TOP SECRET

DO NOT DISTRIBUTE

The Programming Language

LIGMAScript

DRAFT

(page intentionally left blank)

Contents

LIGMAScript

The Programming Language

This book introduces the LIGMAScript programming language.

Intended audience

Anyone with a basic understanding of modern digital computer programming.

1 Introduction

LIGMAScript is a high-level general-purpose programming language for digital computers.

The LIGMAScript contains only a handful of constructs, making the time spent learning of the language and writing an implementation extremely short.

The language is intended to be extremely portable and embeddable, it is without any built-in system calls or any similar contraptions.

In order to achieve these goals, the LIGMAScript language is intended to be compiled to byte-code and run in a virtual-machine.

There currently exists a single implementation of both a LIGMAScript byte-code compiler and a LIGMAScript byte-code virtual-machine. It is implemented in C++98.

2 Getting started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

```
"hello, world" print
```

The syntax of the language consists of words and brackets. That is all that is necessary. In this program *"hello, world"* is a word-literal and *print* is a word-function. When the program gets executed, each word in it also gets executed. As a word-literal, *"hello, world"* is executed by pushing its value unto the value stack. The *print* word, as it is a word-function, gets executed by executing its instructions, i.e. taking the value on the top of the stack and printing it to the console.

There are also word-values with names. These are, what are called, *variables* in other languages. Let's take the variable declaration from the C language:

```
x = 420 + 69;
```

The C language takes the value of the number *420* and the number *69*, computes their sum and them stores it in the variable *x*. In LIGMAScript the equivalent is this:

```
x 420 69 + set
```

The LIGMAScript pushes the word *x* unto the stack. It then pushes the word-literal *420* and the word-literal *69* unto the stack. After that it takes the top two words on the stack, that is, word-literals *420* and *69* and computes their sum and pushes the sum unto the stack. Finally the remaining two values on the stack get taken off of it and the value of the upper word gets copied into the lower value, that is, the integer *489* gets stored in the word *x*.

The LIGMAScript stack is a special memory location in the LIGMAScript virtual-machine, similar to the registers of a conventional processor. Unlike conventional processors, in the case of LIGMAScript, the stack stores references to words, not the words themselves. The contents of the stack can be printed at any time by using the *.s* word-function.

For example, to display the contents of the stack during the computation in the previous example you could use:

```
420 69 .s + .s
```

This would print the stack once after the integer *420* and *69* are pushed unto the stack, and again, after the computation of their sum:

```
[420 | INT] [69 | INT]
[489 | INT]
```

We can also store multiple number values in a single word by using vectors:

```
(vec 1 2 3)
```

All regular arithmetic operations can also be used on these values:

```
(vec 1 2 3) 420 * print
```

This will output the computed value to the console:

```
(420, 840, 1260)
```

It is only allowed to perform an arimthetic operation on a vector if the second value is either a single value or a vector of the same size, like this:

```
(vec 420 420 420) (vec 69 69 69) +
```

Which will compute to value *(489, 489, 489)*.

In case it is needed to store multiple values that are not numbers, or need to frequently change the size of, you can use lists:

```
(list 420 69 "very nice")
```

It is even possible to store lists inside of lists:

```
(list (list 420 69) "very nice")
```

Some word-functions, such as *print* are implemented in the LIGMAScript virtual machine as a bytecode instruction, but we can also create our own word-functions through LIGMAScript code:

```
(lambda "hello, world" print)
```

This statement will push the word-function to the stack. If we assign it to a word, we can execute it like any other word:

```
hello-world (lambda "hello, world" print) set
hello-world
```

This generates the same output as the first example in this book.

LIGMAScript does not support archaic concepts like loops, but is possible to divert the control flow of the program to the beginning of the function by using the ”repeat” key-word:

```
(lambda "hello, world" print repeat)
```

Do not run this function. It will cause your computer to catch on fire and explode. Instead, it is needed to add a stopping condition to the function:

```
hello-worlds (lambda
    (declare hello-loop)
    hello-loop (lambda
        dup 10 > if (               ; check for stopping
            return                  ; if is time to stop, stop
        ) else (                    ; if there is no time to stop
            1 +                     ; increase stopping counter
        )

        "hello, world" print cr     ; print some text
        repeat                      ; start again
    ) set

    0 hello-loop                    ; initialize stopping value and loop
) set
```

3 Continuing after starting

3.1 Words

As with most programming languages, there are restrictions on the names of words. LIGMAScript words cannot begin with the following symbols:

```
( ) [ ] * / + - 0 1 2 3 4 5 6 7 8 9 ‘ " . > < = !
```

Also these names cannot be the names of other words or built-in words.

3.1.1 Word types

The LIGMAScript language defines only these word types:

```
ATOM
INTEGER
FLOATING-POINT
STRING
FUNCTION
LIST-SEGMENT
```

There would be no need for more types, but since language must co-operate with

the C++98 language, there are more types:

```
INT8
INT16
INT32
INT64
FLOAT32
FLOAT64
CSTRING
CFUNCTION
```

These types can only be inserted into the LIGMAScript virtual machine from C++98 code, however their usage in the LIGMAScript code is the same as their respective LIGMAScript types. The vector types of numbers are simply the values being repeated, as in arrays of other languages.

There is no need to declare the global words, the LIGMAScript virtual-machine will declare them when the code is loaded for execution. However, in order to distinguish global words from function-local words, it is necessary to declare the local words like this:

```
(declare word1 word2 word3)
```

This declaration should occur only at the beginning of a function. It can be done anywhere else, but it will cause memory leaks, so it is not recommended to do so.

3.2 Arithmetic

There are built-in arithmetic operations in the LIGMAScript:

```
+ - / * floor
```

The floor operation converts a number to a number that is not a FLOATING-POINT. Other operations are defined in the LIGMAScript Standard Library.

3.3 Logic

There are built-in arithmetic comparisons:

```
> >= < <=
```

These can only be used on number types. Non-arithmetic comparisons:

```
== != is
```

These can also be used on atom, string and list types. The == and *!*= compare the values of the words, the *is* operator compares whether the words are the same word.

All of these word-functions take their arguments and produce an atom *true* or *false*. These atoms can then be further used with logical operators:

```
and or not
```

3.4 Control flow

LIGMAScript supports the *if/if-else* statements common to other languages:

```
if (do-something)
if (do-something) else (do-other-thing)
```

When the control reaches the *if* key-word, a word from the top of the stack will be removed. If this word is the atom *true*, then control flow will step into the subsequent block, otherwise it will continue after it, or if an *else* key-word is present, step into the block subsequent to it.

Also is the possibility to either step out of a function with *return* or go back to the beginning of a function with *repeat*:

```
(lambda do-something if (return) else (repeat))
```

3.5 Comments

In LIGMAScript the comments are preceded by the *;* character. Everything after the character until the next new line, will be ignored by the compiler. For an example:

```
420 69 + print    ; this prints the sum of 420 and 69
"very nice"       ; this prints very nice
```

when executed will output *489 very nice* to the console.

3.6 Functions

Functions in LIGMAScript are simply words that as their value either have a LIGMAScript or a C++98 function.

3.7 Lists

Lists in LIGMAScript are stored in the well-know linked-list manner. If we evaluate

```
(list 1 2 3)
```

we will get the first link of the list on our stack. We can retrieve its value with *data*:

```
(list 1 2 3) data
```

This evaluates to the number value 1. We can retrieve the next segment with *next*:

```
(list 1 2 3) next
```

This evaluates to the second segment of the list. We can then retrieve its value as well,

```
(list 1 2 3) next data
```

will evaluate to the number value 2. If we use *next* to reach the end of the list,

```
(list 1 2 3) next next next
```

we will get the atom *nil* on our stack. You can use

```
(list 1) next nil ==
```

to test for this value.

Here's a small function that will take a linked list on the stack and repeat until it prints all of its values out:

```
(lambda
    dup data print cr
    dup next nil ==
    if (
        drop return
    ) else (
```

```
        repeat
    )
)
```

4 Standard Library

// TODO

5 Modules

LIGMAScript modules are collections of additional *C++98* functions that can be added to the LIGMAScript virtual-machine.

5.1 Minimal runtime

The minimal runtime for implements a single function “exit” which can be used to terminate the runtime’s REPL loop. It takes no parameters and returns nothing.

// TODO

6 Runtime

LIGMAScript is designed to be an embeddable language, and as such is embeddable into any kind of runtime that has the ability to call *C++98* functions.

// TODO

7 Reference

7.1 Keywords

Keyword	Description
`+`	Arithmetic addition of two numbers.
`-`	Arithmetic subtraction of two numbers.
`*`	Arithmetic multiplication of two numbers.
`/`	Arithmetic division of two numbers.
`..`	Prints metadata of a word.
`==`	Value equality comparision of two values. Can be used on atoms, strings and numbers.
`!=`	Value inequality comparision of two values. Can be used on atoms, strings and numbers.
`>`	Arithmetic greater than comparision of two values.
`<`	Arithmetic lesser than comparision of two values.
`>=`	Arithmetic greater than or equal comparision of two values.
`<=`	Arithmetic lesser than of equal comparision of two values.
`and`	Logical conjunction.
`cr`	Prints a new line character.
`data`	Extracts the data reference from a list-link word.
`.s`	Stack print. Prints the stack. Can be used anywhere.
`declare`	Declaration block type declaration. Every symbol inside of this block will be considered by the compiler to be a word local to a function. Can only be used inside of functions.
`do`	Executes a function from the stack.
`dup`	Duplicates the word on the top of the stack.
`drop`	Discards the word on the top of the stack.
`else`	Defines an *else* block. Subsequent to the *if* block and preceding the *else* block in an *if/else* statement.
`explode`	Sets the machine on fire and explodes.

Keyword	Description
`fail`	Atom. Returned by functions that can fail and used to indicate failure of some kind.
`false`	Atom. Returned by logic functions and used to indicate a false logic value.
`floor`	Converts a floating-point word to an integer word, or leaves it as is if it's an integer.
`if`	Defines an *if* block. Preceding the *if* block in an *if/else* statement.
`is`	Compares two references, i.e. if they are a reference to the same word.
`lambda`	Function block type declaration. All code inside of this block will be considered by the compiler to be a part of a function.
`len`	Measures the length of vectors.
`list`	List block type declaration. Every word inside of this block will be considered by the compiler to be a word that is to be appended to a list.
`type`	Measures the type of a word.
`true`	Atom. Returned by logic functions and used to indicate a true logic value.
`maybe`	Atom. Returned by logic functions and used to indicate a third logic value.
`mod`	Arithmetic remainder of a division of two numbers.
`next`	Extracts the next reference from a list-link word.
`nil`	Atom. Used so signify a non-existing word.
`not`	Logical negation.
`or`	Logical disjunction.
`ok`	Atom. Returned by functions that can succeed and used to indicate success of some kind.
`over`	Duplicates the word second word from the top of the stack.
`print`	Prints a word's value.

Keyword	Description
`set`	Sets a word's value to a value from an other word.
`setnext`	Sets the the next reference from a list-link word.
`setdata`	Sets the the data reference from a list-link word.
`swap`	Swaps two words on the top of the stack.
`repeat`	Repeats a function. Can be used only in a function.
`return`	Returns from a function. Can be used only in a function.
`rot`	Rotates three words on the top of the stack, counter-clock-wise.
`vec`	Vector block type declaration. Every word inside of this block will be considered by the compiler to be a value in a vector.

7.2 Functions

// TODO

7.3 Bytecode

// TODO