Arthur.
ok
> (set data #(6 err "Zaphod"))
#(6 err "Zaphod")
> (cond ... )
There's a problem with Zaphod.
ok
>
```

2.5.3 Special Cases

2.5.3.1 `set` in the REPL

Using `set` in the REPL has the following general form:

```
(set <pattern> <expression>)
```

Note that `set` is only valid when running the LFE shell. Example usage:

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

2.5.3.2 Aliases with `=`

Aliases are defined with the following general form:

```
( ... (= <pattern 1> <pattern 2>) ... )
```

Aliases can be used anywhere in a pattern. A quick example of this, updating the previous example with aliases:

```
> (set (= (tuple len status data) (tuple a b c))
       #(8 ok "Trillian"))
#(8 ok "Trillian")
>
```

The same variables that were bound in the previous example are bound in this one:

```
> len
8
> status
```

```
ok
> data
"Trillian"
>
```

In addition, however, we have aliased new variables to these:

```
> a
8
> b
ok
> c
"Trillian"
```

2.5.3.3 Arguments to `defun`

Pattern matching in functions has the following general form:

```
    (defun name
      ((argpat ...) ...)
      ...)
```

We haven't covered functions yet (that's coming up in Chapter 4), so this will be a short preview focusing just on the pattern usage in functions, with more detail coming later.

Proper functions can't be defined in the LFE REPL, so save the following to `func-pttrn.lfe`:

```
(defmodule func-pttrn
  (export (safety-check 2)))

(defun safety-check
  (('ok msg)
    (: io format '"~s seems good.~n" (list msg)))
  (('warn msg)
    (: io format '"There's a problem with ~s.~n" (list msg)))
  (('crit msg)
    (: io format '"Be careful of ~s.~n" (list msg))))
```

As you can see, the usual function arguments have been replaced with a pattern. In particular, this function will accept any of three options with two arguments each: where the first argument is `'ok`, or where it is `'warn`, or where it is `'crit`.

Let's compile our new module from the LFE REPL:

```
> (c '"func-pttrn")
#(module func-pttrn)
>
```

Now let's step it through its paces:

```
> (: func-pttrn safety-check 'ok '"Trillian")
Trillian seems good.
ok
> (: func-pttrn safety-check 'warn '"Arthur")
There's a problem with Arthur.
```

```
ok
> (: func-pttrn safety-check 'crit '"Zaphod")
Be careful of Zaphod.
ok
>
```

If a pattern is not matched in our example (which has no fallback pattern), an error is raised:

```
> (: func-pttrn safety-check 'oops '"Eccentrica Gallumbits")
exception error: #(case_clause #(oops "Eccentrica Gallumbits"))
  in (func-pttrn safety-check 2)
```

2.5.3.4 Arguments to Anonymous Functions

One can use patterns in arguments with anonymous functions similarly to how one does with named functions, demonstrated above. In LFE, this is done with `match-lambda`. Here's an example done in the REPL:

```
> (set safety-check
    (match-lambda
      (('ok msg)
        (: io format '"~s seems good.~n" (list msg)))
      (('warn msg)
        (: io format '"There's a problem with ~s.~n" (list msg)))
      (('crit msg)
        (: io format '"Be careful of ~s.~n" (list msg)))))
#Fun<lfe_eval.31.53503600>
>
```

Usage is similar as well:

```
> (funcall safety-check 'warn '"Arthur")
There's a problem with Arthur.
ok
> (funcall safety-check 'oops '"Eccentrica Gallumbits")
exception error: function_clause

>
```

2.5.3.5 Patterns in Comprehensions

List and binary comprehensions make use of patterns in a limited sense. They have the following general forms:

```
(<- pat guard list-expr)
```

and

```
(<= bin-pat guard binary-expr)
```

where the `guard` in both cases is optional.

You can read more about LFE comprehensions in section 3.3

3 Lists and Simple Data

3.1 Lists

Lists in Erlang and LFE are straight-forward; those coming from another programming language will not find anything surprising about them. Lists are generally good for storing and iterating over data that is of a similar type. There are other types one can use for more structured or complex data type combos.

You can create lists in LFE in the following ways:

```
> (list 1 3 9 27)
(1 3 9 27)
> '(1 3 9 27)
(1 3 9 27)
> (== '(1 3 9 27) (list 1 3 9 27))
true
> (=:= '(1 3 9 27) (list 1 3 9 27))
true
>
```

To get the length of a list, you'll need to use the `length` function from the `erlang` module:

```
> (: erlang length '(1 2 3 4 5 6 7))
7
```

Later, we will discuss Lisp-specific functions that have been implemented in LFE, but this is a good time to mention a few classic functions:

```
> (car '(1 2 3 4 5 6))
1
> (cdr '(1 2 3 4 5 6))
(2 3 4 5 6)
> (cadr '(1 2 3 4 5 6))
2
> (cddr '(1 2 3 4 5 6))
(3 4 5 6)
> (cons '(1 2 3) '(4 5 6))
((1 2 3) 4 5 6)
>
```

There is an Erlang [module dedicated to handling lists] that we can take advantage of:

```
> (: lists append '(1 2) '(3 4))
(1 2 3 4)
> (: lists append (list '(1 2) '(3 4) '(5 6)))
(1 2 3 4 5 6)
>
```

You can also use the ++ operator to combine two lists:

```
> (++ '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
>
```

Here's a `map` example that generates the same list we manually created above:

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

Another one is `filter`, but before we use it, let's first define a predicate that returns `true` for even numbers:

```
> (set evenp
    (lambda (x)
      (== 0 (rem x 2))))
#Fun<lfe_eval.10.53503600>
>
```

Not let's try out `filter` with our new predicate:

```
> (: lists filter evenp '(1 2 3 4 5 6))
(2 4 6)
>
```

There are many, many more highly useful functions in the `lists` module – be sure to give the docs a thorough reading, lest you miss something fun!

3.1.1 I/O Lists

There is another type of list that is used for such things as file and network operations; it's called an `IoList`. An `IoList` is a list whose elements are either

- integers ranging from 0 to 255,
- binaries,
- other `IoLists`, or
- a combination of these.

Here's an example for you:

```
> (list '"hoopy" 42 #b("frood" 210) (list #b(42 84 126) 168 252))
("hoopy" 42 #B(102 114 111 111 100 210) (#B(42 84 126) 168 252))
>
```

You don't need to flatten `IoLists`; they get passed as they are to the various low-level functions that accept an `IoList` and Erlang will flatten them efficiently for you.

We saw an example of this in a previous section when we were playing with strings as binaries. We ended up calling a function that accepted an `IoList` as a parameter and this saved us from having to flatten the list of binaries ourselves. If you recall, `data` was a long string and the `split` function returned a list of binaries:

```
> (: unicode characters_to_list
    (: binary split data (binary "who really knows ")))
```

```
"There's a frood where his towel is."
>
```

3.2 Tuples

Tuples are the data melting pot for Erlang: you can combine any of Erlang's data types (including lists and other tuples) into a single composite data type. This comes in very handy with pattern matching, but in general, makes passing data around much easier.

Creating a tuple can be as simple as:

```
> (tuple)
#()
>
```

But perhaps more useful:

```
> (tuple 'odds '"5 to 1 against")
#(odds "5 to 1 against")
>
```

You could also have done this:

```
> #(odds "5 to 1 against")
#(odds "5 to 1 against")
>
```

Here's a simple data structure:

```
> (set data
    (tuple
      '|things to see|
      (list '"moons of Jaglan Beta"
            '"beaches of Santraginus V"
            '"desert world of Kakrafoon"
            '"heavy river Moth")
      '|things to avoid|
      (list '"Ravenous Bugblatter Beast of Traal"
            '"small piece of fairy cake")))
#(|things to see|
  ("moons of Jaglan Beta"
   "beaches of Santraginus V"
   "desert world of Kakrafoon"
   "heavy river Moth")
  |things to avoid|
  ("Ravenous Bugblatter Beast of Traal" "small piece of fairy cake"))
>
```

Now let's poke around at our new data structure:

```
> (: erlang tuple_size data)
4
> (: erlang element 1 data)
|things to see|
> (: erlang element 3 data)
|things to avoid|
```

```
>
```

Using the erlang module's function is one way to get our tuple data, but you'll probably not use that as much as the next method we show you.

We're going to sneak ahead here a bit, and touch on patterns again; we'll explain in more detail in the actual section on patterns! For now, though, just know that extracting data from structures such as our tuple is very easy with patterns. Take a look:

```
> (set (tuple key1 val1 key2 val2) data)
#(|things to see|
  ("moons of Jaglan Beta"
   "beaches of Santraginus V"
   "desert world of Kakrafoon"
   "heavy river Moth")
  |things to avoid|
  ("Ravenous Bugblatter Beast of Traal" "small piece of fairy cake"))
>
```

To be clear: had we needed to do this in a function, we would have used `let` ;-)

Now we can references our data by the variables we bound when we extracted it with the pattern in our `set` call:

```
> key1
|things to see|
> key2
|things to avoid|
> val1
("moons of Jaglan Beta"
 "beaches of Santraginus V"
 "desert world of Kakrafoon"
 "heavy river Moth")
> val2
("Ravenous Bugblatter Beast of Traal" "small piece of fairy cake")
>
```

3.3 Comprehensions

In the section on lists, we gave an example of building a list using the `map` function:

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

This sort of approach should be familiar to many programmers, even those who aren't adepts at functional programming. This is a well-known pattern. Erlang offers another pattern, though: comprehensions.

LFE supports Erlang comprehensions via two macros: `lc` for list comprehensions and `bc` for bitstring comprehensions.

3.3.1 List Comprehensions

Let's take a look at an example and then discuss it. Here's a list comprehension version of our `map/lambda` combo above:

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

This can be translated to English as "the list of integers whose values are `x` raised to the power of `3` where `x` is taken from the list we provided, iterated in order from first to last."

In Erlang, this would have looked like the following:

```
1> [trunc(math:pow(X,3)) || X <- [0,1,2,3]].
[0,1,8,27]
2>
```

As we can see, the LFE syntax is not as concise as the native Erlang syntax, though it is pretty close. Our original example is 62 characters long; the LFE list comprehension is 49 characters long; the Erlang version is 41 characters.

To a Lisper, the original is probably much more legible. However, in Erlang these is no question that the list comprehensions are shorter and easier to read than using anonymous functions.

3.3.1 Bitstring Comprehensions

For binary data, we have something similar to the list comprehension. Here's what a bitstring comprehension looks like (adapted from the example given by Francesco Cesarini and Simon Thompson in their book, "Erlang Programming"):

```
> (bc
    ((<= (x (size 1)) (binary (42 (size 6)))))
    ((bnot x) (size 1)))
#B((21 (size 6)))
>
```

Note that the bitstring comprehension uses the `<=` operator (not to be confused with the `=<` equality operator!) instead of the `<-` that list comprehensions use.

Here's the Erlang version:

```
2> << <<bnot(X):1>> || <<X:1>> <= <<42:6>> >>.
<<21:6>>
3>
```

As we might expect, the native Erlang version is much more concise. Fortunately, though, in LFE we don't need to enter the whole binary form, just the bit syntax portion. In other words, instead of writing this:

```
(binary (x (size 1)))
```

and this:

```
(binary ((bnot x) (size 1)))
```

we only had to write this:

```
(x (size 1))
```

and this:

```
((bnot x) (size 1))
```

3.4 Property Lists and Hashes

3.4.1 Property Lists

Property lists are just lists whose entries are key/value tuples. Alternatively, an entry may be a single atom, in which case it implies a tuple with the atom as the key and `true` as the value.

Since there's no special type here, we just create a regular list:

```
> (set plist
    (list
      (tuple '|to see|
             '"moons of Jaglan Beta"
             '"beaches of Santraginus V")
      (tuple '|to avoid|
             '"small piece of fairy cake")))
>
```

Let's see what keys we have defined:

```
> (: proplists get_keys plist)
(|to avoid| |to see|)
>
```

Extracting data by key:

```
> (: proplists lookup '|to see| plist)
#(|to see| "moons of Jaglan Beta" "beaches of Santraginus V")
> (: proplists lookup '|to avoid| plist)
#(|to avoid| "small piece of fairy cake")
>
```

If you know that your value is single-valued (e.g., not a list), then you can do this:

```
> (: proplists get_value '|to avoid| plist)
"small piece of fairy cake"
>
```

There is more information about property lists on the docs page for them.

3.4.2 Hashes

There is no builtin “dictionary” or “hash” type in Erlang. However, there are some libraries that support data structures like these. There is also a concept of “records” which we will discuss in another section.

3.4.2.1 The Dictionary

The Erlang `dict` module implements a key/value dictionary part of which is an additional `dict` data type which supplements the built-in Erlang data types.

Here’s how you create a new `dict`:

```
> (set my-dict (: dict new))
#(dict
  0
  16
  16
  8
  80
  48
  #(() () () () () () () () () () () () () () () ())
  #(#(() () () () () () () () () () () () () () () ())))
>
```

Let’s check that there’s no actual data in it:

```
> (: dict size my-dict)
0
```

Now let’s add some!

```
> (set my-dict
    (: dict append '|to see| '"moons of Jaglan Beta" my-dict))
#(dict ...
> (set my-dict
    (: dict append '|to avoid| '"small piece of fairy cake" my-dict))
#(dict ...
>
```

As you might guess from the usage, `dicts` are not updated in-place. A new dictionary is returned with each call to `append`. As such, we need to `reset` with each append. Is everything there?

```
> (: dict size my-dict)
2
>
```

Looking good so far… Now let’s get some data out:

```
> (: dict fetch '|to avoid| my-dict)
("small piece of fairy cake")
>
```

Why the is the function called “append”? Well, `dict` accepts multiple values for keys. Let’s try this out, and then re-query our `dict`:

```
> (: erlang length (: dict fetch '|to see| my-dict))
1
> (set my-dict
    (: dict append '|to see| '"beaches of Santraginus V" my-dict))
#(dict ...
> (: erlang length (: dict fetch '|to see| my-dict))
2
> (: dict size my-dict)
2
> (: dict fetch '|to see| my-dict)
("moons of Jaglan Beta" "beaches of Santraginus V")
>
```

The size of the `my-dict` didn't change because we didn't add a new key; rather, we updated an existing one, appending a new value. The `|to see|` key now has two values in it.

You can also build `dicts` from a list of tuples:

```
> (set my-dict-2
    (: dict from_list '(#('key1 '"foo") #('key2 '"bar"))))
#(dict ...
> (: dict size my-dict-2)
2
>
```

There are many more functions to explore in the dict docs.

3.4.2.2 Other Hash Tables

OTP comes with the `ets` module which provides the ability to store very large quantities of data in an Erlang runtime system. The `ets` module supports hash tables of the following types:

- `set`
- `ordered_set`
- `bag`
- `duplicate_bag`

The documentation for this module is here, though we will be adding information on how to use this from LFE at a later point (likely a dedicated tutorial).

3.5 Records

3.5.1 Just Records

Sometimes lists, tuples, property lists, or hashes are not quite what is needed. With tuples, you can't name keys (without awkward work-arounds), and this makes working with large tuples rather cumbersome. Records are one way around this.

A record is a data structure for storing a fixed number of elements. It has named fields and

LFE provides some convenience functions/macros for interacting with them.

However, it is important to note that record expressions are translated to tuple expressions during compilation. Due to this, record expressions are not understood by the shell in both Erlang and LFE. The examples in this section, therefore, will assume that you are saving the code to a file.

Let's start by defining a record. Save this in a file named `record.lfe`:

```
(defmodule rec)

(defrecord person
  name
  address
  age)
```

Then load it up in the REPL:

```
> (slurp '"record.lfe")
#(ok rec)
>
```

Now let's create some people:

```
> (set ford
    (make-person name '"Ford Prefect"
                 address '"Betelgeuse Seven"
                 age 234))
#(person "Ford Prefect" "Betelgeuse Seven" 234)
> (set trillian
    (make-person name '"Tricia Marie McMillan"
                 age 60))
#(person "Tricia Marie McMillan" undefined 60)
>
```

Let's define a non-person, too:

```
> (set zaphod #("Zaphod Beeblebrox"))
#("Zaphod Beeblebrox")
>
```

Some quick checks:

```
> (is-person ford)
true
> (is-person zaphod)
false
>
```

If you remember working with the tuples, property lists, and dictionaries, then you will enjoy the relative succinctness of the following usages:

```
> (person-name ford)
"Ford Prefect"
> (person-address ford)
"Betelgeuse Seven"
> (person-age ford)
```

```
234
>
```

Let's make some changes to our data:

```
> (set ford
    (set-person-age ford 244))
#(person "Ford Prefect" "Betelgeuse Seven" 244)
> (person-age ford)
244
>
```

Just as we saw with the `dict` examples, `set-person-age` doesn't modify the data in-place, but rather returns a new record. If we want to use that data in the future, we'll need to assign it to a variable (sensibly, we re-use the `ford` variable here).

Also, note that there are also `set-person-name` and `set-person-address`.

3.5.2 Records and ETS

Additional convenience functions for records are provided by LFE, but some of these will only make sense in the context of ETS (Erlang Term Storage), when when the ability to store large amounts of data in memory becomes important. We will be discussing this in detail later, but this section provides a quick preview.

Let's create an ETS table:

```
> (set people
    (: ets new 'people-table '(#(keypos 2) set)))
16401
>
```

Now let's insert the two `person` records that we created above:

```
> (: ets insert people ford)
true
> (: ets insert people trillian)
true
>
```

Now that we have a table with some data in it, we can do some querying. Let's start with the `emp-match` LFE macro. Here's how we can get the name for every record in the table:

```
> (: ets match people (emp-person name '$1))
(("Ford Prefect") ("Tricia Marie McMillan"))
>
```

Or, we can adjust that to return the name and address:

```
> (: ets match people (emp-person name '$1 address '$2))
(("Ford Prefect" "Betelgeuse Seven") ("Tricia Marie McMillan" undefined))
>
```

With the `match-person` LFE macro, we can do more sophisticated querying

```
> (: ets select people
    (match-spec (((match-person name found-name age found-age))
                 (when (> 100 found-age))
                 found-name)))
> ("Tricia Marie McMillan")
>
```

Here we've done a select in the "people" table for any person whose age is less than 100.

Here's what it looks like when multiple records are returned:

```
> (: ets select people
    (match-spec (((match-person name found-name age found-age))
                 (when (< 21 found-age))
                 found-name)))
> ("Ford Prefect" "Tricia Marie McMillan")
>
```

This should be enough of an ETS taste to last until you get to the dedicated tutorial ;-)

3.6 `.hrl` Header Files

4 Functions and Modules

4.1 Functions

4.1.1 Intro and Recap

4.1.2 Parity

4.1.3 More Patterns

4.1.4 Anonymouns Functions

4.1.5 Higher-Order Functions

4.1.5.1 As Input

4.1.5.2 As Output

4.2 LFE-Specific Functions and Macros

4.2.1 Core Forms

```
(quote e)
(cons head tail)
(car e)
(cdr e)
```

```
(list e ... )
(tuple e ... )
(binary seg ... )
(lambda (arg ...) ...)
(match-lambda
  ((arg ... ) (when e ...) ...)
  ... )
(let ((pat (when e ...) e)
      ...)
  ... )
(let-function ((name lambda|match-lambda)
               ... )
  ... )
(letrec-function ((name lambda|match-lambda)
                  ... )
  ... )
(let-macro ((name lambda-match-lambda)
            ...)
  ...)
(progn ... )
(if test true-expr false-expr)
(case e
  (pat (when e ...) ...)
   ... ))
(receive
  (pat (when e ...) ... )
  ...
  (after timeout ... ))
(catch ... )
(try
  e
  (case ((pat (when e ...) ... )
          ... ))
  (catch
     (((tuple type value ignore) (when e ...)

      ... )
     ... )
  (after ... ))
(funcall func arg ... )
(call mod func arg ... )

(define-function name lambda|match-lambda)
(define-macro name lambda|match-lambda)
```

4.2.2 Macro Forms

```
(: mod func arg ... ) =>
        (call 'mod 'func arg ... )
(? timeout default )

(++ ... )
(list* ...)
(let* (...) ... )
(flet ((name (arg ...) ...)
       ...)
  ...)
(flet* (...) ... )
(fletrec ((name (arg ...) ...)
```

```
          ...)
  ...)

(cond ... )
(andalso ... )
(orelse ... )
(fun func arity)
(fun mod func arity)
(lc (qual ...) ...)
(bc (qual ...) ...)
(match-spec ...)
```

4.2.3 Common Lisp Inspired Macros

```
(defun name (arg ...) ...)
(defun name
  ((argpat ...) ...)
  ...)

(defmacro name (arg ...) ...)
(defmacro name arg ...)
(defmacro name
  ((argpat ...) ...)
  ...)

(defsyntax name
  (pat exp)
  ...)

(macrolet ((name (arg ...) ...)
           ...)
  ...)
(syntaxlet ((name (pat exp) ...)
            ...)
  ...)

(defmodule name ...)
(defrecord name ...)
```

4.2.4 Scheme Inspired Macros

```
(define (name arg ...) ...)
(define name lambda|match-lambda)
(define-syntax name
  (syntax-rules (pat exp) ...)|(macro (pat body) ...))
(let-syntax ((name ...)
             ...)
  ...)
(begin ...)
(define-module name ...)
(define-record name ...)
```

4.2.5 Additional Lisp Functions

```
(<arith_op> expr ...)
(<comp_op> expr ...)
        The standard arithmentic operators, + - * /, and comparison
```

```
        operators, > >= < =< == /= =:= =/= , can take multiple
        arguments the same as their standard lisp counterparts. This
        is still experimental and implemented using macros. They do,
        however, behave like normal functions and evaluate ALL their
        arguments before doing the arithmetic/comparisons operations.

(acons key value list)
(pairlis keys values list)
(assoc key list)
(assoc-if test list)
(assoc-if-not test list)
(rassoc value list)
(rassoc-if test list)
(rassoc-if-not test list)
        The standard association list functions.

(subst new old tree)
(subst-if new test tree)
(subst-if-not new test tree)
(sublis alist tree)
        The standard substituition functions.

(macroexpand-1 expr environment)
        If Expr is a macro call, does one round of expansion,
        otherwise returns Expr.

(macroexpand expr environment)
        Returns the expansion returned by calling macroexpand-1
        repeatedly, starting with Expr, until the result is no longer
        a macro call.

(macroexpand-all expr environment)
        Returns the expansion from the expression where all macro
        calls have been expanded with macroexpand.

        NOTE that when no explicit environment is given the
        macroexpand functions then only the default built-in macros
        will be expanded. Inside macros and in the shell the variable
        $ENV is bound to the current macro environment.

(eval expr environment)
        Evaluate the expression expr. Note that only the pre-defined
        lisp functions, erlang BIFs and exported functions can be
        called. Also no local variables can be accessed. To access
        local variables the expr to be evaluated can be wrapped in a
        let defining these.

        For example if the data we wish to evaluate is in the variable
        expr and it assumes there is a local variable "foo" which it
        needs to access then we could evaluate it by calling:

        (eval `(let ((foo ,foo)) ,expr))
```

4.3 Modules

4.3.1 What Modules Do

4.3.2 What Modules Don't Do

4.3.3 Creating a Module

4.3.4 Parameterized Modules

```
(defmodule (zaphod-rest-api request)
  (export (get-greeting 2)))

(defun get-greeting
  (('GET ())
   (tuple 'output '"Zaphod says 'hello!'"))
  (('GET _)
   (tuple 'output '"Zaphod says 'hello' to anything...")))
```

```
> (set req (: zaphod-rest-api new '"a request"))
#(zaphod-rest-api "a request")
> (call req 'get-greeting 'GET ())
#(output "Zaphod says 'hello!'")
> (call req 'get-greeting 'GET '"stuff")
#(output "Zaphod says 'hello' to anything...")
>
```

4.4 Projects with Rebar

In this section we'll be exploring how rebar can be used to manage LFE projects.

This section will make use of two example projects on github:

-
-

4.4.1 Collections of Modules

The first question we should probably address is this: How are we defining a project?

An LFE project is a set of modules developed and distributed to accomplish a particular goal. The project should have a rebar configuration file, a source directory with `.lfe` files in it, possibly an include directory, andn ideally unit tests in a `test` directory.

4.4.2 Project Structure

Let's expand upon the project definition given above, focusing on the directory structure of a prototypical project and some of the files one might find in an LFE project.

```
├── ebin
|   └── libexample.app
├── src
|   └── libexample.lfe
├── Makefile
├── README.md
└── rebar.config
```

This is from a sample project whose purpose is to provide a library for use by other LFE (or Erlang!) projects. More on that below.

Rebar supports LFE files. All that it needs is the standard `rebar.config` and an `.app` file in the `ebin` directory. With these, Rebar will be able to download the project dependencies and compile the `*.lfe` files in `src` to the `ebin` directory as `*.beam` files.

4.4.3 Dependencies

Dependencies are handled very nicely with Rebar: just add a git repo in your `rebar.config` file like so:

```
{deps, [
    {lfe, ".*", {git, "git://github.com/rvirding/lfe.git", "develop"}}
  ]}.
```

Any dependencies listed here will be downloaded with the `rebar get-deps` command. Once downloaded, issuing the `rebar compile` command will not only compile your project's `src/*` files into its `ebin` directory, but will compile all dependency project source code as well.

4.4.4 Defining a Library Project

We've seen the directory structure above for a library project. We're defining a "library" project as one that doesn't start up any services as part of its basic operations. Instead, it provides code that other projects make use of.

In particular, in a library project, you do not need to define `mod` in your `ebin/libname.app` file. Similarly, in your `src` directory, you do not need to create application nor supervisor files.

We're making a rather arbitrary distinction here (between "library" projects and "service" projects, and one with undoubtedly many blurry lines. Regardless, it may be instructive or useful as a guideline.

4.4.5 Defining a Service Project

4.4.6 Distributing A Project

4.4.7 Installing Projects

5 Recursion

5.1 See Section 5

Sorry, couldn't resist.

5.2 A Brief History

In functional languages, recursion plays an important role. For Erlang in particular, recursion is important because variables can't be changed. It is therefor often very useful to take advantage of recursion in order to work with changing values (examples are given in the latter half of this chapter).

However, recursion is interesting in and of itself. The roots of functional programming languages such as Lisp, ML, Erlang, Haskell and others, can be traced to the concept of recursion in general and the λ-calculus in particular.

The Italian mathematician Giuseppe Peano seems to have been one of the first to have made prominent use of recursion when defining his axioms for the natural numbers. Furthermore, Peano gave Bertrand Russell a copy of his "Formulario" (in fact, he gave Russell *all* of his published works!). This impacted Russell hugely and quite possibly influenced his work on the "Principia Mathematica" which he coauthored several years later.

It was from the Principia that Alonzo Church derived his lambda notation. When Church's student, John McCarthy, created Lisp, he used both the lambda notation and the related concept of recursion in his new language. (Interestingly enough, McCarthy and Dijkstra both advocated for the inclusions of recursion in ALGOL.) From John McCarthy's work onward, the lambda and recursion have been our constant companions.

5.3 A Preview

In the sections of the user guide, we explore various aspects of recursion as they can be formulated in Lisp Flavored Erlang. We will cover the following:

- The Dedekind-Peano Axioms
- Primitive Recursive Functions
- Total Recursive Functions
- The λ-Calculus
- Practical Examples in Computing
- Tail-Calls

If you just want to jump to the practical examples, please do so! You should feel no guilt when enjoying LFE or reading about LFE :-) The other sections are provided simply because it is very rare to find a practical coverage of the foundations of recursion and the λ-Calculus. There may be readers out there who want to know this reasons and history behind the concepts studied; most of this chapter is for them.

5.4 The Dedekind-Peano Axioms

For those that are math-averse, don't let this frighten you – this will be a peaceful journey that should not leave you bewildered. Rather, it will provide some nice background for how recursion came to be used. With the history reviewed, we'll make our way into practical implementations.

5.4.1 Foundations

Despite the fluorescence of maths in the 17th and 18th centuries and the growing impact of number theory, the ground upon which mathematics were built was shaky at best. Indeed, what we now consider to be the foundations of mathematics had not even been agreed upon (and this didn't happen until the first half of the 20th century with the maturation of logic and rise of axiomatic set theory).

One of the big problems facing mathematicians and one that also prevented the clarification of the foundations, was this: a thorough, precise, and consistent definition of the natural numbers as well as operations that could be performed on them (e.g., addition, multiplication, etc.). There was a long-accepted intuitive understanding, however, this was insufficient for complete mathematical rigor.

Richard Dedekind addressed this with his method of cuts, but it was Giuseppe Peano that supplied us with the clearest, most easily described axioms defining the natural numbers and arithmetic, wherein he made effective use of recursion. His definitions can be easily found in text books and on the Internet; we will take a slightly unique approach, however, and cast them in terms of LFE.

5.4.2 A Constant and Equality

The first five Peano axioms deal with the constant (often written as "0") and the reflexive, symmetric, transitive and closed equality relations. These don't relate recursion directly, so we're going to skip them ;-)

5.4.3 Successor Function

The concept of the "successor" in the Peano axioms is a primitive; it is taken as being true without having been proved. It is informally defined as being the next number following a given number "n".

In LFE:

```
(defun successor (n)
  (+ n 1))
```

The things to keep in mind here are that 1) we haven't defined addition yet, and 2) you must not interpret ”+” as addition in this context, rather as the operator that allows for succession to occur. In the world of the Peano axioms, ”+” is only validly used with “n” and “1”.

This function is defined as being “basic primitive recursive”. The basic primitive recursives are defined by axioms; the term was coined by Rózsa Péter.

5.4.4 The Remaining Axioms

The remaining three Peano axioms do not touch upon recursion directly, so we leave them to your own research and reading pleasure.

5.5 Primitive Recursive Functions

In the previous section, we leaned about the primitive recursive function called the “successor”, one that was used by Peano in his axioms. There are other primitive recursive functions as well, and these are usually given as axioms (i.e., without proof):

- the “zero function”
- the “projection function”
- “identity function”

These combined with the Peano axioms allow us to define other primitive recursive functions.

5.5.1 Addition

In the literature, the definition for Peano addition is done in the following manner:

```
a + 0 = a,
a + S(b) = S(a + b)
```

where `S` is the successor function defined in the previous section.

First we have an identity function: any number that has zero added to it yields the result of the number itself.

Secondly, a number, when added with the successor of another number is equal to the successor of the two numbers combined. Let’s take a look at an example:

- The number 0, when applied to the successor function yields 1 (`S(0) = 1`)
- Therefore, `a + S(0) = a + 1`
- By Peano’s definition of addition then, we have `a + 1 = S(a + 0)`
- Which then gives `a + 1 = S(a)`

In other words, the successor of `a` is `a + 1`.

These rules for addition are sometimes given in the following pseudo code:

```
add(0, x) = x
add(succ(n), x) = succ(add(n, x))
```

In LFE, we'd like to maintain symmetry with this. We could try to construct a function that had both definitions as pattern arguments, thus alleviating the need for two function definitions. However, to perfectly map the pseudo code to LFE, we'd have to put a function call in our pattern… and that's not possible.

If, though, we do a little algebraic juggling, we can work around this. In our pseudo code we have two parameters: `succ(n)` and `x`. If we apply a "predecessor" function to `succ(n)`, we'll just have `n` – which would do nicely for a matched function argument in LFE. But we'll also need to apply this predecessor function to the `n` on the other side of the equation.

Let's create such a "predecessor" function:

```
(defun predecessor
  ((0) 0)
  ((n) (- n 1)))
```

Now, we can recast the canonical form above using the workaround of the `predecessor` primitive recursive function, allowing us to use one function to define Peano's addition axiom:

```
(defun add
  ((0 x) x)
  ((n x) (successor (add (predecessor n) x))))
```

All of this may seem rather absurd, given what we do in every-day programming. Remember, though: the verbosity of these axioms and their derived definitions serves to explicitly show that no assumptions are being made. With a foundation of no assumption, we can be certain that each brick we lay on top of this sound (if possibly baroque) basis will be unshakable (baring the random proof by Gödel, of course).

5.5.2 Subtraction

Next up, let's take a look at subtraction:

```
sub(0, x) = x
sub(pred(n), x) = pred(sub(n, x))
```

Similar to addition above, we make some adjustments for the convenience of pattern matching:

```
(defun subtract
  ((0 x) x)
  ((n x) (predecessor (subtract (predecessor n) x))))
```

Due to the manner in which we have defined our functions, the usual usage is reversed for our `subtract` function. The first operand is not the number that is being subtracted from, but rather the number that is being *subtracted*.

We can see this in action if we put our definitions in a file called `prf.lfe` (named for "primitive recursive functions") and `slurp` it in the LFE REPL:

```
> (slurp '"prf.lfe")
#(ok prf)
```

```
> (subtract 1 100)
99
>
```

5.5.3 Multiplication

The last one of these that we will look at is multiplication, and then we'll move on to something a little more complicated :-)

```
mult(0, x) = 0
mult(succ(n), x) = x + (x * n)
```

Again, using our pattern workaround:

```
(defun multiply
  ((0 x) 0)
  ((n x) (add x (multiply x (predecessor n)))))
```

5.6 Partial Recursive Functions

We've covered primitive recursive functions in the previous section; now we'll take a brief look at what are called "partial recursive functions". These are functions that provide an output for given input but which may not be defined for *every* possible input.

Partial recursive functions are also referred to as "computable functions" and can be defined using Turing machines or the λ-calculus (among others). In fact, an equivalent definition of partial recursive function is actually a function that can be computed by a Turing machine.

5.7 Total Recursive Functions

As opposed to a partial recursive function, a *total* recursive function is one that is defined for all possible function inputs. Every primitive recursive function is total recursive. There are, however, total recursive functions that are *not* primitive recursive. The Ackermann function is one such.

5.7.1 The Ackermann Function

The Ackermann function is one of the simplest and earliest-discovered examples of a total recursive function that is not primitive recursive. The variant of the function that we present below is the two-variable version developed by Rózsa Péter and Raphael Robinson (the original was more verbose and with three variables).

Here is the function in LFE

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

As we can see, this function quite clearly calls itself ;-)

Here's some example usage:

```
> (c '"prf")
#(module prf)
> (: prf ackermann 0 0)
1
> (: prf ackermann 0 1)
2
> (: prf ackermann 1 0)
2
> (: prf ackermann 1 1)
3
> (: prf ackermann 1 2)
4
> (: prf ackermann 2 2)
7
> (: prf ackermann 2 4)
11
> (: prf ackermann 4 1)
65533
>
```

5.8 The λ-Calculus

Oh, yeah. We just went there: the λ-calculus.

Take heart, though… this is going to be fun. And after this bit, we'll finally get to the practical coding bits :-)

Keep in mind that the Peano axioms made use of recursion, that Giuseppe Peano played a key role in Bertrand Russell's development of the Principia, that Alonzo Church sought to make improvements on the Principia, and the lambda calculus eventually arose from these efforts.

Church realized when creating the λ-calculus that with only a lambda at his disposal, he could define numbers and perform arithmetic upon them. This is known as "Church encoding". Using what we have defined above, we should be able to peer into this forest of lambdas and perhaps perceive some trees.

Church, with his now-famous students Stephen Kleene and J. Barkley Rosser, established the λ-calculus as equivalent to a Turing machine for determining the computability of a given function. In particular, the Church-Turing Thesis states that the class of functions which are partial recursive functions has the same members as the class of functions which are computable functions.

Previously, we examined natural numbers and operations such as addition in the context of positive integers. However, in the sections below, we will be leaving behind the comfort of the familiar. The λ-calculus does not concern itself with natural numbers per se; rather the ability to do something a given number of times.

5.8.1 A Quick Primer

In the literature, you will see such things as:

```
λx.x
```

or

```
(λx.x)y
```

or

```
(λwyx.y(wyx))(λsz.z)
```

This is standard notation for the λ-calculus, and here's how you read it:

- an expression can be a name, a function, or an application, e.g.: `x` or `λx.x` or `(λx.x)3`
- a function is represented by a lambda followed by a name, a dot, and an expression, e.g.: `λx.x`
- an application is represented as two expressions right next to each other, e.g.: `(λx.x)3`

As such, one says that `λx.x` is a function that takes one parameter, `x`, and produces one output, `x`. `λxy.y` takes two parameters, `x` and `y` and produces one output, `y`.

5.8.2 Church Encoding

Let's get our feet wet with figuring out how we can define the natural numbers under Church encoding, starting with `zero`. In the standard λ-calculus, this is done in the following manner:

```
λs.λx.x
```

We are defining the successor function from above as `s`. We are also defining `x` as "that which represents zero". So this reads something like "We pass our counting function represented as `s` as the first parameter; there's nothing to do but then pass the second parameter `x` to the next function, which returns `x`". We never do anything with `s` and only return `x` itself.

In the λ-calculus, zero is defined as taking the successor function, doing nothing with it, and returning the value for zero from the identify function. In LFE, this is simple:

```
(defun zero ()
  (lambda (s)
    (lambda (x) x)))
```

We've got some nested functions that represent "zero"; now what? Well, we didn't use the successor inside the zero function, if we *do*, we should get "one", yes? But how? Well, we'll "apply" the successor function that is passed in, as opposed to ignoring it like we did in `zero`. Here's the Church numeral definition for `one`:

```
λs.λx.s x
```

Let's try that in LFE:

```
(defun one ()
  (lambda (s)
    (lambda (x)
      (funcall s x))))
```

Congratulations, you've written your second Church numeral in LFE now :-) Successive numbers are very similar: an additional `(funcall s` before the `(funcall s x`.

A small but significant caveat: technically speaking, the functions `zero` and `one` are not actual Church numerals, rather they wrap the Church numberals. Once you call these functions, you will have the Church numberals themselves (the lambda that is returned when the numberal functions are called).

Now that we see Church numerals are nested `lambdas` with nested calls on the successor function, we want to peek inside. How does one convert a Church numeral to, say, an interger representation? Looking at the `one` function, we can make an educated guess:

1. We will need to call `one` so that the top-most lambda is "exposed".
2. We will need to apply (`funcall`; it's more convenient) our choice of successor function to that top-most lambda.
3. We will need to apply our representation of "zero" to the next lambda.

With each of those done we end up with a solution that's actually quite general and can be used on any of our Church numerals. Here's a practical demonstration:

```
> (slurp '"church.lfe")
#(ok church)
> (one)
#Fun<lfe_eval.10.53503600>
> (funcall (funcall (one) #'successor/1) 0)
1
```

Typing that into the REPL whenever we want to check our Church numeral will be tedious. Let's write a function that allows us to get the integer representation of a Church numeral more easily. There are a couple ways to do this. First:

```
(defun church->int1 (church-numeral)
  (funcall (funcall church-numeral #'successor/1) 0))
```

This would require that we call our Church numeral in the following manner (assuming that we've re-`slurp`ed the `church.lfe` file):

```
> (church->int1 (one))
1
```

Alternatively, we could do this:

```
(defun church->int2 (church-numeral)
  (funcall (funcall (funcall church-numeral) #'successor/1) 0))
```

This second approach lets us pass the Church numeral without calling it:

```
> (church->int2 #'one/0)
```

As mentioned earlier, we know that we can get successive Church numerals by adding more `(funcall s` applications (i.e., incrementing with a successor function). For instance, here is the Church numeral `four`:

```
(defun four ()
```

```
  (lambda (s)
    (lambda (x)
      (funcall s
        (funcall s
          (funcall s
            (funcall s x)))))))
```

Using this method of writing out Church numerals is going to be more tedious that putting beans in a pile to represent integers. What can we do? Well, we need a general (i.e., non-integer) λ-calculus representation for a successor function. Then we need to be able to apply that to `zero` `n` times in order to get the desired Church numeral. Let's start with a Church numeral successor function:

```
λn.λs.λx. s (n s x)
```

We translate that to LFE with the following:

```
(defun church-successor (n)
  (lambda (s)
    (lambda (x)
      (funcall s
        (funcall
          (funcall n s) x)))))
```

Next we need a function that can give us a Church numeral for a given number of applications of the `church-successor`:

```
(defun get-church (church-numeral count limit)
  (cond ((== count limit) church-numeral)
        ((/= count limit)
         (get-church (church-successor church-numeral) (+ 1 count) limit))))
```

We're getting a little bit ahead of ourselves, since we haven't yet talked about countdown or countup recursive functions in LFE; consider this a teaser ;-)

Our `get-church` function keeps track of how many times it is recursed and returns the appropriate Church numeral when the limit has been reached. However, it's a bit cumbersome to use. Let's see if we can do better:

```
(defun get-church (integer)
  (get-church (zero) 0 integer))
```

Now we've got two `get-church` functions, each with different arity. `get-church/1` calls `get-church/3` with the appropriate initial arguments, at which point `get-church/3` calls itself until the limit is reached and returns a Church numeral.

Let's take a look:

```
> (slurp '"examples/church.lfe")
#(ok church)
> (get-church 0)
#Fun<lfe_eval.10.53503600>
> (== (zero) (get-church 0))
true
> (get-church 1000)
#Fun<lfe_eval.10.53503600>
```

Looks good so far. Let's check out the values:

```
> (church->int1 (get-church 1)))
1
> (church->int1 (get-church 50)))
50
> (church->int1 (get-church 100)))
100
> (church->int1 (get-church 2000)))
2000
> (church->int1 (get-church 10000)))
10000
>
```

That last one caused a little lag in the LFE REPL, but still quite impressive given the fact that it just applied so many thousands of lambdas!

How fortunate that we didn't have to type 10,000 `funcalls` (and the corresponding set of opening and closing parentheses 10,000 times).

5.8.3 Arithmetic

5.8.4 Logic

5.9 Practical Examples in Computing

5.9.1 A Simple Example

5.9.2 With an Accumulator

5.9.3 With Return Values

5.9.4 With Lists

5.10 Tail Calls in LFE

5.10.1 Tail Call Optimization

6 Checks, Errors, and Tests

6.1 Guards

6.2 Exception Handling

Erlang, and thus LFE, provide a means of evaluating expressions and not only handling normal results, but abnormal termination as well. This is done with `(try ... (catch ... ))`.

Note that `(try ... )` doesn't need to have a `(catch ...)`, however, since we will be exploring exception handling in this section, all of our examples will be using `(catch ...)`.

6.2.1 A Simple Case

TBD

6.3 EUnit

6.3.1 The Face of a Unit Test

6.3.2 Mixed Tests or Separate Modules?

6.3.3 Running Unit Tests

6.3.4 Distributing Code with Unit Tests

6.3.5 A Unit Test in Detail

6.3.5.1 Erlang EUnit Assert Macros

6.3.5.2 Fixtures: Setup and Cleanup

6.3.5.3 Generating Tests

6.3.6 Mocking with Meck

6.4 TDD

6.4.1 Creating an API and Writing Tests

6.4.2 Making Tests Pass

6.4.2.1 Factoring Out Common Test Logic

6.4.3 Testing the Server

6.4.4 Testing the Client

6.4.5 Cleaning Up After Tests

6.4.6 Handling Logged Errors

6.4.7 Resolving a Bug

6.4.8 Code Coverage

7 Processes and Servers

8 External Data

9 Additional Topics

9.1 Macros

9.2 Writing for Multi-Core