LK Overview

# Introduction

The lk scripting language was developed by Aron Dobos at NREL. It is an open source product available at <lkscript.org>. For background reading on compiler development, the [tutorial](http://compilers.iecc.com/crenshaw/) by Jack Crenshaw is a lengthy introduction. The tutor program developed by Aron more succinctly expresses some of the basics of compiler development and provides a framework for how lk is constructed. The following are my own notes and interpretation of how lk is constructed and works.

# Software architecture

The language contains a lexicographical tokenizer to take a set of input instructions and break them into chunks recognizable to the compiler. The parser sets up a recursive descent tree, and then the code is either evaluated directly or assembly instructions are generated for the lk virtual machine. The available to lk is very similar to the [C-programming language](https://www.amazon.com/Programming-Language-Brian-W-Kernighan/dp/0131103628/ref=sr_1_1?ie=UTF8&qid=1489434583&sr=8-1&keywords=the+c+programming+language) (see library at end of this reference).

## Parser

The lk parser (parse) contains the functions used to take an lk script and parse it into a tree. The parser is broken up into layered components

* Script (scripts contain blocks of code)
  + Blocks (blocks of code contain statements)
    - Statements (basic statements that need to be evaluated
      * Tests
      * Enumerations
      * Loops
      * Etc.

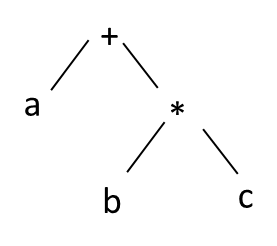
At a high level, the recursive algorithm takes the instruction set and evaluates it according to the recursive logic:

The parser takes the input set of instructions and constructs the tree by calling the lexicographical library (lex). The lexer extracts expressions to be evaluated by the parser. As the set of instructions is traversed, a tree of nodes is created.

For example, consider the statement: . Following the logic outlined above:

1. The character is found. This is not an expression, so nothing happens.
2. The next character is an operator . This is an expression, and so gets placed into a node. The character is placed into the left branch.
3. The right branch of the expression is , must be split further. Clearly is a term. Place that in the right branch of the first node. Then assign and to the left and right branches of that.

This results in the tree structure of:



## Evaluator

Once the instruction set has been generated it must be evaluated. The original lk functionality was to use the *eval* library; however, this used the system stack. Using the system stack creates problems with adding debugging capability, which is why the lk virtual machine (*vm*) was developed. The lk virtual machine generates the byte code from the parsed instruction set and runs the code. By having the code execute in a stack-based virtual environment, debugging hooks can be added to step through the instructions and verify they are behaving as expected.

To generate the assembly instructions, consider that the variables get loaded into memory (either the VM memory, or system memory). Then, the tree is ascended. Consider the example that

LOAD ‘a’ // 3

LOAD ‘b’ // 1

LOAD ‘c’ // 7

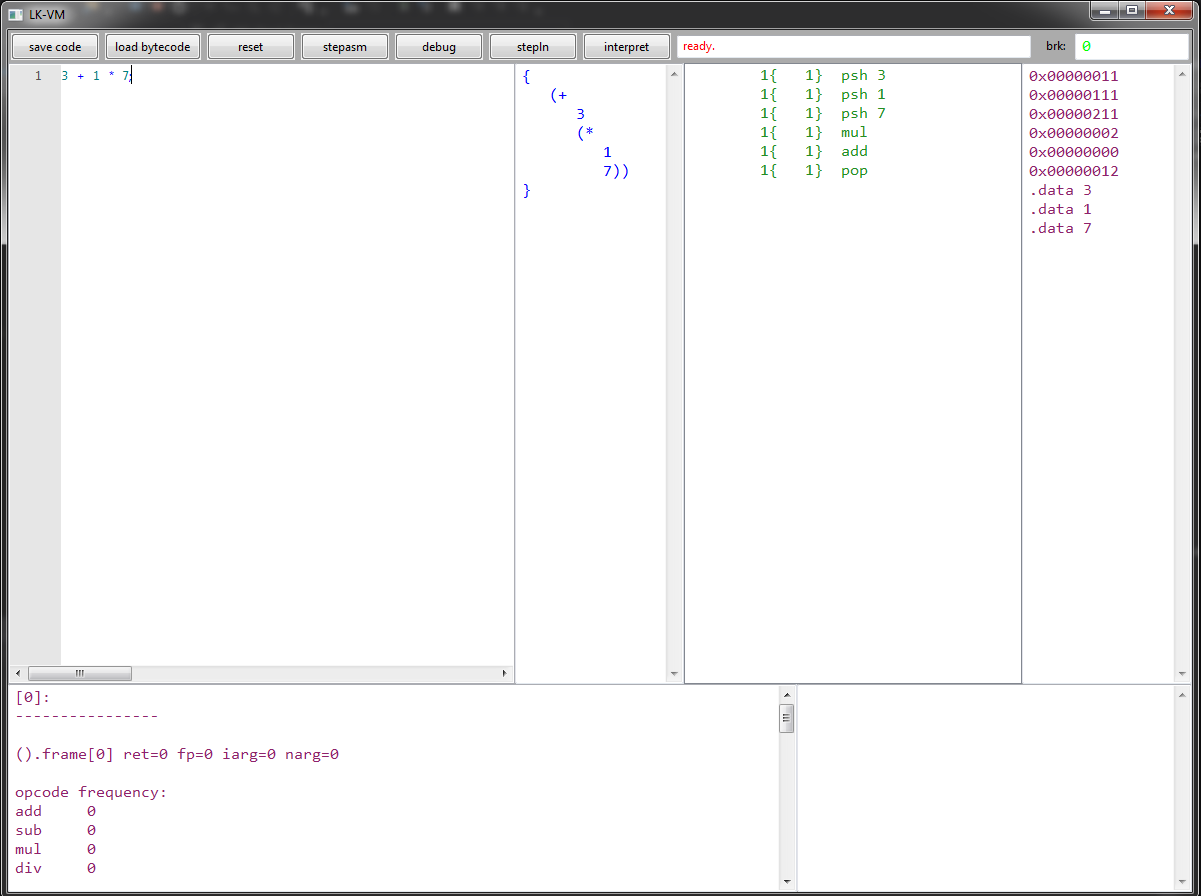
MUL ‘b’, ‘c’ // 7

ADD ‘a’, ‘b\*c’ // 10

As items are loaded, they are pushed onto the stack. As they are operated on, the size of stack changes. For instance, after the three loads, there are three items on the stack, with 3 on the bottom, 7 on top. The multiply results in 7 and 1 being popped off the stack, multiplied, and then 7 is pushed onto the stack. The add results in 7 and 3 being popped, and added, and 10 pushed onto the stack.

## LK debugger

To see the instructions being generated, one can run lkscript.cpp in wex and click “Shift + a”, or run the sandbox from the lk project. As valid lk is typed (left-most pane), the resulting parsed tree is generated in the next pane to the right. The subsequent assembly code generated in seen in the next pane, and finally, the generated hex byte code is shown.



## Putting together

There are of course more complexities (handling functions, more advanced operators), but at the end of the day, lk is a just a tool that parses a set of instructions into lexicographically recognized chunks which get parsed into a recursive descent tree and then evaluated (or generated and then executed in the VM). To see an example of a basic lk program, the example on lkscript.org is instructive.