

# DPL Music Compile Magic

Dimitri Harding  
School of Computing and Information  
Technology  
University of Technology, Jamaica  
Jamaica

Sheyinka Harry  
School of Computing and Information  
Technology  
University of Technology, Jamaica  
Jamaica

Andre Hutchinson  
School of Computing and Information  
Technology

University of Technology, Jamaica  
Jamaica

Megan Hutchinson  
School of Computing and Information  
Technology  
University of Technology, Jamaica  
Jamaica

**Abstract—** The goal of our Music Compiler Magic project is to perform lyric compilation for a given song, and then matching these lyrics to synthesized beats. It processes the lyrics using the different compiler phases and the uses a text to speech API and some predefined instrumentals to produce music. The project can be ran from the terminal or used as a client-server.

## I. INTRODUCTION

The musical compiler was coded using the Node.js environment which allowed for the use of Javascript both on the client and server side. This platform was chosen as it presented the team with a “different” way of coding as well as it allowed the team to look at easier ways to implement the different phases of the compiler. Node.js is also platform independent which meets the requirement for cross-platform compilation of the DPLMusic Compiler.

## II. PREPROCESSING

In a compiler the first phase is the preprocessing which normally would expand macros or include some libraries. We decided to mimic this by expanding lines in the lyrics that has a multiplier

delimiter - [x4]. The replication expression must be written with the ‘x’ before stating the number of times the lines are to be repeated. If this phase realizes this pattern ([3x]) it will show an error stating that this is the incorrect format. The compiler will not move to the next state phase this is corrected.

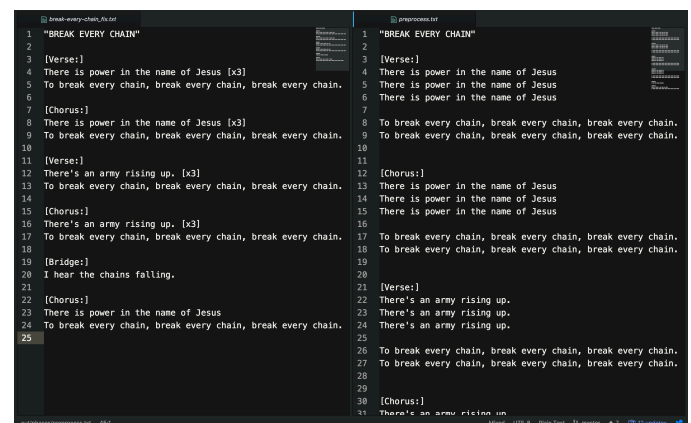


Figure 1 A side-by-side view of the original text file and the output after preprocessing

## III. LEXICAL ANALYSIS

The lexical analysis phase is used to convert the sequence of characters from the preprocess phases into a sequence of tokens: strings with an identified

meaning. It also scans and add useful information about the position of a word lexeme. It accounts for punctuations, words, sentences, categories, and a title which are all verified by a regular expression before adding the token to the token stream. Anything else is define as an unknown token and a lex error is thrown and the compiler stops at this phase until this is corrected.

Below is apart of a lyrics file:

"BREAK EVERY CHAIN"

[Verse:]  
There

Tokenized and represented by the following table:

Lexeme	Token Category
"BREAK EVERY CHAIN"	"Title"
[Verse:]	"Category"
There	"Sentence"
There	"Word"

Some of the regular expressions that were used in this phase:

- /(.+)/gmi - matches a line except whitespaces
- /^([\.\+\])/mi matches a category
- /^(").+\$/mi - matches title
- /s/ - matches whitespaces
- /w+/i - matches any word
- /w+\w+/i - matches any word with apostrophe
- /([a-z ].+)/i - matches any sentence

#### IV. SYNAX ANALYSIS

Also known as the parsing phase where the tokens are analyzed to see if they conform to the rules that we created for accepting a lyrics file.

Rules:

- Categories can only be *Verse*, *Chorus* or *Bridge*
- Title should be enclosed in quotes and be all caps
- Any punctuation token should be any of the punctuation that we are looking for (`(\.\!|\'|\?|\:|\,|\/)`)
- Sentences should begin with capital letters

As each token is verified that they match a particular rule, we generate an Abstract Syntax Tree (AST) on the fly as the tokens are traversed. The AST is represented as a JSON object with different nodes and children of those nodes. In the AST the first node is the LyricsNode and the children of that are the TitleNode and CategoryNode. The TitleNode is the source of the TextNode which contains the value of the title and the subsequent WordNodes which are its' children. On the other hand the CategoryNode has the different subcategory nodes (VerusNode, ChorusNode, and BridgeNode) as its' children. Each one of these children then have subsequent SentenceNodes, which are the parent of following WordNodes and PuncNode. If there is an error, it throws it else moves on to calling the semantic analyzer.

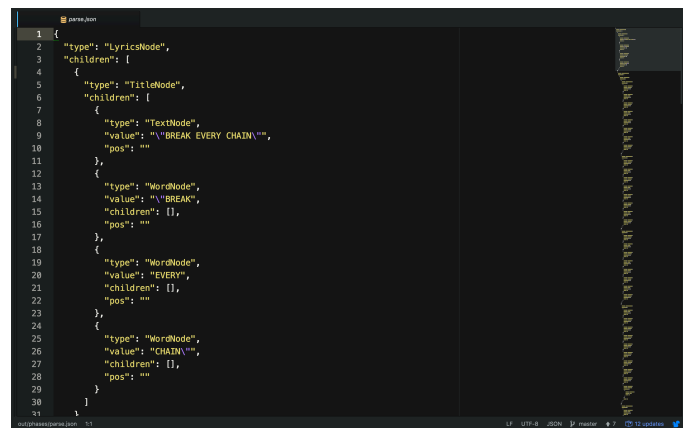


Figure 2 Abstract Syntax Tree

#### V. SEMANTIC ANALYSIS

This phase is uses functions that return a Boolean to determine if the lyrics structure provided by the AST is correct, also it searches for certain words that should be properly capitalized e.g Gospel:-

Lord, God, Jesus etc, and words that shouldn't exist in lyrics like curse words. It utilizes a dictionary that provide they information to aid with the checking. It also checks the spelling of all words in the AST to see if they were spelt correct. The dictionary used for spell checks are native binding for dictionary that exist within the environment that the compiler is running on. Once everything returns true then the compiler moves on to Intermediate Code Generation.

## VI. INTERMEDIATE CODE GENERATION

The Intermediate Code Generation is responsible for traversing the AST and retrieve all the leaf nodes containing a word. It checks to see if a sentence ends with a punctuation, and if not it inserts a "comma" at the end of the sentence. Once the lyrics is represented in this form, this phase then converts it to ASCII code, Allowing for the IR to be transferable across different platforms.

## VII. OPTIMIZATION

In this project we introduced two level of optimization. The first being the removal of square brackets from the IR – ASCII representation. While the second being the compression of the IR- ASCII representation. It uses the Huffman compression algorithm to achieve this.

## VIII. CODE GENERATION

And finally this phase takes the optimized IR and then convert it to binary.

## IX. SYMBOL TABLE

Thorough the compilation phases a symbol table was use to store some relevant details about the lyrics. Those include: Genre, Title, Song Structure, Lines, and Word Count.

## X. TRANSLATION

The translation option that is used in the UI triggers whether or not the IR code is going to be converted to Spanish after the front-end phases were completed. A call was made to the Microsoft Translation API that accepted the data to be

translation, the source language and the expected translation language and then returns the translation data.

## XI. TEXT TO SPEECH

The utilization of the speech synthesis in the Web Speech API which allowed us to access speech synthesis within a browser once it is supported. This was used to produce the singing of the song that was compiled from the lyrics.

## XII. INTEROPERABILITY

In our project we allowed the sharing of the IR code with all connected clients. A client is given the option to share with other clients the song that they have compiled. Once all of the compilation phases ran with no errors and the share with other clients is selected, a message is sent to all clients except for the one who created the event. This message asks other clients if they want to hear the song that was shared with them. If this message was accepted, then the server sends the client the IR which is then used to create the song on the accepting client.

## XIII. PROJECT STRUTURE

This section shows the overall structure of the project and where each relevant files or data can be accepted, and also what the project was built on. From here on "**root**" will represent the root directory of the project.

- The Node module Express was used to create the Client – Server of the compiler.
- Request is made to the server using custom API and also Sockets which is used to trigger specific events
- All of the compilation phases code can be found in the **root/lib** folder.
- We created some helpers (ast-builder.js, linked-list.js etc. ) to use through our project and they can be found in: **root/util**
- Request to the server were handled by the index.js in: **root/routes**
- The client page is located in: **root/views**
- Lyrics that were used for this project are in the **root/lyrics** folder

- Dictionary resources:
- **root/resources/dictionary**
- Output from all the phases can be used in the UI or in the **root/out** folder
- Documentation: **root/Documentation**

#### XIV.PROJECT USAGE

This assumes that node is already installed

Localhost

- \$ npm install nodemon -g
- Clone project from repository
- cd into project folder
- \$ npm install
- \$ nodemon ./bin/www

- Project should now be running on localhost:3000

CLI

- Clone project from repository
- cd into project folder
- \$ npm install
- node ./index.js [path to lyrics file]

Web App

There is also the option of using the app online:

- <https://music-compiler-magic.herokuapp.com/>

where the option is given to paste in the lyrics or upload a file.