Electrical and Computer Engineering Department
University of Puerto Rico at Mayagüez

ICOM4036 - Programming Languages
LCL(Logical Circuits Language)
Final Report

By

Eduardo O. Nieves Colon
Luis F. Ruelas Lisboa

For
Prof. Wilson Rivera
Electrical and Computer Engineering Department
University of Puerto Rico at Mayagüez
March 29, 2018

Introduction

The Logical Circuits Language, or LCL for short, is a simple programming language that allows a user to simulate a logical circuit with a few lines of code which are parsed into a Java intermediate code. After the successful compilation and execution of LCL intermediate code the user will be presented with their circuit in a GUI that allows them to see the result of their circuit simulation as well as edit the current circuit in real time to simulate other logical circuits by following the LCL code syntax.

Why use LCL?

Sometimes you want to simulate a simple logical circuit and you don't want to learn how to use software tools or simulators to accomplish a simple simulation of your logical circuit. You also don't want to have to learn to use programming languages like C, C++, C#, Python, etc., so that you can make your own simple simulation of a logical circuit. This is where LCL comes into play by writing simple grammatical expressions LCL provides a bridge between the simple and the complex. It provides a GUI, driven by Java, that is simple and allows you to edit your logical circuit in real-time by using its simple grammatical expressions.

Language Features:

- Translate LCL's simple grammar into complex Java code that can be compiled and run.
- Create a GUI that presents your logical circuits and simulates them and allows for real-time editing of those circuits.

Language Tutorial

The LCL code consists of the following parts:

1. Assigning the variable values
2. Writing the logical expressions

Assigning the variable values

In order to declare a new variable you need to open the lclcode.txt file. Then you can start assigning values to the variables you create following the LCL Syntaxis. An example input would look like this:

A=1

B=0

-Note that the variable names, need to be uppercase in order to work, and values must be either 1 or 0.

In the case of assigning new variables in the GUI, you need to press the edit button on the top left corner. This will open a new screen where you can edit or assign new variables(See figure 2 in the GUI Instructions section). Once you are done, press the run button and the simulation will update.

Writing the Logical Expressions

In order to declare a circuit in the simulation, the user must write the logical expression in the lclcode.txt file. Once open, the user can start declaring circuits following the LCL Syntaxis. An example circuit declaration would look like this:

E=CorD

M=DandNorG

F=(AxorE)nor(BandC)

-Note that variable names must be in uppercase letters and operations in lowercase, this will allow the LCL program to differentiate between variables and operations.

In the case of declaring circuits in the GUI, the user must press the edit button in the top left corner of the screen. This will prompt a new screen where the user can start declaring or editing circuits (See figure 2 in the GUI Instructions section). Once done, press the run button and the simulation will update.

Full Example LCL Code:

A = 1

B = 0

C = 1

D = AandBorC

E = CorD

F = (AxorE)nor(BxnorC)

LCL Compilation and Execution Instructions:

1)Download zip file of LCL repository.

2)Unzip LCL repository file in an adequate location.

3)Open Terminal and cd into the directory containing the LCL files.

4)To create your LCL code, open and edit the text file: 'lclCode.txt'

5)Save your changes to the 'lclCode.txt' file.

6)Now in the Terminal write: python LCLAPP.py and press 'Enter'

7)After receiving successful confirmation message the your Java intermidiate code is ready to be compiled and executed.

8)For Java compilation, in the Terminal write: 'javac -d runnable -sourcepath LCLCompilable LCLCompilable/app/LCLApplication.java' and press 'Enter'

9)To run your Java compiled code, in the Terminal write: 'java -classpath runnable app.LCLApplication' and press 'Enter'.

10) You will be presented with a GUI showcasing your logical circuit simulation.

GUI Instructions:

1)If you want to edit the inputs values or edit the logical circuit created, press the Edit Inputs button in right side of the navigation bar (See figure 1) .

2) You will be presented with a GUI showcasing your input values and the logical circuit code (See Figure 2).

3) Make the desired changes following the LCL code syntax.

4) Once finished press the Run button, a message will appear indicating either a successful Circuit Simulation or an error (See figure 3 & 4).

5) If there was no errors, you should see your new circuit shown in the GUI.

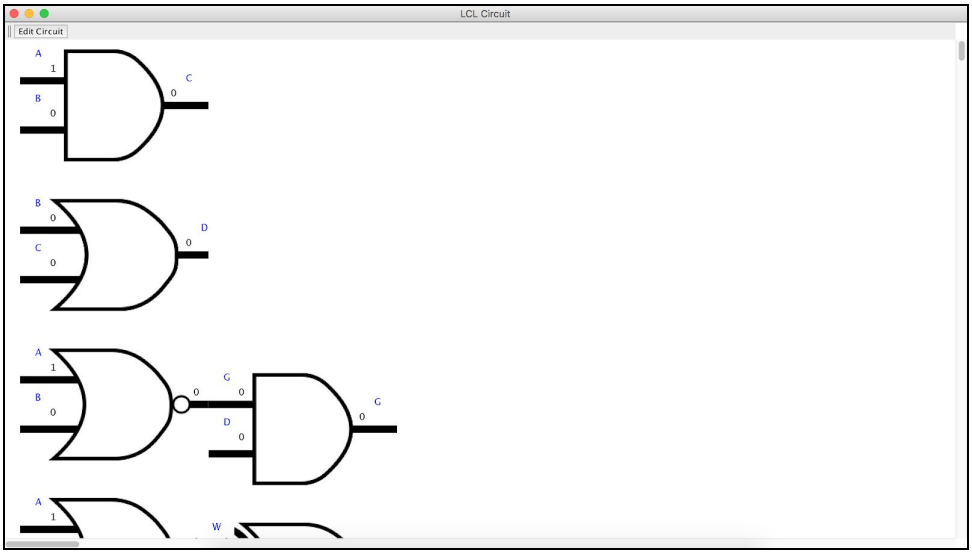


Figure 1: LCL Circuit GUI

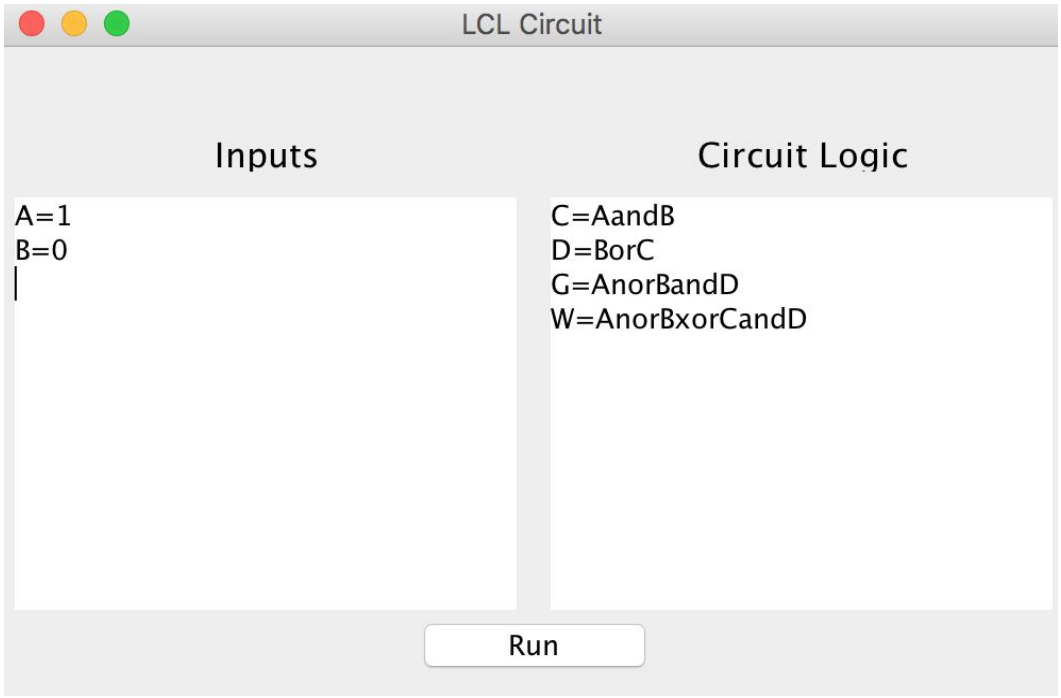


Figure 2: Edit Circuit GUI.

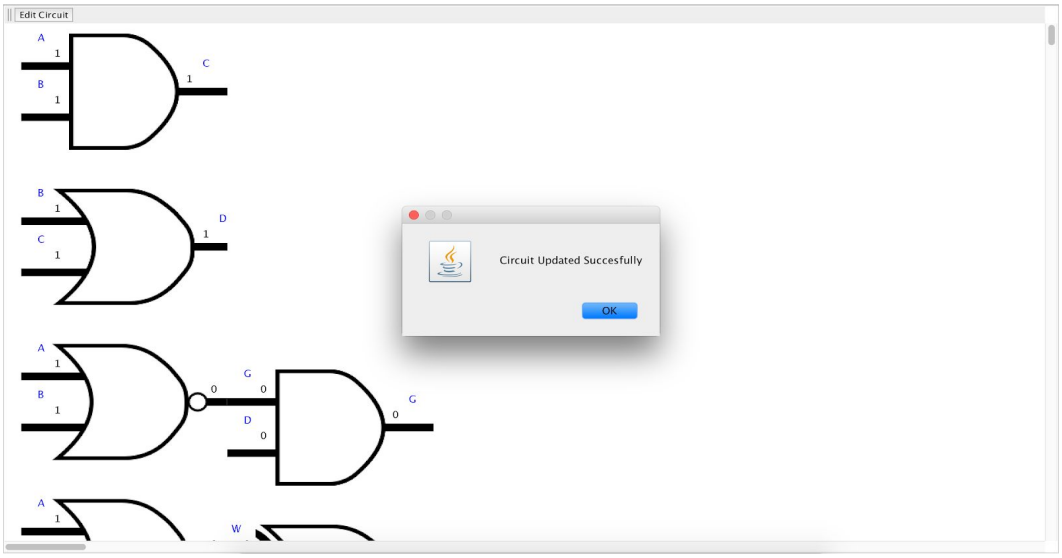


Figure 3: Success Message.

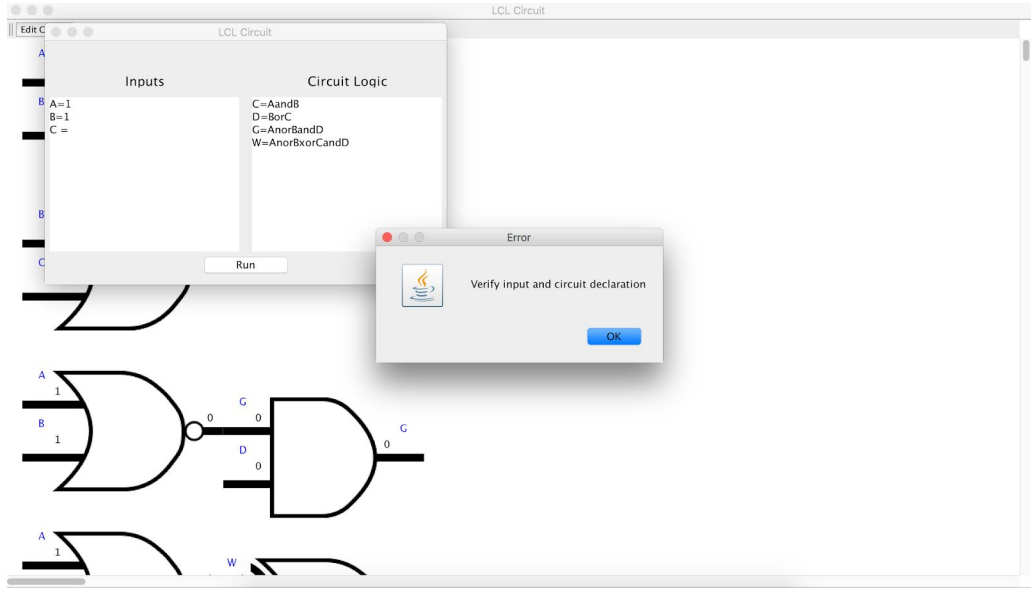
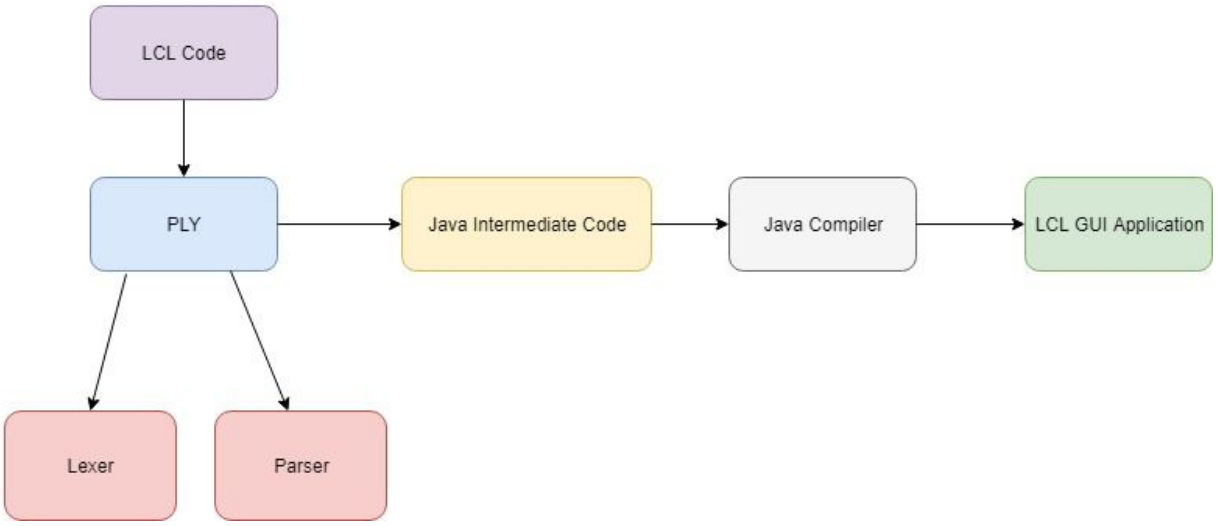


Figure 4: Error Message.

LCL Translator Architecture



LCL Code

The code written by the user. Here the user specifies the inputs and logical circuits it he wants to simulate.

PLY

This is an implementation of a lexer and parser tools written for Python.

Lexer

This tokenizes everything in the LCL Code and sends it to the parser.

Parser

The parser verifies the language grammar rules and uses a translator script to transform the LCL code into the Java code.

Java Intermediate Code

This holds the essential LCL application. Here the GUI is built along with all the logical circuits given by the parser.

Java Compiler and LCL GUI Application

Even though this isn't technically part of the translation, the Java compiler takes the java intermediate code and generates a java file that contains the LCL GUI Application which can be ran to simulate the logical circuits.

How the LCL Module Interfaces Work

The LCL Architecture works in the following way:

1) First the user writes code in the lclcode.txt file following the LCL grammar.
2) Then the user runs the LCL translator which reads the LCL code in the lclcode.txt file and is sent to the PLY Lexer which tokenizes the code and which is later sent to the PLY Parser which parses it.
3) When the PLY Parser finishes the Java Intermediate is created.
4) The Java Intermediate Code creates the GUI and logical circuits.
5) After the code is ready, the user can compile it using the Java Compiler.
6) The LCL GUI Application can be executed to simulate the logical circuits.

Language Reference Manual

Applications Included:

a)LCLAPP.py - This translates the LCL code found in the 'lclcode.txt' file and creates the required Java files that need to be compiled and ran using the Java Compiler.

Requirements:

a) LCL Code in the lclcode.txt file.

b) Latest Java JDK installed(Java 9 or higher).

c) Python 2.7 installed.

d) Files found in LCL repository.

Application Execution:

When the above requirements are met run LCLApp.py and then run the following Java terminal commands from within the LCL directory:

a) javac -d runnable -sourcepath LCLCompilable LCLCompilable/app/LCLApplication.java

b) java -classpath runnable app.LCLApplication

Code Rules:

I. Input and Circuit Declaration Name

-Both input and circuit declarations names must be a capital letter from A-Z.

II. Input Declaration Values:

-Input declaration follow the following scheme:

InputName = ‘1’ or ‘0’

III. Circuit Declaration:

-Circuit Declarations follow the following scheme:

CircuitName = ‘InputName or PreviousCircuitName’ ‘operation’ ‘InputName or PreviousCircuitName’

-Note: Circuit Declarations can have multiple operations chained together and the use of parentheses is allowed for clearer reading of the code.

IV. Operations:

-These are the valid operations which must be written in lowercase:

1) and - for AND operation
2) or - for OR operation
3) xor - for XOR operation
4) nand - for NAND operation
5) nor - for NOR operation
6) xnor - for XNOR operation

Software Development Environment

Since the LCL uses two different programming languages for its implementation: Python and Java, two IDEs where used.

Eclipse JEE (Oxygen Package)

This popular IDE was used for the development of the intermediate code in Java.

Visual Studio Code

This lightweight IDE was used for the development of the LCL translation(including its lexer and parser) in Python.

Test Methodology

Lexer

For this part the testing was done by giving the lexer a text file to analyze and tokenize the input into its respective categories and printing out the result, which we would later verify.

Parser

After we defined our grammar rules, we developed the parser for LCL. We tested the parser by sending our LCL code and verifying that it recognized the correct syntax and followed our grammar rules while also recognizing and notifying errors in code.

Python to Java

This part was simple as this was just writing strings, obtained from the parser, into a Java text file, if any errors were made the Java compiler or Eclipse would let us know.

Java Intermediate Code

This code was tested by compiling and running the Java code both in Eclipse and also in the Terminal using Java commands.

Programs Used for Translator Testing

We tested everything as described in the section above, however we did make LCLApplication.py Python script that allowed us to automate the process of translating our LCL code (found in lclcode.txt), which would later be a script that the user would use to test their own code. After the script was finished we would compile and run the Java intermediate code to verify it completed its translation correctly.

Conclusion

A great way to understand how a programming language works is to develop your own programming language. Sometimes you would like for an easier way to do something, but simply there isn't one, so you have to create it yourself, which isn't something easy either. This was one of the reasons for developing LCL, we know for experience that dealing with logical circuits can be a bit confusing and intimidating, and online simulators can be more complex than what we want to do. We wanted to give the users a way to see how their circuit would look by simply writing the logical expression and its values. By writing this language combining Java and Python not only we made a useful tool for logical circuits, but it allowed us to grow as developers by learning new skills and refining those that we already knew. I'd like to say that when you want an easier way for something, but there isn't one, create it yourself, not only you will get better at it, but you will feel an immense satisfaction once you are done.