funlisp Documentation

Stephen Brennan

Mar 08, 2019

Contents:

funlisp is a small, simple, easy-to-embed scripting language written in C. It is a dialect of lisp, but doesn't adhere to any particular standard. This documentation contains information for those looking to learn the language, as well as those looking to embed and extend it.

You can get started by cloning the repository from GitHub, and then compiling the main program:

```
$ git clone https://github.com/brenns10/funlisp
$ make bin/funlisp
$ bin/funlisp
> (define hello (lambda () (print "hello world")))
<lambda hello>
> (hello)
hello world
```

If you do not have the a POSIX.2 system, or if you cannot install the Readline library (on which the main program depends), you can use the simpler (but less feature-rich) programs `bin/repl` and `bin/runfile`.

From there, you can evaluate some arithmetic and write some lambdas! See the table of contents below for guidance on the language, and how to embed the interpreter in your program. There is also some (limited) documentation on the internals of the interpreter, to help interested readers of my code (and probably me someday).

Visit the repository for more information, to view the source, to report bugs, or even to begin contributing!

CHAPTER 1

Language Guide

1.1 Hello World

As any good programming language documentation should, let's start with hello world:

```
(define main
  (lambda (args)
    (print "hello world")))
```

Here we see many of the normal expectations from a hello world: a function declaration and a print statement. `define` binds a name to any value you'd like. In this case, we're binding the symbol `main` to be a function.

Functions are declared using the `lambda` statement. They are enclosed in parentheses, and first contain a list of arguments, then an expression that is their implementation. Our `main` function simply prints a string. We can run this script using the `bin/runfile` utility bundled with funlisp:

```
$ make bin/runfile
# some make output
$ bin/runfile scripts/hello_world.lisp
hello world
```

1.2 Comments

Line comments are created in funlisp by starting them with a semicolon (`;`). Everything after that, until the end of the line, is ignored by the parser. Here are some examples:

```
; i'm a comment
(print "i'm some code" ; with a comment in the middle
)
```

1.3 Types

To accelerate our learning, try compiling the REPL. You might need to install libedit to build it.

```
$ make bin/repl
# some make output
$ bin/repl
>
```

Like most REPLs, this takes a line at a time, executes it, and prints the result. We'll use this for demonstrating some of funlisp's types.

```
> 5
5
> "blah"
blah
```

The two most obvious data types are the integer and string, as you can see above.

```
> 'hello
hello
> '(1 2 3)
(1 2 3 )
```

Above are two less obvious types, the "symbol" and the "list". Symbols are like strings, but they are used to bind names to values in lisp. Lists are the fundamental data structure of funlisp, used to hold both data and code! Both of those had quotes in front of them. Quoting in this way prevents funlisp from evaluating them, since symbols and lists behave a bit differently when they're evaluated:

```
> hello
error: symbol not found in scope
```

The symbol was looked up in the scope, but it turns out that there's no `hello` in scope. However, we know several things that are in the scope, from our hello world program:

```
> define
<builtin function define>
> lambda
<builtin function lambda>
> print
<builtin function print>
```

Neat! The symbols are names for builtin functions.

Lists behave in a special way when evaluated too. Funlisp tries to "call" the first item on the list, with the other items as arguments. We saw this with hello world as well:

```
> (1 2)
error: not callable!
> (print 2)
2
```

Turns out that "1" is not callable, but the function attached to the `print` symbol is!

1.4 Integer Functions

So, we've seen some neat tricks with the four basic builtin types. Now let's see how to manipulate integers:

```
> (+ 1 1)
2
> (- 5 1)
4
> (/ 4 3)
1
> (* 2 2)
4
```

Those basic arithmetic functions behave like any other function call. They look a bit odd because we expect arithmetic operators to be in the middle of an expression, but you'll get used to it!

```
> (= 5 5)
1
> (> 5 6)
0
> (<= 4 5)
1
```

Comparison operators look like that too. They return integers, which are used for conditionals in funlisp the same way that C does.

1.5 Control Flow

Speaking of control-flow, funlisp has a handy if statement:

```
> (if (= 5 4) (print "impossible") (print "boring"))
boring
```

Since we try to make everything in funlisp into an expression, if statements must have both a "value if true" and a "value if false". You cannot leave out the else.

Funlisp also has the `cond` statement, which allows you to test whether one of several conditions match. To test this out, we'll define some functions that use cond.

```
(define divisible
  (lambda (number by)
    (= number (* by (/ number by)))))

(define talk-about-numbers
  (lambda (number)
    (cond
       ((divisible number 2) (print "The number " number " is even!"))
       ((divisible number 3) (print "It may not be even, but " number " is divisible␣
↪by 3!"))
       (1 (print "The number " number " is odd, and not even divisible by 3.")))))
```

Don't worry about the function stuff, we'll revisit it later. Next is an example using these functions in the interpreter.

```
> (talk-about-numbers 6)
The number 6 is even!
```

(continues on next page)

(continued from previous page)

```
> (talk-about-numbers 5)
The number 5 is odd, and not even divisible by 3.
> (talk-about-numbers 9)
It may not be even, but 9 is divisible by 3!
```

The cond statement can use "1" as a default truth condition. However, it need not have a default case - when cond falls through, it returns nil, the empty list.

Funlisp doesn't currently have any form of iteration. However, it supports recursion, which is a very powerful way of iterating, and handling objects like lists.

1.6 Functions and Recursion

We've already seen the lambda syntax of creating functions for our hello world. Now let's check out some others:

```
> (define double (lambda (x) (* 2 x)))
<lambda function>
> (double 2)
4
```

We can recursively call our own function, for great good:

```
(define factorial
  (lambda (x)
    (if (= 0 x)
      1
      (* x (factorial (- x 1))))))
```

We can also use that capability to process a list of elements:

```
(define increment-all
  (lambda (x)
    (if (null? x)
      '()
      (cons (+ 1 x) (increment-all (cdr x))))))
```

Oops, looks like I've introduced you to cons and cdr. These are from a family of list processing functions that do the following:

- `(cons x l)` - put x at the beginning of list l
- `(car l)` - return the first item in l
- `(cdr l)` - return the elements after the first one in l

1.7 Scoping

Sometimes, you want to evaluate a value once, and save it for use multiple times. You can achieve this with `let`. Let has the syntax `(let BINDING-LIST expressions...)`. BINDING-LIST is a list of pairs, mapping a symbol to a value. Here's an example:

```
> (let ((x (+ 5 5)) (y (- x 2))) (+ x y))
18
```

Here the binding list contains the pair `(x (+ 5 5))`, mapping `x` to the evaluated value 10. The next binding, `(y (- x 2))` maps y to 8. Notice that the second binding may refer to the earlier one in the list. This can happen in reverse as well, if for example the first binding is a function:

```
> (let ((return1 (lambda () x)) (x 1)) (return1))
1
```

This needlessly complicated piece of work shows that the lambda bound to `return1` can access the name x, if it accesses it *after* x is bound.

1.8 Higher Order Functions

Now that we've incremented each item in a list, what if we want to decrement? We'd have to rewrite the whole function again, replacing the plus with a minus. Thankfully, we can do a better job, using `map`:

```
(define increment-all (lambda (l) (map (lambda (x) (+ 1 x) l))))
```

Map is a function which takes another function, as well as a list of items, and applies it to each item, returning the list of results.

We also have access to the reduce function, which applies that function to the list in pairs:

```
> (reduce + '(1 2 3))
6
```

1.9 Macros + Advanced Quoting

Funlisp provisionally supports some more powerful features of lisp: macros and quoting. Macros are similar to functions which take un-evaluated code (s-expressions), and return new code. Since code is represented as lists of symbols internally, it can be manipulated quite nicely with lisp code as well. The result is capable of defining syntatic shortcuts and other niceties. Here is an example that takes the common `(define function-name (lambda (args) ...))` pattern and shortens it into a more convenient construct: `(defun name (args) ...)`.

```
(define defun (macro (name args code)
  (list 'define name (list 'lambda args code))))
```

This uses the `(list)` builtin which constructs a list from evaluated expressions. The first argument, `name` is a symbol representing the name of the function. It gets substituted after the `define` symbol. The remaining arguments are part of the lambda declaration and get substituted later. The result is a macro that you can use to define functions like this:

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

However, you can see that it would get a bit cumbersome to use the `list` builtin everywhere, so a more advanced quoting system exists. In this "quasiquoting" system allows you to quote most of the data using a backtick, but "turn off" quoting for certain parts using a comma. Here is the same macro rewritten:

```
(define defun (macro (name args code)
  `(define ,name (lambda ,args ,code))))
```

You can see this is shorter and less error-prone. Simply write the code the way you'd like to see it, but use commas to substitute evaluated values.

Warning: While macros are usually evaluated at compile/parse time, funlisp currently evaluates them after the fact – just before the code is about to be run. Further, funlisp evaluates the macros *each time* they are used, rather than once only. The result is that macros are slightly less efficient than one might expect. But, you're not using funlisp for its efficiency, right?

1.10 The End

This is the end of our short guide to funlisp. Hopefully as time goes by, the language will grow, and maybe even obtain a standard library! But for now, funlisp remains sleek and small.

CHAPTER 2

Embedding

Funlisp is simply a static library. The 'fun' part comes when it is linked with a program that can make creative use of a scripting language. Funlisp comes with some useful example programs - a REPL (read-eval-print loop) and a script runner. In this section, we'll go over some basic concepts of funlisp, describe the implementation of these tools, and then go into how your program can integrate with the library.

2.1 Basic Components

To use the lisp interpreter, there are a few basic concepts to understand.

The interpreter has a *lisp_runtime* object associated with it. It holds all sorts of contextual information about the interpreter, especially having to do with garbage collection. You need to have an instance of a runtime in order use most of the rest of the library. You can create a runtime with *lisp_runtime_new()* and once initialized, you must destroy it with `lisp_destroy()`.

> **Warning:** Destroying a runtime also ends up garbage collecting all language objects created within that runtime, so if you want to access language objects, do it before destroying the runtime.

In order to run any code, you will need to have a global *lisp_scope*. This object binds names to funlisp values, including several of the critical built in functions and constructs of the language. You can use the function *lisp_new_default_scope()* to create a scope containing all the default language constructs (these are pretty important). If you ever need a new, empty scope, you can create it with *lisp_new_empty_scope()*.

Objects contained within the scope (and in fact, the scope itself) are all of type *lisp_value* - aka a funlisp object. This means they all share some common fields at their head, they all support some common operations (such as printing), and they are all managed by the garbage collector. So, you won't need to manually free a scope. Instead, use the garbage collector.

Speaking of this garbage collector, how does it work? Funlisp uses a mark and sweep garbage collector. This means that every so often the application must pause the program, mark all reachable language objects, and free everything that is unreachable. To do this, you need a "root set" of objects, which is typically your global scope. You should call

lisp_mark() on this root set, followed by *lisp_sweep()* on the runtime to free up all objects associated with your runtime, which are not reachable from your root set.

Warning: The garbage collector can make it easy to shoot yourself in the foot. It has no qualms with freeing a *lisp_value* that you'll be using on the very next line of code. So be sure to explicitly mark not only the objects that your scripts will need (this is usually just the scope), but also the objects that your C code would like to access, when you run the GC.

2.2 The REPL

With this knowledge, a REPL is pretty easy to make! Here is an outline of the steps:

1. First, create a language runtime and a global scope.
2. Read a line of input.
3. Parse the input. Parsed code is simply a *lisp_value* like any other language object.
4. Evaluate the input within the global scope.
5. If an error occurred, print it and continue. If nothing of interest is returned, do nothing. Otherwise, print the output, and a trailing newline.
6. Mark everything in scope, then sweep unreachable objects.
7. Repeat steps 2-7 for each line of input.
8. Destroy the language runtime to finish cleaning up memory.

Here is some basic code that demonstrates embedding a simple lisp interpreter, without any custom functions. It uses the `editline` implementation of the `readline` library for reading input (and allowing line editing).

```
/*
 * repl.c: very basic read-eval-print loop
 *
 * Stephen Brennan <stephen@brennan.io>
 */

#include <stdio.h>
#include <stdlib.h>

#include "funlisp.h"

int main(int argc, char **argv)
{
        char input[256];
        lisp_runtime *rt = lisp_runtime_new();
        lisp_scope *scope = lisp_new_default_scope(rt);

        (void)argc; /* unused parameters */
        (void)argv;

        for (;;) {
                lisp_value *value, *result;
                int bytes;

                printf("> ");
```

(continues on next page)

(continued from previous page)

```
                fflush(stdout);
                if (!fgets(input, sizeof(input), stdin))
                        break;

                bytes = lisp_parse_value(rt, input, 0, &value);
                if (bytes < 0) {
                        /* parse error */
                        lisp_print_error(rt, stderr);
                        lisp_clear_error(rt);
                        continue;
                } else if (!value) {
                        /* empty line */
                        continue;
                }
                result = lisp_eval(rt, scope, value);
                if (!result) {
                        lisp_print_error(rt, stderr);
                        lisp_clear_error(rt);
                } else if (!lisp_nil_p(result)) {
                        lisp_print(stdout, result);
                        fprintf(stdout, "\n");
                }
                lisp_mark(rt, (lisp_value*)scope);
                lisp_sweep(rt);
        }

        lisp_runtime_free(rt);
        return 0;
}
```

Notice here that *lisp_eval()* returns NULL in case of an error. If that happens, then you can use *lisp_print_error()* to print a user-facing error message, and *lisp_clear_error()* to clear the error from the interpreter state.

2.3 The Script Runner

Running a script is a slightly different story. Scripts generally define functions, including a main function, and then want to be executed with some command line arguments. So after reading and executing all the lines of code in a file, you'll want to execute the main function. We can use a nice utility for that, *lisp_run_main_if_exists()*. What's more, the `lisp_parse()` function used in the REPL can only parse a single expression. We have to use a more powerful option that can handle a whole file full of multiple expressions: *lisp_load_file()*.

Below is the source of the bundled script runner:

```
/*
 * runfile.c: Run a text file containing lisp code
 *
 * Stephen Brennan <stephen@brennan.io>
 */

#include <stdio.h>
#include <stdlib.h>

#include "funlisp.h"
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char **argv)
{
        FILE *input;
        lisp_runtime *rt;
        lisp_scope *scope;
        lisp_value *result;
        int rv;

        if (argc < 2) {
                fprintf(stderr, "error: expected at least one argument\n");
                return EXIT_FAILURE;
        }

        input = fopen(argv[1], "r");
        if (!input) {
                perror("open");
                return EXIT_FAILURE;
        }

        rt = lisp_runtime_new();
        scope = lisp_new_default_scope(rt);

        lisp_load_file(rt, scope, input);
        if (lisp_get_error(rt)) {
                lisp_print_error(rt, stderr);
                rv = 1;
                fclose(input);
                goto out;
        }
        fclose(input);

        result = lisp_run_main_if_exists(rt, scope, argc - 2, argv + 2);
        if (!result) {
                lisp_print_error(rt, stderr);
                rv = 1;
        } else {
                rv = 0;
        }
out:
        lisp_runtime_free(rt); /* sweeps everything before exit */
        return rv;
}
```

2.4 Calling C Functions From Lisp

Typically, an embedded interpreter will not be of much use to your application unless you can call into your application, or your application can call into the interpreter. The most straightforward way to add your own functionality to the interpreter is by writing a "builtin" - a C function callable from lisp. Builtins must have the following signature:

```
lisp_value *lisp_builtin_somename(lisp_runtime *rt,
                                  lisp_scope *scope,
                                  lisp_value *arglist,
```

(continues on next page)

(continued from previous page)

```
                                    void *user);
```

The scope argument contains the current binding of names to values, and the arglist is a list of arguments to your function, which **have not been evaluated**. These arguments are essentially code objects. You'll almost always want to evaluate them all before continuing with the logic of the function. You can do this individually with the *lisp_eval()* function, or just evaluate the whole list of arguments with the *lisp_eval_list()* function.

Warning: As we've noticed in the previous example programs, evaluating code can return NULL if an error (e.g. an exception of some sort) occurs. A well-behaved builtin will test the result of all calls to *lisp_eval()* and *lisp_call()* using the macro lisp_error_check() in order to propagate those errors back to the user. *lisp_eval_list()* propagates errors back, and so it should be error checked as well.

The one exception to evaluating all of your arguments is if you're defining some sort of syntactic construct. An example of this is the if-statement. The if statement looks like (if condition expr-if-true expr-if-false). It is implemented as a builtin function, which first evaluates the condition. Then, only the correct expression is evaluated based on that condition.

Finally, when you have your argument list, you could verify them all manually, but this process gets annoying very fast. To simplify this process, there is *lisp_get_args()*, a function which takes a list of (evaluated or unevaluated) arguments and a format string, along with a list of pointers to result variables. Similar to sscanf(), it reads a type code from the format string and attempts to take the next object off of the list, verify the type, and assign it to the current variable in the list. The current format string characters are:

- d: for integer
- l: for list
- s: for symbol
- S: for string
- o: for scope
- b: for builtin
- t: for type
- *: for anything

So, a format string for the plus function would be "dd", and the format string for the cons function is "**", because any two things may be put together in an s-expression. If nothing else, the *lisp_get_args()* function can help you verify the number of arguments, if not their types. When it fails, it returns false. It sets an internal interpreter error depending on what happened (too many arguments, not enough, types didn't matche, etc). You can handle this by simply returning NULL from your builtin. If argument parsing doesn't fail, your function is free to do whatever logic you'd like.

Finally, note that the signature includes a void *user parameter. This "user context" is specified when you register the builtin function, and passed back to you at runtime.

2.5 Basics of Lisp Types

In order to write any interesting functions, you need a basic idea of how types are represented and how you can get argument values out of the *lisp_value* objects. This is not a description of the type system (a future page in this section will cover that), just a list of available types and their values.

Warning: Currently, the structs defining these types are not in the public header file, `funlisp.h`. We're still working on the best way to expose types to the API.

The current types (that you are likely to use) are:

- `lisp_list`: contains a `left` and a `right` pointer.
  - `left` is usually a value of the linked list, and `right` is usually the next list in the linked list. However this isn't necessarily the case, because this object really represents an s-expression, and the right value of an s-expression doesn't have to be another s-expression.
  - The empty list is a special instance of `lisp_list`. You can get a new reference to it with `lisp_nil_new()` and you can check if an object is nil by calling `lisp_nil_p()`.
  - You can find the length of a list by using `lisp_list_length()`.
- `lisp_symbol`: type that represents names. Contains `sym`, which is a `char*`.
- `lisp_integer`: contains attribute `x`, an integer. Yes, it's allocated on the heap. Get over it.
- `lisp_string`: another thing similar to a symbol in implementation, but this time it represents a language string literal. The `s` attribute holds the string value.

There are also types for builtin functions, lambdas, scopes, and even a type for types! But you probably won't use them in your average code.

2.6 Adding Builtins to the Scope

Once you have written your functions, you must finally add them to the interpreter's global scope. Anything can be added to a scope with `lisp_scope_bind()`., but the name needs to be a `lisp_symbol` instance and the value needs to be a `lisp_value`. To save you the trouble of creating those objects, you can simply use `lisp_scope_add_builtin()`, which takes a scope, a string name, a function pointer, and a user context pointer.

Here is a code example that puts all of this together, based on the REPL given above.

```
/*
 * hello_repl.c: very basic read-eval-print loop, with builtins
 *
 * Stephen Brennan <stephen@brennan.io>
 */

#include <stdio.h>
#include <stdlib.h>

#include "funlisp.h"

/* (1) Here is our builtin function declaration. */
static lisp_value *say_hello(lisp_runtime *rt, lisp_scope *scope,
                             lisp_list *arglist, void *user)
{
        char *from = user;
        lisp_string *s;
        (void) scope; /* unused */

        if (!lisp_get_args(rt, arglist, "S", &s)) {
                return NULL;
        }
```

(continues on next page)

(continued from previous page)

```
        printf("Hello, %s! I'm %s.\n", lisp_string_get(s), from);

        /* must return something, so return nil */
        return lisp_nil_new(rt);
}

int main(int argc, char **argv)
{
        char input[256];
        lisp_runtime *rt = lisp_runtime_new();
        lisp_scope *scope = lisp_new_default_scope(rt);

        (void)argc; /* unused parameters */
        (void)argv;

        /* (2) Here we register the builtin once */
        lisp_scope_add_builtin(rt, scope, "hello", say_hello, "a computer", 1);
        /* (3) Now register the same function with a different context object */
        lisp_scope_add_builtin(rt, scope, "hello_from_stephen", say_hello, "Stephen",␣
↪1);

        for (;;) {
                lisp_value *value, *result;
                int bytes;

                printf("> ");
                fflush(stdout);
                if (!fgets(input, sizeof(input), stdin))
                        break;

                bytes = lisp_parse_value(rt, input, 0, &value);
                if (bytes < 0) {
                        /* parse error */
                        lisp_print_error(rt, stderr);
                        lisp_clear_error(rt);
                        continue;
                } else if (!value) {
                        /* empty line */
                        continue;
                }
                result = lisp_eval(rt, scope, value);
                if (!result) {
                        lisp_print_error(rt, stderr);
                        lisp_clear_error(rt);
                } else if (!lisp_nil_p(result)) {
                        lisp_print(stdout, result);
                        fprintf(stdout, "\n");
                }
                lisp_mark(rt, (lisp_value*)scope);
                lisp_sweep(rt);
        }

        lisp_runtime_free(rt);
        return 0;
}
```

In this example, we've added a builtin function, defined at (1). This function takes a string, and prints a greeting message. It uses its "user context" object as a string, to introduce itself. During startup, we register the builtin function once, with name "hello" (2). We provide it with a user context of "a computer." We register it again at (3), with the name "hello_from_stephen", and the user context "Stephen."

An example session using the builtin shows how the functions may be called from the REPL, and how the user context objects affect the builtin:

```
> (hello "Stephen")
Hello, Stephen! I'm a computer.
> (hello_from_stephen "computer")
Hello, computer! I'm Stephen.
> (hello 1)
error: expected a string!
> (hello 'Stephen)
error: expected a string!
```

2.7 Calling Lisp Functions From C

Just like it's possible to call C from lisp, you can call lisp from C. You've already seen plenty of examples of this, in the runfile tool. The runfile tool uses a helper function called `run_main_if_exists()`, but here's how it works.

1. First, you need to have a runtime and scope. If you're working in a builtin, you may have been given one. Otherwise, you'll probably have to create a default scope, and probably load in some user code too.
2. Second, you need to look up the function you want to call. Typically, you'll look it up in your scope using *`lisp_scope_lookup()`*.
3. Third, you need to build your arguments. Arguments are always a list of **un-evaluated** values. Since the lisp function will evaluate its arguments, you need to make sure they will end up being what you intended. For example, evaluating a list will result in calling the first item with the rest of the items as arguments. A fool proof way of ensuring that your data makes it thruogh unscathed is to quote it using *`lisp_quote()`*.
4. Fourth, you should use *`lisp_call()`* to call the function with your arguments.

Here's a full example!

```c
/*
 * call_lisp.c: example demonstrating how to call lisp functions from C
 *
 * Stephen Brennan <stephen@brennan.io>
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "funlisp.h"

int call_double_or_square(lisp_runtime *rt, lisp_scope *scope, int x)
{
        int rv;
        lisp_list *args;
        lisp_value *res;
        lisp_value *function = lisp_scope_lookup_string(rt, scope,
                "double_or_square");
        assert(function != NULL);
```

(continues on next page)

(continued from previous page)

```
        args = lisp_list_new(rt,
                (lisp_value *) lisp_integer_new(rt, x),
                (lisp_value *) lisp_nil_new(rt));
        res = lisp_call(rt, scope, function, args);
        assert(lisp_is(res, type_integer));
        rv = lisp_integer_get((lisp_integer *) res);
        printf("(double_or_square %d) = %d\n", x, rv);
        return rv;
}

int main(int argc, char **argv)
{
        lisp_runtime *rt;
        lisp_scope *scope;
        lisp_value *code;
        int bytes;

        (void) argc; /* unused parameters */
        (void) argv;

        rt = lisp_runtime_new();
        scope = lisp_new_default_scope(rt);
        bytes = lisp_parse_value(rt,
                "(define double_or_square"
                "  (lambda (x)"
                "    (if (< x 10)"
                "      (* x x)"
                "      (* x 2))))",
                0, &code
        );
        assert(bytes >= 0);
        lisp_eval(rt, scope, code);

        call_double_or_square(rt, scope, 5);
        call_double_or_square(rt, scope, 7);
        call_double_or_square(rt, scope, 9);
        call_double_or_square(rt, scope, 11);
        call_double_or_square(rt, scope, 13);

        lisp_runtime_free(rt);
        return 0;
}
```

A few things to note about this:

- We wrapped the lisp function in a C one. This is nice for getting rid of some complexity.
- There was no need to quote the integer, because integers evaluate to themselves.
- We had to build the argument list from scratch :(

This program produces exactly the output you'd expect:

```
$ bin/call_lisp
(double_or_square 5) = 25
(double_or_square 7) = 49
(double_or_square 9) = 81
```

(continues on next page)

(continued from previous page)

```
(double_or_square 11) = 22
(double_or_square 13) = 26
```

2.8 User Contexts

Usually, an application will want to have access to some of its own data. To that end, the embedding application may associate a "user context" with the *lisp_runtime* using *lisp_runtime_set_ctx()*. This context may be some global state, etc. Later (e.g. within a builtin function), the context may be retrieved with *lisp_runtime_get_ctx()*.

Furthermore, each builtin function may associate itself with a user context object as well (provided upon registration). This allows the same C function to be registered multiple times as multiple Funlisp functions. It also allows the C function to access additional context, which it may not have been able to get through the context object attached to the runtime.

2.9 Building Lists

Many parts of the funlisp API require constructing lists. As has been mentioned earlier, funlisp "list" types are singly linked lists. Each node's `left` pointer points to the data contained by the node, and the `right` pointer points to the next node in the list. `nil`, returned by *lisp_nil_new()*, is the empty list, and every list is terminated by it.

You have several options available to you for constructing lists. First is *lisp_list_new()*, which allows you to create a new list given a left and a right pointer. This may be used for constructing single list nodes. It may also be used to construct a list in reverse, starting with the last node and continuing to the first one. You may also directly set the left and right pointer of a list node, using *lisp_list_set_left()* and *lisp_list_set_right()* respectively.

> **Warning:** Note that lisp lists are not mutable. In Lisp, any modification to list results in a new one. Thus, you should never modify list nodes, unless you have just created them yourself and are constructing a full list.
>
> Also note that lisp lists' `left` and `right` pointers should never contain `null` – this will certainly cause a segmentation fault when they are used. However, you can set these pointers to `null` during the construction of a list, so long as the end result is a valid list with no nulls.

The simplest option is *lisp_list_append()*. This function allows you to construct a list forwards (rather than reverse). It requires double pointers to the head and tail of the list (where tail is the last non-nil item in the list). See the generated documentation for full information, but below is a fully working example of using it:

```c
/*
 * example_list_append.c: example demonstrating how to append to a list
 *
 * Stephen Brennan <stephen@brennan.io>
 */

#include <stdio.h>

#include "funlisp.h"

int main(int argc, char **argv)
{
```

(continues on next page)

(continued from previous page)

```
        lisp_runtime *rt;
        lisp_list *head, *tail;

        (void) argc; /* unused parameters */
        (void) argv;

        rt = lisp_runtime_new();

        head = tail = (lisp_list*) lisp_nil_new(rt);
        lisp_list_append(rt, &head, &tail, (lisp_value*)lisp_integer_new(rt, 1));
        lisp_list_append(rt, &head, &tail, (lisp_value*)lisp_integer_new(rt, 2));
        lisp_list_append(rt, &head, &tail, (lisp_value*)lisp_integer_new(rt, 3));
        lisp_print(stdout, (lisp_value*)head);
        fprintf(stdout, "\n");

        lisp_runtime_free(rt);
        return 0;
}
```

Finally, there are a few specialized options for constructing lists. *lisp_singleton_list()* allows you to construct a list with one item. *lisp_list_of_strings()* is useful for converting the argv and argc of a C main function into program arguments for a funlisp program.

2.10 Advanced Topics

From here, you should be equipped to get a fair amount out of funlisp. If you're looking for more, check the source code! In the future, I hope to write some documentation on implementation concepts to make this easier.

CHAPTER 3

API Reference

Contents

3.1 Funlisp Runtime

typedef struct *lisp_runtime* **lisp_runtime**

This is a context object, which tracks all language objects which have been created, and is used for garbage collection as well as holding any other information about your instance of the interpreter. The context can be created with *lisp_runtime_new()* and destroyed with *lisp_runtime_free()*. The context is passed to nearly every function in the library, and builtin functions receive it as well.

The context may contain a "user context" (simply a void pointer) that an embedding application may want its builtin functions to have access to. Context is added with *lisp_runtime_set_ctx()* and retrieved with *lisp_runtime_get_ctx()*.

*lisp_runtime** **lisp_runtime_new**(void)

Allocate and initialize a new runtime object. You must use *lisp_runtime_free()* to cleanup every runtime you create.

Return new runtime

void **lisp_runtime_set_ctx**(*lisp_runtime* * *rt*, void * *user*)
Set the user context of a *lisp_runtime*.

Parameters

- `rt`: runtime
- `user`: user context to set

void* **lisp_runtime_get_ctx**(*lisp_runtime* * *rt*)
Get the user context from a *lisp_runtime*.

Return the user context object

Parameters

- `rt`: runtime

void **lisp_runtime_free**(*lisp_runtime* * *rt*)
Clean up all resources and free a runtime object.

Warning This will invoke the garbage collector, freeing every language object associated with the runtime. Once calling this, ALL pointers to funlisp objects become invalid.

Parameters

- `rt`: runtime to free

void **lisp_enable_strcache**(*lisp_runtime* * *rt*)
Enable runtime support for caching strings.

When string caching is enabled, strings created with *lisp_string_new()* will be looked up from a cache first, and if they already exist, a cached object will be returned. This reduces the number of unique objects and memory objects, which both improves memory utilization and garbage collection times.

Parameters

- `rt`: runtime to enable string caching on

void **lisp_enable_symcache**(*lisp_runtime* * *rt*)
Enable runtime support for caching symbols.

When symbol caching is enabled, symbols created with *lisp_symbol_new()* will be looked up from a cache first, and if they already exist, a cached object will be returned. This reduces the number of unique objects and memory objects, which both improves memory utilization and garbage collection times.

Parameters

- `rt`: runtime to enable symbol caching on

void **lisp_disable_strcache**(*lisp_runtime* * *rt*)
Disable string caching.

Parameters

- `rt`: runtime to disable caching

void **lisp_disable_symcache**(*lisp_runtime* * *rt*)
Disable symbol caching.

Parameters

- `rt`: runtime to disable caching

3.2 Lisp Values

typedef struct *lisp_value* **lisp_value**
In funlisp, (almost) everything is a *lisp_value*. That is, it can be cast to a `lisp_value *` and operated on. Integers, Strings, Code, etc. The only thing which is not a *lisp_value* is the *lisp_runtime*.

typedef struct *lisp_type* **lisp_type**
A type object is a *lisp_value* containing operations that must be supported by every type of object. It is not garbage collected, and every *lisp_value* contains a pointer to its type object (even lisp_types themselves!).

The only external use for a type object is that you can use it wih *lisp_is()* to type check any *lisp_value*. Every type named lisp_X will have a corresponding type_X object available.

See *lisp_is()*

*lisp_type** **type_type**
Type object of *lisp_type*, for type checking.

See *lisp_is()*

void **lisp_print**(FILE * *f*, *lisp_value* * *value*)
Prints a string representing *value* to *f*. This output is not meant to contain all the information necessary to recreate *value*, just enough to give you an idea what it is.

Parameters

- `f`: file open for writing
- `value`: value to print

*lisp_value** **lisp_eval**(*lisp_runtime* * *rt*, *lisp_scope* * *scope*, *lisp_value* * *value*)
Evaluate the *lisp_value* in a given context. Since lisp values represent code as well as data, this is more applicable to some data structures than others. For example, evaluating a scope will not work. However, evaluating a symbol will look it up in the current scope, and evaluating list `l` will attempt to call `(car l)` with arguments `(cdr l)`.

When an error occurs during execution, this function returns NULL and sets the internal error details within the runtime.

Return the result of evaluating *value* in *scope*

Parameters

- `rt`: runtime associated with scope and value
- `scope`: the scope to use for evaluation (used when looking up symbols)
- `value`: the value to evaluate

Return Value

- `NULL`: when an error occurs

*lisp_value** **lisp_call**(*lisp_runtime* * *rt*, *lisp_scope* * *scope*, *lisp_value* * *callable*, *lisp_list* * *arguments*)
Call a callable object with a list of arguments. Many data types are not callable, in which case a NULL is returned and an error is set within the runtime.

Return the result of calling *callable* with args *arguments* in scope *scope*.

Parameters

- `rt`: runtime
- `scope`: scope in which we are being evaluated
- `callable`: value to call
- `arguments`: a *lisp_list* containing arguments (which *have not yet been evaluated*)

Return Value

- `NULL`: when an error occurs

int **lisp_compare**(*lisp_value* * *self*, *lisp_value* * *other*)

Compare two values for equality by value (not pointer). Generally this comparison should only be valid among objects of the same type.

Return comparison result as a boolean

Parameters

- `self`: value to compare with
- `other`: other value

Return Value

- `0`: NOT equal
- `nonzero`: equal

int **lisp_is**(*lisp_value* * *value*, *lisp_type* * *type*)

Perform type checking. Returns true (non-zero) when *value* has type *type*.

```
lisp_value *v = lisp_eval(rt, some_code, some_scope);
if (lisp_is(v, type_list)) {
    // do something based on this
}
```

Parameters

- `value`: value to type-check
- `type`: type object for the type you're interested in

Return Value

- `true`: (non-zero) if *value* has type *type*
- `false`: (zero) if *value* is not of type *type*

3.3 Lisp Scopes

typedef struct *lisp_scope* **lisp_scope**

Scope objects bind *lisp_symbol*'s to *lisp_value*'s. In order for the language to function correctly, the root scope needs to contain all of the language built-in features. You can obtain a scope like this by calling *lisp_new_default_scope()*, or you can create an empty one with *lisp_new_empty_scope()*.

*lisp_type** **type_scope**

Type object of *lisp_scope*, for type checking.

See *lisp_is()*

*lisp_scope** **lisp_new_default_scope** (*lisp_runtime * rt*)
Create a new scope containing the default builtins (lambda, define, arithmetic operators, etc). This is just a shortcut for using *lisp_new_empty_scope()* followed by lisp_scope_populate_builtin().

Return new default scope

Parameters

- `rt`: runtime

*lisp_scope** **lisp_new_empty_scope** (*lisp_runtime * rt*)
Create a new empty scope. This would be most useful when creating a new nested scope, e.g. for a function body.

Return new empty scope

Parameters

- `rt`: runtime

void **lisp_scope_populate_builtins** (*lisp_runtime * rt*, *lisp_scope * scope*)
Add all language defaults to a scope. This is critical for the language work, at all, since most language elements are implemented as builtin functions. This function is used internally by *lisp_new_default_scope()*.

Parameters

- `rt`: runtime
- `scope`: scope to add builtins too

void **lisp_scope_bind** (*lisp_scope * scope*, *lisp_symbol * symbol*, *lisp_value * value*)
Bind a symbol to a value in a scope.

Parameters

- `scope`: scope to define the name in
- `symbol`: symbol that is the name
- `value`: what the symbol is bound to

*lisp_value** **lisp_scope_lookup** (*lisp_runtime * rt*, *lisp_scope * scope*, *lisp_symbol * symbol*)
Look up a symbol within a scope. If it is not found in this scope, look within the parent scope etc, until it is found. If it is not found at all, return NULL and set an error within the interpreter.

Return value found, or a NULL when not found

Parameters

- `rt`: runtime
- `scope`: scope to look in
- `symbol`: symbol to look up

*lisp_value** **lisp_scope_lookup_string** (*lisp_runtime * rt*, *lisp_scope * scope*, char * *name*)
Lookup a name within a scope. Uses a string argument rather than a *lisp_symbol* object. Behavior is the same as *lisp_scope_lookup()*.

Return value found, or NULL when not found

Parameters

- `rt`: runtime
- `scope`: scope to look in
- `name`: string name to look up

3.4 Lisp Lists

typedef struct *lisp_list* **lisp_list**

Lisp is a list-processing language, and *lisp_list* is a building block for lists. It is somewhat mis-named, because it actually represents a s-expression, which is just a simple data structure that has two pointers: left and right. Normal lists are a series of s-expressions, such that each node contains a pointer to data in "left", and a pointer to the next node in "right". S-expressions may be written in lisp like so:

```
> '(left . right)
(left . right)
```

Normal lists are simply syntactic sugar for a series of nested s-expressions:

```
> '(a . (b . '()))
(a b )
```

*lisp_type** **type_list**

Type object of *lisp_list*, for type checking.

See *lisp_is()*

*lisp_list** **lisp_list_new** (*lisp_runtime* * *rt*, *lisp_value* * *left*, *lisp_value* * *right*)

Create a new list node with left and right value already specified. This interface only allows you to create lists from end to beginning.

Return newly allocated *lisp_list*

Parameters

- `rt`: runtime
- `left`: item to go on the left side of the s-expression, usually a list item
- `right`: item to go on the right side of the s-expression, usually the next *lisp_list* instance

*lisp_list** **lisp_singleton_list** (*lisp_runtime* * *rt*, *lisp_value* * *entry*)

Given a *lisp_value*, put it inside a list of size 0 and return it.

Return a singleton list

Parameters

- `rt`: runtime
- `entry`: item to put inside a list

*lisp_list** **lisp_list_of_strings** (*lisp_runtime* * *rt*, char ** *list*, size_t *n*, int *flags*)

Convert the array of strings into a lisp list of string objects.

Return *lisp_list* containing *lisp_string* objects

Parameters

- `rt`: runtime
- `list`: an array of strings
- `n`: length of the array
- `flags`: same flags passed to *lisp_string_new()*

int **lisp_list_length** (*lisp_list* * *list*)

Return the length of a list.

Return length of the list

Parameters

- `list`: list to find the length of

*lisp_value** **lisp_list_get_left** (*lisp_list * l*)
Retrieve the left item of a list node / sexp.

Return left item of list node

Parameters

- `l`: list to retrieve from

void **lisp_list_set_left** (*lisp_list * l*, *lisp_value * left*)
Set the left item of a list node.

Warning Lisp lists are not mutable! This should only be used during construction of lists.

Parameters

- `l`: list node to set
- `left`: item to set the left pointer to

*lisp_value** **lisp_list_get_right** (*lisp_list * l*)
Retrieve the right item of a list node / sexp

Return right item of list node

Parameters

- `l`: list to retrieve from

void **lisp_list_set_right** (*lisp_list * l*, *lisp_value * right*)
Set the right item of a list node.

Warning Lisp lists are not mutable! This should only be used during construction of lists.

Parameters

- `l`: list node to set
- `right`: item to set the right pointer to

void **lisp_list_append** (*lisp_runtime * rt*, *lisp_list ** *head*, *lisp_list ** *tail*, *lisp_value * item*)
Append *item*to the end of a list. This routine accepts double pointers to the head and tail of a list, so that it can update them if they change.

To create a list, you can append onto `nil`. After that, you may continue appending onto the list. Here is a complete example:

```
lisp_list *head, *tail;
head = tail = lisp_nil_new(rt);
lisp_list_append(rt, &head, &tail, (lisp_value*) lisp_integer_new(rt, 1));
lisp_list_append(rt, &head, &tail, (lisp_value*) lisp_integer_new(rt, 2));
lisp_list_append(rt, &head, &tail, (lisp_value*) lisp_integer_new(rt, 3));
lisp_print(stdout, (lisp_value*) head);
// prints (1 2 3 )
```

Parameters

- `rt`: runtime
- `head`: double pointer to the first item in the list (or nil, if appending to an empty list)

- `tail`: double pointer to the last non-nil item in the list (or nil, if appending to an empty list)
- `item`: the value to append

*lisp_value** **lisp_nil_new**(*lisp_runtime* * *rt*)

Return a nil instance. Nil is simply a "special" *lisp_list*, with left and right both set to NULL. It is used to terminate lists. For example, the list `'(a b)` is internally: `lisp_list(a, lisp_list(b, lisp_list(NULL, NULL)))`

Note This function is named "new" for uniformity. However, it does't actually allocate a "new" nil value every time. Instead, each *lisp_runtime* has a singleton nil instance, which is never garbage collected.

Return the nil value

Parameters

- `rt`: runtime

int **lisp_nil_p**(*lisp_value* * *l*)

Return true if the *lisp_value* is "nil" (an empty list).

Parameters

- `l`: value to check

Return Value

- `1`: (true) if l is nil
- `0`: (false) if l is non-nil

3.5 Lisp Types

typedef struct lisp_text **lisp_symbol**

Symbols are tokens (non-numeric, non parentheses) which occur in funlisp code, not surounded by double quotes. For example, in the following code:

```
(define abs
  (lambda (x)
    (if (< x 0)
      (- 0 x)
      x)))
```

The symbols are: define, abs, lambda, x, if, and <.

typedef struct *lisp_integer* **lisp_integer**

lisp_integer contains an int object of whatever size the C implementation supports.

typedef struct lisp_text **lisp_string**

This is a string (which occurs quoted in lisp source)

typedef struct *lisp_builtin* **lisp_builtin**

This data structure contains a native C function which may be called by funlisp code. The C function must be of type *lisp_builtin_func*.

typedef struct *lisp_lambda* **lisp_lambda**

Data structure implementing a lisp lambda function.

*lisp_type** **type_symbol**

Type object of *lisp_symbol*, for type checking.

See *lisp_is()*

*lisp_type** **type_integer**
Type object of *lisp_integer*, for type checking.

See *lisp_is()*

*lisp_type** **type_string**
Type object of *lisp_string*, for type checking.

See *lisp_is()*

*lisp_type** **type_builtin**
Type object of *lisp_builtin*, for type checking.

See *lisp_is()*

*lisp_type** **type_lambda**
Type object of *lisp_lambda*, for type checking.

See *lisp_is()*

*lisp_string** **lisp_string_new** (*lisp_runtime* * *rt*, char * *str*, int *flags*)
Return a new string. This function takes a "safe" approach, by copying your string and using the copy. The pointer will be owned by the interpreter and freed when the *lisp_string* object is garbage collected. This is roughly equivalent to duplicating the string using strdup(), and then creating a new owned string with that pointer.

Note This is also safe to use with string literals, but it is not the most efficient way, since the string gets copied.

Return a new *lisp_string*

See *LS_CPY*

See *LS_OWN*

Parameters

- `rt`: runtime
- `str`: string to copy and use in an owned string
- `flags`: flags related to copying and ownership of *str*

char* **lisp_string_get** (*lisp_string* * *s*)
Return a pointer to the string contained within a *lisp_string*. The application must **not** modify or free the string.

Return the contained string

Parameters

- `s`: the lisp string to access

*lisp_symbol** **lisp_symbol_new** (*lisp_runtime* * *rt*, char * *string*, int *flags*)
Return a new symbol. This function will copy the *string* and free the copy it on garbage collection (much like *lisp_string_new()*).

Return the resulting symbol

See *LS_CPY*

See *LS_OWN*

Parameters

- `rt`: runtime
- `string`: the symbol to create

- `flags`: flags related to copying and ownership of *string*

char* **lisp_symbol_get** (*lisp_symbol* * *s*)
Return the string contained in the symbol.

Return the string contained in the symbol

Parameters

- `s`: the symbol to retrieve the string from

*lisp_integer** **lisp_integer_new** (*lisp_runtime* * *rt*, int *n*)
Create a new integer.

Return newly allocated integer

Parameters

- `rt`: runtime
- `n`: the integer value

int **lisp_integer_get** (*lisp_integer* * *integer*)
Retrieve the integer value from a *lisp_integer*.

Return the int value

Parameters

- `integer`: *lisp_integer* to return from

LS_CPY
Flag instructing string/symbol creation routines that they should copy the string buffer itself, and use the copy rather than the original argument. This could be useful in case callers would like to free the string after creating a lisp symbol/string from it.

Warning If you use this without *LS_OWN*, you will have memory leaks, because funlisp will allocate a new string, but never free it.

See *LS_OWN*

See *lisp_string_new()*

LS_OWN
Flag instructing string/symbol creation routines that when the wrapper object (lisp_string/lisp_symbol) is freed, the string itself should also be freed. Put in other words, the lisp context should "own" the reference to the string.

When this is unset, we expect that the string exists for the duration of the lisp_context, and we do not free it under any circumstances. This is good for C string literals, or strings that you know you will keep around for longer than the *lisp_runtime* will exist.

Warning If you unset this, but set *LS_CPY*, you will have memory leaks!

See *LS_CPY*

See *lisp_string_new()*

3.6 Builtin Functions

typedef *lisp_value** **(* lisp_builtin_func)** (*lisp_runtime* *, *lisp_scope* *, *lisp_list* *, void *)
A built-in function. Takes four arguments:

1. The *lisp_runtime* associated with it. This may be used to retrieved the runtime's user context object (see *lisp_runtime_get_ctx()*).
2. The *lisp_scope* this function is being called executed within. Most builtin functions will want to evaluate this with *lisp_eval_list()*.
3. The arguments to this function, as a *lisp_list*. These may or may not have been evaluated, depending on whether `evald` was set when creating the builtin object.
4. The user context associated with this builtin.

*lisp_builtin** **lisp_builtin_new**(*lisp_runtime* * *rt*, char * *name*, *lisp_builtin_func* *call*, void * *user*, int *evald*)

Create a new *lisp_builtin* from a function pointer, with a given name.

Warning Namse of builtins are not garbage collected, since they are almost always static. If you need your name to be dynamically allocated, you'll have to free it after you free the runtime.

Return new builtin object

Parameters

- `rt`: runtime
- `name`: name of the builtin. the interpreter will never free the name!
- `call`: function pointer of the builtin
- `user`: a user context pointer which will be given to the builtin
- `evald`: non-zero if arguments should be evaluated before being given to this builtin. Zero if arguments should be given as-is.

void **lisp_scope_add_builtin**(*lisp_runtime* * *rt*, *lisp_scope* * *scope*, char * *name*, *lisp_builtin_func* *call*, void * *user*, int *evald*)

Shortcut to declare a builtin function. Simply takes a function pointer and a string name, and it will internally create the *lisp_builtin* object with the correct name, and bind it in the given scope.

Parameters

- `rt`: runtime
- `scope`: scope to bind builtin in
- `name`: name of builtin
- `call`: function pointer defining the builtin
- `user`: a user context pointer which will be given to the builtin
- `evald`: non-zero if arguments should be evaluated before being given to this builtin. Zero if arguments should be given as-is.

*lisp_list** **lisp_eval_list**(*lisp_runtime* * *rt*, *lisp_scope* * *scope*, *lisp_list* * *list*)

Given a list of arguments, evaluate each of them within a scope and return a new list containing the evaluated arguments. This is most useful for implementing builtin functions.

Return list of evaluated function arguments

Parameters

- `rt`: runtime
- `scope`: scope to evaluate within
- `list`: list of un-evaluated function arguments

Return Value

- `NULL`: if an error occured during evaluation

*lisp_value** **lisp_progn** (*lisp_runtime * rt*, *lisp_scope * scope*, *lisp_list * l*)

Given a list of *lisp_value*'s, evaluate each of them within a scope, returning the last value. This is similar to *lisp_eval_list()*, but rather than constructing a full list of results, it merely returns the last one. It is used in the `progn` builtin, but it also is useful for doing things like evaluating everything in a file or allowing implimenting "implicit progns".

Parameters

- `rt`: runtime
- `scope`: scope
- `l`: list of expressions to evaluate

int **lisp_get_args** (*lisp_runtime * rt*, *lisp_list * list*, char * *format*, ...)

Given a list of function arguments, perform type checking and verify the number of arguments according to a format string. The following formats are recognized:

```
d - integer
l - list
s - symbol
S - string
o - scope
e - error
b - builtin
t - type
* - anything
R - Rest of arguments
```

As an example, a function which takes an integer and a string, and prints the string N times, might use the format string `dS`.

The remaining variadic arguments are pointers to object pointers, and they will be assigned as each argument is parsed. EG:

```
lisp_integer *arg1;
lisp_string *arg2;
lisp_get_args(rt, args, "dS", &arg1, &arg2);
```

Note The format code 'R' is special and deserves some more attention. When used, it immediately ends argument processing, so it should only be used at the end of a format string. It will resolve to the remaining unprocessed arguments as a list, provided that there is at least one (i.e. R will fail if the rest of the args is an empty list).

Parameters

- `rt`: runtime
- `list`: Argument list to type check and count
- `format`: Format string
- `...`: Destination pointer to place results

Return Value

- `1`: on success (true)
- `0`: on failure (false)

3.7 Embedding Tools

int **lisp_parse_value** (*lisp_runtime* * *rt*, char * *input*, int *index*, *lisp_value* ** *output*)

Parse a *single* expression from *input*, starting at *index*. Sets the result as a *lisp_value* in *output*. Return the number of bytes parsed from *input*.

When a parse error occurs, the return value is negative, and *output* is set to NULL. The error code and line number are set in the runtime, and may be retrieved with *lisp_get_error()*.

When the string contains no expression (only whitespace or comments), the return value will still be non-negative. *output* will be set to NULL. This situation is typically not an error, either meaning empty REPL input or the end of the file you are parsing.

Return number of bytes processed from *input*

Parameters

- `rt`: runtime to create language objects in
- `input`: string to parse
- `index`: position in *input* to start parsing
- `output`: pointer to `lisp_value **` where we store the parsed expression.

Return Value

- `-1`: when an error occurs during parsing

*lisp_value** **lisp_parse_progn** (*lisp_runtime* * *rt*, char * *input*)

Parse every expression contained in *input*. Return the parsed code as a list, with the first element being the symbol `progn`, and the remaining elements being the parsed code. This may be evaluated using *lisp_eval()*.

When a parse error occurs, NULL is returned. Note that parse errors typically occur after memory allocation has occurred. Memory allocated by this function is not cleaned up on error, and must be garbage collected.

Note that if the string is entirely empty, or only contains comments, then the progn will be empty, which currently causes an exception when evaluated.

Return the code, fully parsed, within a progn block

Parameters

- `rt`: runtime
- `input`: string to parse

Return Value

- `NULL`: when an error occurs (see *lisp_print_error()*)

*lisp_value** **lisp_parse_progn_f** (*lisp_runtime* * *rt*, FILE * *file*)

Parse every expression contained in *file*, and return the parsed code as a `progn` block. This function behaves same as *lisp_parse_progn()*. Additional errors may be raised due to I/O errors on *file*.

Return the code, fully parsed, within a progn block

Parameters

- `rt`: runtime
- `file`: file to parse

Return Value

- `NULL`: when an error occurs (see *lisp_print_error()*)

*lisp_value** **lisp_load_file** (*lisp_runtime * rt*, *lisp_scope * scope*, FILE * *input*)

Parse a file and evaluate its contents. This is roughly equivalent to:

```
lisp_value *progn = lisp_parse_progn_f(rt, scope, input)
lisp_eval(rt, scope, progn);
```

Return the result of evaluating the last item

Parameters

- `rt`: runtime
- `scope`: scope to evaluate within (usually a default scope)
- `input`: file to load as funlisp code

Return Value

- `NULL`: on empty file, or file read error

*lisp_value** **lisp_run_main_if_exists** (*lisp_runtime * rt*, *lisp_scope * scope*, int *argc*, char ** *argv*)

Lookup the symbol `main` in the scope, and run it if it exists. Calls the function with a single argument, a *lisp_list* of program arguments. argc and argv should not include the main executable (just the script name and args).

Return result of evaluation

Parameters

- `rt`: runtime
- `scope`: scope to find main in
- `argc`: number of arguments
- `argv`: NULL-terminated argument list

Return Value

- `a`: nil list when there is no main symbol
- `NULL`: on error

void **lisp_mark** (*lisp_runtime * rt*, *lisp_value * v*)

Mark an object as still reachable or useful to the program (or you). This can be called several times to mark many objects. Marking objects prevents the garbage collector from freeing them. The garbage collector performs a breadth first search starting from your marked objects to find all reachable language objects. Thus, marking an object like a *lisp_scope* will save all symbols and language objects contained within it, from being freed. Normal use is to mark and sweep each time you've evaluated something:

```
lisp_value *result = lisp_eval(rt, scope, some_cool_code);
lisp_mark(rt, (lisp_value*) scope);
lisp_mark(rt, result);
lisp_sweep(rt);
```

Warning Be explicit about marking. If we had left out the third line of the code sample above, there's a good chance that `result` would have been freed when *lisp_sweep()* was called.

Parameters

- `rt`: runtime

- `v`: value to mark as still needed. This value, and all values reachable from it, are preserved on the next *`lisp_sweep()`* call.

void **`lisp_sweep`**(*lisp_runtime* * *rt*)

Free every object associated with the runtime, which is not marked or reachable from a marked object.

Parameters

- `rt`: runtime

*lisp_list** **`lisp_quote`**(*lisp_runtime* * *rt*, *lisp_value* * *value*)

Return *value*, but inside a list containing the symbol `quote`. When this evaluated, it will return its contents (*value*) un-evaluated.

This function is used during parsing, to implement the single-quote syntax feature. For example `'(a b c)`, evaluates to the list containing a, b, and c, rather than calling a on b and c. This is because the expression is transparently converted to the more verbose `(quote (a b c))`.

Return value but quoted

Parameters

- `rt`: runtime
- `value`: value to return quoted

3.8 Error Handling

`enum error::lisp_errno`

Values:

`LE_ERROR` =1

`LE_EOF`

`LE_SYNTAX`

`LE_FERROR`

`LE_2MANY`

`LE_2FEW`

`LE_TYPE`

`LE_NOCALL`

`LE_NOEVAL`

`LE_NOTFOUND`

`LE_EXIT`

`LE_ASSERT`

`LE_VALUE`

`LE_MAX_ERR`

`const char* lisp_error_name[LE_MAX_ERR]`

const char* const **`lisp_version`**

*lisp_value** **lisp_error**(*lisp_runtime* * *rt*, enum lisp_errno *errno*, char * *message*)
Raise an error in the interpreter and return NULL.

This function is meant to be used within code that implements builtins. When an error condition is reached, functions may simply do something like this:

```
if (some_error_condition())
    return lisp_error(rt, LE_ERROR, "you broke something");
```

Return NULL

Parameters

- `rt`: runtime
- `errno`: error number, for easy programatic acccess
- `message`: message to show the user

void **lisp_dump_stack**(*lisp_runtime* * *rt*, *lisp_list* * *stack*, FILE * *file*)
Dump the execution stack to a file. This is useful if you want to print a stack trace at your current location. This functionality can also be accessed via the `dump-stack` builtin function.

Parameters

- `rt`: runtime
- `stack`: When NULL, the runtime's execution stack is used. When non-NULL, the *stack* argument is used to specify what stack to dump.
- `file`: where to dump stack trace to

void **lisp_print_error**(*lisp_runtime* * *rt*, FILE * *file*)
Prints the last error reported to the runtime, on *file*. If there is no error, this prints a loud BUG message to FILE, indicating that an error was expected but not found.

Parameters

- `rt`: runtime
- `file`: file to print error to (usually stderr)

char* **lisp_get_error**(*lisp_runtime* * *rt*)
Returns the error text of the current error registered with the runtime.

Return error string

Parameters

- `rt`: runtime

enum lisp_errno **lisp_get_errno**(*lisp_runtime* * *rt*)
Returns the error number of the current error registered with the runtime.

Return error number

Parameters

- `rt`: runtime

void **lisp_clear_error**(*lisp_runtime* * *rt*)
Clears the error in the runtime.

Parameters

- `rt`: runtime

lisp_error_check(value)

A macro for error checking the return value of a *lisp_eval()* or *lisp_call()* function. This will return NULL when its argumnet is NULL, helping functions short-circuit in the case of an error.

```
lisp_value *v = lisp_eval(rt, my_code, my_scope);
lisp_error_check(v);
// continue using v
```

Parameters

- `value`: value to error check

CHAPTER 4

Advanced Topics

This should help you if you want to understand the inner workings, implement your own types, or contribute to funlisp.

4.1 Type System

As previous pages have mentions, lisp objects rely on a mark and sweep garbage collection system to manage their lifetimes. This page describes how the type system is implemented, which includes the garbage collection implementation.

In Java, everything is an object. In this language, everything is a *lisp_value*. This means two things:

1. Every object contains a `type` pointer.
2. Every object has a variable to store its `mark`.
3. Every object has a pointer to the object allocated after it, forming a linked list.

Every type declares these using the `LISP_VALUE_HEAD` macro, like so:

```
typedef struct {
  LISP_VALUE_HEAD;
  int x;
} lisp_integer;
```

You can cast pointers to these objects to `lisp_value*` and still access the type object. All objects are passed around as `lisp_value*`.

In order to allow objects to be treated differently based on their type (but in a generic way to calling code), we use the type object.

A type object is just another object, with type *lisp_type*. However, it is NOT managed by the garbage collector. It contains the string name of a type, along with pointers to implementations for the following functions: print, new, free, eval, call, and expand. Therefore, if you have `lisp_value *object` and you want to print it, you can do:

```
object->type->print(stdout, object);
```

Unfortunately that's very verbose. To simplify, each of the functions implemented by a type object has an associated helper function. So you can instead do:

```
lisp_print(stdout, object);
```

Which, under the hood, simply gets the type object and uses the print function. But there's no magic or switch statements involved here–we're simply using the type object.

All of the type object operations have their own helper functions like print. The advantage to doing this, besides less verbose code, is that shared operations can be done together. For example, the `lisp_new()` function does some garbage collection related operations that none of the type object `new()` methods need to know about.

As a result of these type objects, it's not too difficult to add a type to the language! All you need to do is declare a struct for your type, implement the basic functions, and create a type object for it. The type object must implement the following operations:

- print: writes a representation of the object to a file, without newline
- new: allocates and initializes a new instance of the object
- free: cleans up and frees an instance
- expand: creates an *iterator* of ALL references to objects this object owns (see the *garbage collection documentation*)
- eval: evaluate this in a scope
- call: call this item in a scope with arguments

4.2 Iterators

Iterators are one of several internal APIs used by funlisp but not exposed in the public API. They are similar to the lazy evaluation constructs supported by other languages (e.g. generators in Python), but for C. Here is what an iterator looks like:

```
struct iterator {
    void *ds;        /* the container data structure */
    int index;       /* zero-based index for the iterator */
    int state_int;   /* some state variables that may help */
    void *state_ptr;

    /* do we have a next item? */
    bool (*has_next)(struct iterator *iter);

    /* return the next item (or null) */
    void *(*next)(struct iterator *iter);

    /* free resources held by the iterator */
    void (*close)(struct iterator *iter);
};
```

The iterator holds some state, and three operations:

- has next: tell us whether the iterator can be advanced
- next: advance the iterator and return the next value
- close: free any resources held by the iterator

Any data structure could implement these (e.g. an array, a linked list, a hash table, or others), and could be iterated over by code that knows nothing about the underlying implementation. Iterators can also be composed in interesting ways, with some of the helpers given below:

- empty iterator: return an empty iterator
- single value iterator: return an iterator that contains one item
- concatenate: given an array of iterators, yield from each of them until they run out

4.3 Garbage Collection

Garbage collection is implemented in Lisp using a mark-and-sweep system. In this system, values are first marked as to whether they are "reachable", and then all objects which are unreachable can be freed.

To implement this, we need two key components. First, we need a way to mark reachable objects. Second, we need a way to track all objects so that we can find the unreachable ones to free. It turns out the second one is pretty easy: just maintain a linked list of values as they are created. This logic is nicely handled inside of `lisp_new()`.

The second one is trickier. My strategy was for every type to implement its own `expand()` operation. This is a function which returns an iterator that yields every reference the object contains. For example, strings and integers yield empty iterators. However, lists yield their left and right objects (if they exist). Scopes yield every symbol and then every value stored within them. They also yield a reference to their parent scope, if it exists.

The mark operation then simply performs a breadth-first search. It starts with a single value, marks it "black", then uses the expand operation. It goes through each value, adding all "white" items to the queue, marking them "grey" as it goes. Then it chooses the next item in the queue and does the same operation, until the queue is empty.

To do the breadth-first search, we use a "ring buffer" implementation, which implements a circular, dynamically expanding double-ended queue. It is quite simple and useful. It can be found in `src/ringbuf.c`.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

L