The LAC LISP Library Manual

for version 0.1.0.

Gianluca Guida

This manual is for the LAC LISP Library (version 0.1.0).

Table of Contents

1 Introduction

This manual is an introduction and a reference for the LAC LISP Library (LAC hereinafter).

LAC is a simple LISP implementation. Its interpreter is meant be embedded in your application (see Chapter 6 [Adding LAC to your program], page 9).

LAC is also extensible: you can add new atom types and execute your own functions written in lower level, compiled languages (see Chapter 5 [Extending the language], page 8).

For more information about the language itself, see Chapter 3 [The LAC Language], page 6.

2 Using LAC

LAC is a library that implements a LISP interpreter with a system library that contains many useful functions to handle lists, symbols and integers.

The interpreter itself can be extended to add features, and this is the real power of LAC.

The rest of the chapter will introduce two common ways of using LAC: *REPL extension* and *embedding*. The difference between the two is that in the former the interpreter has control of the application, while in the latter the LISP interpreter can be called at will by the application to execute commands.

Please note that this chapter is just introductory. For a more in-depth discussion of the topic, see Chapter 6 [Adding LAC to your program], page 9.

2.1 REPL Extension

Suppose you have a low-level library that implements some features in a specific domain, and you need a language capable of using those features to create small scripts, write prototypes, explore the library capabilities.

What you can do is to first use LAC to create a REPL[1] and then extend the LAC language with elements from your library.

2.1.1 Writing a REPL for LAC

Writing a REPL in LAC is extremely simple. The C program in Figure 2.1 produces an interactive command line REPL capable of executing LAC statements, e.g.:

```
LAC>(+ 2 2)
4
```

This will be essentially an interpreter for a basic LISP-like language, and you will be able to do integer, list and symbol processing with ease. You will also be able to create macros to extend the language to your needs. See Chapter 3 [The LAC Language], page 6.

[1] REPL stands for *Read*, *Eval*, *Print Loop*. It is used to describe the core functionality of an interpreter: reading an input, evaluating the value of the input and printing the calculated output value.

```
#include <stdio.h>
#include <readline/readline.h>
#include <lac.h>

int main (int argc, char *argv[])
{
  lenv_t *null_env;

  /* Initialize LAC (create NULL environment) */
  null_env = lac_init();
  if (null_env == NULL)
    {
      fprintf(stderr, "Could not initialize lac");
      return -1;
    }

  /* REPL loop. */
  while (1)
    {
      char *buf;
      lreg_t res;

      /* Read an input line */
      buf = readline ("LAC>");
      if (buf == NULL)
        break;

      /* Evaluate S-Expressions. */
      res = sexpr_eval_string (buf, null_env);

      /* Print the result */
      sexpr_fprint(stdout, res);
      printf("\n");
    }

  /* Exit REPL */
  return 0;
}
```

Figure 2.1: A simple REPL for the LAC library

2.1.2 Extending the interpreter

The usefulness of LAC emerges when you add support for your own libraries to this language, and you can do this in two ways: adding *Native Procedures* and *Atom Types*.

2.1.2.1 Native Procedures

Procedures in LISP-like languages are similar to what functions are in C. There are some major differences with C functions, for example in LAC procedures are *first-class functions*[2] and that they can be defined and modified at runtime.

A *Native Procedure* is a procedure written in a low level language, opaque to the interpreter (cannot be inspected or changed). LAC defines a standard C interface for the *low-level procedures*, and functions to hook this function to symbols in the LISP machine.

[2] This essentially means that a procedure (and its scope) can be treated as a value, passed as a parameter and returned by another procedure

Let's clarify the previous sentence with an example: the following C function is a *Native Procedure*.

```
LAC_API static lreg_t
proc_hello (lreg_t args, lenv_t * argenv, lenv_t * env)
{
  fprintf(stdout, "Hello, World!");
  return NIL;
}
```

It is a simple procedure that, ignoring all arguments passed, will always print to standard output the string "Hello, World!", and return the empty value NIL.

LAC_API is mostly needed to tell the compiler to follow certain rules (mostly alignment) that will ensure that it can be processed internally by the interpreter.

Adding this line to the code in Figure 2.1 is not enough though to access it. We need to associate this function to a symbol, and this is called *registering*.

```
/* PRE: lac_init() has already been
   called, and has returned null_env.  */
lac_extproc_register (null_env, "hello", proc_hello);
```

This will hook the procedure defined above to the symbol *HELLO*. If we add to our code and run the REPL again, we will have a new command available in our interpreter. Note that symbols are case insensitive.

```
LAC>HELLO
<#LLPROC>
LAC>(HELLO)
Hello, World!()
LAC>(hello)
Hello, World!()
LAC>
```

The function, as you can see above, first prints out to the screen the word "Hello, World!", and then returns, NIL, the void value of LISP, which is also the empty list, and this gets printed by the *Print* phase of the REPL as `()`.

To complete the example, we can change `proc_hello` to return a string instead of printing it to screen directly:

```
LAC_API static lreg_t
proc_hello(lreg_t args, lenv_t * argenv, lenv_t * env)
{
  return lac_string_box("Hello, World!");
}
```

If we register the above function, the REPL will return a string *atom*:

```
LAC>(hello)
"Hello, World!"
LAC>
```

2.1.2.2 Adding Atom Types

2.2 Embedding LAC

3 The LAC Language

3.1 The LAC System Library

4 The LAC Library

5 Extending the language

6 Adding LAC to your program

7 GNU Free Documentation License

Index