The Salmon Lisp Bible

Kai Daniel Gonzalez

October 12, 2022

Contents

1 The Start of It All

Salmon Lisp is a very powerful general purpose programming language with a very rich standard library for creating everything from basic I/O programs. To resource intensive algorithms.

This book will teach you everything you need to know about Salmon including a detailed explanation of every function in the current standard library. So without further-ado, let's begin shall we?

2 The Beauty of The 'sexpr'

An 'sexpr' (also known as an '*S-expression*') is a certain type of grammar which contains parenthesis and is usually denoted using a type of Top-down parser. (although, not always.)

A sample S-expression looks something like:

```
(function [forms . . .])
```

Which is essentially the entire base of the language. To print "Hello, world" in SLisp (*Salmon Lisp*) would be:

```
(print Hello, world!")
```

In Salmon, you denote the beginning of an expression using the **LEFT PARENTHESIS** symbol: '('

2.1 The Idea of Root Expressions

When you run a statement, there must be a root statement. Whether it's a conditional block or a function, there should always be a root statement. For example, if you have a statement like:

```
(case (= variable 1)
        (print "Variable is one!")
        (print "Variable is not one!")))
```

The root statement would be the 'case' call. Therefore, it will not return anything. On that note there are two types of statements, **root** statements, and **embedded** statements. Root statements do not return anything, they are not meant to and are only ran at the start of a program, kind of like the fire-starter.

If you were to print '1 + 1', it would return '2', like any language would. But if you were to **just add 1 + 1** like the following:

```
(+ 1 1)
```

It would not return anything. As it should *not.* This is expected behavior, but even though it's meant to return something, it would not, the value would be discarded. So even though it's expected to return something, it's value's nowhere to be found in memory because it was discarded.

Values are only used if a statement is used within a statement, like the following:

```
(print (+ 1 (+ 1 1)))
```

At the beginning, the 'print' function may not return a value, but we would not know if we ran it as a root statement, to save it's value you would use something like *set* to save and return it's value.

```
(set print-output (print "hello, world!"))
```

When you set the variable, it will run the print function that is inside of the *[value]* field and set that as it's value. Which should 100% be *nil.*

3 Function defaults

By default, SLisp contains a very strict set of functions for the user to experiment with, and a few keywords. **progn**, **set**, and **print** just to name a few.

3.1 Keywords Versus Functions - Convenience

While hacking SLisp, ninety-five percent of the time you won't ever need to create any sort of keywords (which are stored in the SalmonEnvironment.pluginKeywords array)

Keywords are as low-level as it gets, they don't get the arguments as Salmon values, like regular functions do, instead, they get the arguments as **RAW** strings, meaning that a keyword like **progn** has multiple different statements as it's arguments, and they are not evaluated.

4 D API

The D API is open to anyone, for anyone to create their own distributions of SLisp if they so need to, with it's own custom standard library, if they need to.

This book DOES NOT go over the basics of the D Programming Language. If you read this expecting to learn the D programming language, you will not find it here. Please view the D homepage.

4.1 Including Headers

First of all, the SLisp headers are compliant with the SLisp library, which you should've gotten and linked when you installed SLisp.

As D only sees the function outline and not the implementation, to use the interpreter functions or any classes, you would need to link the SLisp file to your program, using the '-lsalmon' flag. You may also need to include the SLisp directory by using the -I flag as well.

4.2 A Very Basic Library

To create a library, you would first need to include the SLisp headers.

```
import sal_auxlib;
import sal_shared_api;
import salinterp; /* OPTIONAL: include Salmon interpreter */
```

Now before actually writing the functions, let me show you the function that SLisp looks for in an external library.

Review the following function, and after that I'll break it down for you.

```
extern (C):
/* function with the environment */
int sal_lib_init(SalmonEnvironment env)
{
    saL_register(env, "myfunction", &func);
    return 0;
}
```

Done? good!

Basically what the above function does is register a function called "func" which should be constructed as such:

```
int func(SalmonSub sub)
```

The 'SalmonSub' class, *previously named 'SalmonInfo'*, contains information for an external function to run. There's no convenient way to get all of the arguments from SLisp without this class.

In essence, it's a copy of the SLisp environment that the function runs in. When running the function it has the information for all of the arguments that the function was run with, and the value that the function should return, ran as such:

```
sub.returnValue(new SalmonValue());
```

(and yes, the SalmonValue by default returns 'nil')

4.3 Building This Library

To build the library, using the D compiler, **gdc**, you would run it like so:

```
$ gdc −I/path/to/slisp/headers/   \
        −lsalmon my−library.d   \
        −o my−library.so −shared −fPIC
```

Then you have a file called my-library.so in the current directory, if everything went well.

Then, from Salmon, just include the library and run the function, libraries are usually in two places, 'libs/', or the current directory.

```
(import "my−library")
(myfunction)
```

5 The Standard Library

Note: this is the rest of the book, you're free to finish reading, or continue to the manual below.

5.1 Base

5.1.1 print (&rest text)

Prints all of the arguments to stdout. If hello,world is passed, it will print hello and world on a new line.

5.1.2 println (&rest text)

Does the same as print.

5.1.3 null obj (boolean)

Returns 'true' if OBJ is **nil**.

5.1.4 type obj (str)

Returns the type of 'obj' as a string. This function is the concatenated version of the parameter's type enum.

```
(print (type "hello")) ; str
```

5.1.5 return obj (¿ any)

Returns the obj OBJ without any modifications. This is the default behavior with the last statement of functions.

```
(defun return_15 ()
        (return 15))
```

```
; OR

(defun return_15 ()
        15)
```

5.1.6 position {list} pos (¿ list@pos)

Returns list at the position [pos].

5.1.7 strcat {str1} {str2} (¿ str)

Concatenates 'str1' and 'str2' and returns it. (str1 + str2)

5.1.8 string-trim {string}

Trims both sides of {string} and returns it.
To trim a variable, use the following syntax:

```
(set variable (string-trim variable))
```

5.1.9 substr {string} {start} {end} (¿ str)

Returns (substr{start - end})

```
>(substr "hello" 0 3)
hel (str)
```

5.1.10 length {object [string — list]}

If OBJECT is string, return the length of the string, otherwise return the length of the list.

5.2 FileSystem

5.2.1 probe-file {name}

Returns 'true' if the file under {name} exists.

5.2.2 import {modname}

If the modname exists with a '.so' prefix, it will include the symbol which loads the low-level functions, otherwise, read and load the code from the source file.

5.2.3 require {modname}

Same as 'import', although this did come out first but is less powerful and extensive.

5.2.4 append {list} {obj}

Adds {obj} to {list} and returns the newly created variable.

5.3 Iteration

5.3.1 while {cond} {form*}

While {COND} is true, run {form*}.

5.3.2 until {cond} {form}

Until {cond} is true, run {form}

5.3.3 with {scope} {form}

While {scope} is not NIL, run {form}

5.4 Operators

5.4.1 + (&rest nums)

Adds all of the numbers [nums] and returns the result. If 1,6,8 is passed, it will return '15'

5.4.2 / &rest nums

Divides all of the numbers {NUMS} in order and returns them.

5.4.3 * &rest nums

Multiplies all of the numbers {NUMS} in order and returns them.

5.4.4 % {mod1} {mod2}

Modulus function.

5.5 Copy-and-write

CaW functions edit a list and return the newly created one.

5.5.1 truncate {list} {bound1} {bound2}

Returns list@[bound1 - bound2]

Note: if 'bound2' is '*', it will return the rest of the list from 'bound1'.

5.6 Math

5.6.1 intersection {list1} {list2}

Returns the intersection between 'list1' and 'list2'. As an example, if 1,4,6 and 2, 4,6 are passed into it, the function would return 4,6.

5.6.2 atan {&real x}

atan({x})

5.6.3 sin {&real x}

sin({x})

6 Math - Function Binds

6.0.1 cos {&real x}

cos({x})

6.0.2 tan {&real x}

tan({x})

6.0.3 asin {&real x}

asin({x})

6.0.4 acos {&real x}

acos({x})

6.0.5 asinh {&real x}

asinh({x})

6.1 Conditional Statements

6.1.1 if [stat] [expr]

Runs {expr} if [stat] is true.

6.1.2 case [stat] [true-expr] [false-expr]

If {stat} is true, run {true-expr}, otherwise, run {false-expr}.