Codable Media Mashup

A language. A command line tool. A quick and dirty video editor for the power user.

Nathan Karasch
Gregory Steenhagen

11 December 2016

Table of Contents

Overview 3

New and Complex Material 5

Working at Higher Levels of Bloom's Taxonomy 7

- Creation 7
- Analysis 10
- Evaluation 12

Example CoMM File 13

Future Plans 14

Resources 16

Screenshots 17

Overview

Codable Media Mashup (CoMM) is both a language and a command line tool used for slicing and splicing online videos together. The purpose is to allow people to quickly and efficiently make "highlights reel" or “compilation” types of movies using a collection of online videos. The *traditional* video editing process might look something like this:

1. Individually download the videos, and ***wait*** for them to complete.
2. Load them into a bulky video editing application, and ***wait*** for it to process each video for thumbnails.
3. Select clips from the various source videos, adjust the time sliders, ***wait*** for the CPU to catch up as the graphics card gets bogged down.
4. ***Waiting***, ***waiting***, ***waiting***… you get the point.

By contrast, with a “codable” video editing experience, users can avoid the wait! Or at least most of it. For many people, video editing shouldn’t need a feature-heavy application. It just needs to put pieces of videos together. The less time spent doing it, the better! In addition to it being faster, all the downloading and processing is done in the same contiguous block of time. This allows the user to “set it and forget it” so they can work on something else until their video is complete.

Potential use cases:

- Game footage highlights from the season
- Top ten funniest Seinfeld moments
- Cat fail compilations
- Caruso one-liners montage
- Compilation of every time Neo says "No", "Why", or "I don't understand". Wake up Neo…

Project Complexity

This project involved creating a fully functioning interpreter that performs lexical analysis, grammar parsing, translation, and execution. Up until this point, we have only created lexers and parsers with ANTLR. The interpreter had to be able to provide helpful error messages and output useful log files as well, which involved understanding StandardError/StandardOutput redirection in a Java program running a bash script.

Creation, Analysis, and Evaluation

Creation began by defining what we wanted the language to do and writing the grammar from that definition. We focused on getting end-to-end integration and a working prototype

before drilling down on the finer details of the program. Translation and execution were implemented by using a bash script as an intermediate language, and then running that script from within the Java application. The basic features of the program allow the user to define the output video file name and cache, download numerous videos, slice them into shorter clips, and then join those clips into a single video.

Once the initial prototype was functional, it was analyzed and improved upon in additional iterations. The repository was completely restructured and refactored to comply to better coding standards and to promote readability and maintainability. The process involved decoupling distinct modules, restructuring the file directory, updating build scripts, removing cruft, refactoring variables and functions, and adding comments and documentation.

The application and project goals were evaluated intermittently during development, and plans had to be adjusted to arrive at a satisfactory end result within the allotted time. We determined that it was more important to produce a stable application with a few core features and robust error handling than it was to make a buggy, unstable application with lots of features. Original plans for a client-server web application were shifted to target a command line application, and extra features were tabled to focus on the more essential pieces of functionality.

The result, Codable Media Mashup, is a fun new way to edit videos with a few easy commands!

New and Complex Material

During ComS 319, we covered the basics of making ANTLR lexers and parsers to handle different grammars. This project extended upon that knowledge by fully implementing a language interpreter. Not only does CoMM tokenize and parse the grammar, but it also executes the code. Wikipedia defines several program execution strategies for interpreters, the second of which is "[to translate] source code into some efficient intermediate representation and immediately execute this." [1] This is what CoMM does. The intermediate representation is a bash script that employs youtube-dl and ffmpeg to do the heavy lifting of downloading, slicing, and splicing the videos.

The entire process employed by the interpreter can be outlined as follows:

1. **Lexical Analysis** - "Tokenizes" the input
2. **Parsing** - Creates a parse tree from the token stream
3. **Translation** - Creates a bash script as the parse tree is walked. The script takes the following form:
    - CoMM definition 1
        - File Management - Readies the cache directory for CoMM 1
        - Video Downloads - Uses youtube-dl to download any source videos to the cache for CoMM 1
        - Video Slicing - Uses ffmpeg to slice sections out of the source videos
        - Video Joining - Uses ffmpeg to join the various slices into a single video
    - CoMM definition 2
        - …
    - …
4. **Execution** - Executes the bash script, logging all output to a file for ease of debugging.

If any errors occur during steps 1 through 3, the errors are displayed to the user and the Execution step does not occur.

The error handling during translation constituted another new and complex element of the project. ANTLR provides some built-in error-handling during the lexing and parsing steps, but we wanted to display additional helpful errors to the user to help troubleshoot problems with a CoMM. Some of the different errors that it looks for includes:

- Duplicate output filenames in the same CoMM file.
- Using a variable that hasn't been defined.
- Empty CoMM definitions

[1] https://en.wikipedia.org/wiki/Interpreter_(computing)

- Stop time occurring before start time in the **add()** function
- String literal to integer parsing errors

We didn't want the bash script to run if there were errors, but we also wanted to look at the entire file for errors. As the program walks the parse tree, errors that we find are written to a buffer and the **errorStatus** is set to "true". Once the entire parse tree is walked, if the **errorStatus** is "true," it will print out the error buffer and exit the program.

Another troubleshooting / error handling method we used was the creation of log files. Since the bash script itself can produce runtime errors, we decided it would be best to create log files for the bash script execution rather than dump it to the terminal for several reasons:

1. It's a cleaner user experience;
2. It's faster, since printing thousands of lines to the terminal has a time cost; and
3. It allows for easier debugging, since the bash commands can be run verbosely and the resulting log files can be opened and reviewed in a separate text editor if necessary.

This would have been really easy to do with basic output redirection if the bash script was simply run from a bash shell. However, since the script was being run from within a Java program, there was a significant amount of work involved in figuring out the redirection piece. Numerous rabbit trails were followed, but eventually a solution was found that worked!

Working at Higher Levels of Bloom's Taxonomy

Creation

The creation of Codable Media Mashup began with the design of the language syntax. We started by writing the README file as the documentation for the language, and then turned that documentation into an ANTLR grammar file. The process was pretty straightforward, with only a few minor changes occurring as we began implementing the grammar. This represented a sort of "top down" approach to the design process.

The grammar took very little time to write, and we were able to quickly move on to the "translation and execution" part of the interpreter. As mentioned previously, this was new ground for us. It took a bit of digging, but the most helpful resource on this topic turned out to be a book called The Definitive ANTLR 4 Reference by Terence Parr, the creator of ANTLR. Thankfully, it was available in "full text" through ISU's online library system.

The "translation and execution" code underwent several iterations as our understanding of the ANTLR API increased. During the first iteration, the major drive was to reach a state of end-to-end integration. The saying "integrate early and often," (as well as the ghosts of projects past) gave us the following goal: Get a working prototype that translates a CoMM file to a bash script. Once this goal was reached, the biggest hurdle was behind us. Then we just had to clean things up, improve error handling, find a way to execute the bash script, and log the script output.

The following is a list of all the CoMM grammar that ended up being implemented, as well as the description of how each is used and a short example. It serves as the current documentation for the language.

```
CoMM <videoName> [cache(uniqueCacheName)] ;
```

Sets the name of the output video and defines which cache namespace the program will use when re-using video URLs or variable names. If the **cache()** option is used, then the **uniqueCacheName** provided will define the cache. If the **cache()** option is left out, the video name will define the cache. The cache option is useful if you want to define multiple CoMM's in one file using a shared set of videos. By using the same cache, shared videos won't be re-downloaded. Acceptable name and cache namespace characters are **[a-zA-Z0-9_]**.

Example:

```
// The first three videos use the same cache, so shared
// videos between them will only download once.

CoMM Highlights_Reel_1 cache(monday_night_game_footage);
...

CoMM Highlights_Reel_2 cache(monday_night_game_footage);
...

CoMM Highlights_Reel_3 cache(monday_night_game_footage);
...

CoMM And_Now_For_Something_Completely_Different;
// Doesn't use the same cache as the other three videos
```

// Comments

Everything after a double forward slash until the end of the line is a comment and will be disregarded at runtime.

Example:

```
// This line is a comment.
CoMM MyCoolVideo;  // This is also a comment to the line's end.
```

var <varName> = <varName | string | int | bool ...>;

Defines a variable to be used elsewhere in the program. String literals require enclosing double quotation marks, but the rest don't. Acceptable variable name characters are **[a-zA-Z0-9_]**.

Example:

```
var bananaPhone = "https://youtu.be/j5C6X9vOEkU";
var startTime   = "1:05";
var duration    = 12;
var upscale     = false;
```

add(string url, [string startTime, string stopTime]);

Adds a video or video clip to the stream. If no start/stop times are given, the entire video will be added. Start and stop times can be provided to use only a certain clip from the video. The start and stop times must be either both present or both absent. The times are given in "minute:second" format.

Example:

```
// Add the whole video?
add(graduationVid);

// Or maybe we'll just add my glorious moments on stage...
add(graduationVid, "59:12", "63:07");
```

config.<option>;

Sets a configuration option for the CoMM video as a whole. There were originally more options (See **Future Plans** below), but most of them were dropped.

config.noCache() - Tells the program NOT to use cached video from previous runs. It will slow down re-runs, but you're guaranteed to get the most recent versions of the source videos. *NOTE: This will disregard any **cache()** option given in the **CoMM <videoName>** definition.*

Analysis

The first iteration of the interpreter may have been to get a working prototype as quickly as possible, but the following iterations definitely took a more analytical approach. We looked at the code with the following questions in mind:

Can the repository be better structured?

Initially all the main development files were kept in a single folder, and no package structure was used. This worked for rapid development, but a long-term, maintainable code repository needs more structure. By adding a `**@header**` definition to the g4 file, we were able to give all the generated files a package name, and by updating the build script we could organize the files much better. (It was especially nice having all the class files in a separate location from the source files!) Researching and figuring out the necessary changes to the build script took a couple hours, and the resulting commands are a little more complex than the originals:

```
java -Xmx500M -cp “src/main/resources/antlr-4.5.3-complete.jar”  \
      org.antlr.v4.Tool src/main/java/comm_grammar/comm_grammar.g4
javac -cp “src/main/resources/antlr-4.5.3-complete.jar” -d class \
      src/main/java/interpreter/*.java                           \
      src/main/java/comm_grammar/*.java                          \
      src/main/java/utils/*.java
jar cmf0 src/main/resources/META_IF/MANIFEST.MF Comm.jar -C class .
```

What can be done to make the code more readable and maintainable?

This was the boring, tedious work of documentation and cleanup. “Cruft” was removed from early development. Public API were documented. Inline comments were added for readability. Variable names were refactored to make more sense, and functions were created to promote code reusability.

What are the distinct modules, and how can we decouple them?

As previously mentioned, all the files were originally in one folder with no package structure. After restructuring the repository, it was time to restructure some of the code to distinguish separate elements. Originally all the interpreter java was in one giant “God class.” This was split into several separate classes, each with its own file. With the addition of a sound utility, the resulting structure looks like so:

- src/main/java
  - comm_grammar
    - comm_grammar.g4
    - (java files generated by ANTLR)
  - interpreter
    - CodeGenerator.java
    - Comm.java
    - CommLocation.java
  - utils
    - SoundUtils.java

Additionally, it was important to remove all the java code from the grammar file, because ideally you want your ANTLR grammar to work independent of a certain application-specific language. The only java code left in the file was the header containing the **package** statement that goes at the top of all the generated java files.

Evaluation

The original plan of this project was to host the command line tool on a web server that gets accessed through the browser. The experience would look like so:

1. The user uploads a CoMM file or enters a CoMM definition in a textarea.
2. The file is sent to the server, which runs the command line application.
3. Once complete, the video file is sent back to the user for download.

The cached files would only stick around for 24 hours before being auto-magically cleaned up by the server. Restrictions would need to be in place to prevent abuse, but overall we think it would be pretty doable. We liked the idea of letting a really fast server do the downloading/processing instead of using the user's own computer, thus freeing up the user's memory and bandwidth for more important things!

This plan, however, did not come to full fruition. As the project deadline approached, we made the decision that it would be better to shoot for a stable application with core features and robust error handling rather than an unstable, buggy application with lots of features. We separated the *needs* from the *wants* and made the following decisions:

- The application *needed* to parse and execute the CoMM grammar, creating a movie from slices of online videos.
- We *wanted* it to have a browser GUI, but it wasn't necessary. The client-server integration, cache cleanup measures, and security precautions would have added a significant amount of work, so time restraints forced us to cut that feature.
- We *wanted* it to have several configuration options to give the grammar more substance, but there wasn't time to implement all the features. These could easily be added towards the end if we had more time, so we decided to table these to focus on more essential functionality.
- It *needed* to have features to help the user troubleshoot, such as helpful error messages and execution logs.

The result of these decisions was a command line application with the core functionality we were looking for. Cache cleanup can be done by the users at their discretion, and since it all runs on the user's own machine, there isn't really any major security hazards. If someday we implement the **requestVideoCredentials()** command (see **Future Plans** section below), this will be much easier to do in a command line application than it would be as a web application.

Example CoMM File

Here is a basic example of a CoMM file with several video definitions:

```
CoMM banana_phone_clips cache(BananaPhone);

// This part is just showing the different types of variable assignments
var bananaPhone = "https://youtu.be/j5C6X9vOEkU";
var startTime1 = "1:20";
var stopTime1  = "1:30";
var someInt    = 20;
var someBool   = true;
var sameUrl    = bananaPhone;

// We can add the video with variables or literals
add("https://youtu.be/j5C6X9vOEkU", startTime1, stopTime1);
add(bananaPhone, "2:00", "2:05");


// This will be a second, separate video with a different cache
CoMM duckSong;

// Adding the entire "Duck Song"
add("https://www.youtube.com/watch?v=MtN1YnoL46Q");

// We can still use the variables from earlier definitions, but since
// this video is in a different cache, it will re-download the banana
// phone video.
add(sameUrl, "0:00", "0:05");


// This third video will use the same cache as the first video
CoMM dance_moves cache(BananaPhone);

// King Louie's moves
add("https://www.youtube.com/watch?v=9JDzlhW3XTM", "0:40", "1:28");

// Even though the variables `startTime` and `stopTime` were defined
// earlier in the file, we still have to use the `var` keyword when
// we assign them again.
var startTime = "1:35"; var stopTime = "2:17";
add(bananaPhone, startTime, stopTime);
```

Future Plans

As mentioned above, we made the decision that it would be better to shoot for a stable application with core features and robust error handling rather than an unstable, buggy application with lots of features. The following features were originally part of the plan and have been built into the grammar, but they have not yet been implemented in the translator.

`config.scaleByWidth(int width)`

Scales the video to the desired width, keeping the height proportionate to the tallest video added. For example, if we have a 320x240 video and a 500x500 video and we call **config.scaleByWidth(448);**, the CoMM's scale will be set to 448x448. The smaller video will have black bars added to the top and bottom, and the larger video will be scaled to 448x448.

`config.scaleByHeight(int height)`

Scales the video to the desired height, keeping the width proportionate to the widest video added.

`config.scale(int width, int height, bool keepProportions)`

Sets the size of the video, with the option to keep or ignore the original proportions.

`config.preventUpscaling(bool constrainScale)`

Prevents videos from being scaled larger than their original size. If **constrainScale** is set to **true**, the entire CoMM's scale dimensions will be reduced to the largest permissible video scale. Otherwise, it will add a black border around videos that cannot be upscaled.

Example:

```
// In each of these examples, there are two videos: One is 640x480 and
// the other is 320x240

CoMM video1;
config.scale(448, 336, true);   // Sets the scale of the CoMM to 448x336.
config.preventUpscaling(true);  // Since the smaller of the two videos is
320x240,
                                // the CoMM scale is reduced to 320x240 because
                                // we have set `constrainScale` to true.
...

CoMM video2;
config.scale(448, 336, true);   // Sets the scale of the CoMM to 448x336.
```

```
config.preventUpscaling(false); // Since we're not constraining the scale here,
                                // the CoMM scale remains at 448x336 and black
                                // bars are added around the smaller video
                                // instead of upscaling it.
...
```

`requestVideoCredentials(string url1, string url2, ...);`

This allows the program to download and manipulate private videos that require credentials. The username and password will be requested at runtime so nothing gets written in plain text in the code. They are not cached or stored on the server, so you will have to input a username and password every time the program is run, once for each **requestVideoCredentials()** function. All the videos listed in a single function will receive the same username and password.

Example:

You have three videos, with variable names **funny1**, **funny2**, and **funny3**, on your YouTube account that you want to use, but they're private. You have another private video from a different website assigned to the variable name **yoloFail**. You should make 2 separate calls to this method, like so:

```
requestVideoCredentials(funny1, funny2, funny3);
requestVideoCredentials(yoloFail);
```

At runtime, the program will ask for a username and password for the first video set, and then it will ask for a username and password for the second video set.

Other Potential Features

- Audio integration (e.g. adding background music to a video compilation)
- Overlaid text (a quick video meme machine!)
- Using local video files
- Finer precision on time start/stop (nanoseconds?)
- More output format options
- Output to audio (mp3)

Resources

GitHub Repository:

https://github.com/KrashLeviathan/codable_media_mashup

ANTLR:

http://www.antlr.org/

The Definitive ANTLR 4 Reference

Parr, Terence. *The Definitive ANTLR 4 Reference*. Dallas, TX: Pragmatic Bookshelf, 2013. *ProQuest / Safari Books Online*. Web. 11 Dec. 2016.

Screenshots

```
nkarasch@nkarasch-HP-EliteBook-8570w:~/GitStuff/codable_media_mashup$ tools/build.sh
antlr4 src/main/java/comm_grammar/comm_grammar.g4     (abbreviated)
javac src/main/java/*.java                            (abbreviated)
jar cmf0 src/main/resources/META-IF/MANIFEST.MF Comm.jar -C class .

To run, use the following command:

    java -jar Comm.jar input_file.comm

nkarasch@nkarasch-HP-EliteBook-8570w:~/GitStuff/codable_media_mashup$ java -jar Comm.jar examples/simple_example.comm
[*] Video Definition
        Filename: banana_phone_clips.mp4
        Cache:    BananaPhone
        Path:     ./comm_caches/BananaPhone
[*] Video Definition
        Filename: duckSong.mp4
        Cache:    duckSong
        Path:     ./comm_caches/duckSong
[*] Video Definition
        Filename: dance_moves.mp4
        Cache:    BananaPhone
        Path:     ./comm_caches/BananaPhone
[*] Saving run script to
        ./comm_caches/BananaPhone/RUN_banana_phone_clips.bash
[*] Running the script... Please be patient! If necessary, you can 'cat'
    the log to the terminal to see what's happening. The logs are located at:
        ./comm_caches/BananaPhone/RUN_banana_phone_clips.bash.log
```

The image above shows what a successful CoMM execution looks like.

```
nkarasch@nkarasch-HP-EliteBook-8570w:~/GitStuff/codable_media_mashup$ java -jar Comm.jar examples/error_handling.comm
line 14:32 mismatched input '.' expecting ';'
line 15:24 token recognition error at: '!'
line 20:0 missing ';' at 'var '
line 22:8 extraneous input 'what' expecting {<EOF>, 'add', 'var ', 'requestVideoCredentials', 'config', 'CoMM '}
line 22:35 token recognition error at: '?'
line 22:72 token recognition error at: '?'
line 28:29 mismatched input '"String literals not allowed here"' expecting VNAME
line 33:30 no viable alternative at input '1'
line 33:31 token recognition error at: ':'
line 34:28 mismatched input ')' expecting ','
line 35:12 no viable alternative at input 'add()'
line 36:36 mismatched input ',' expecting ')'
line 39:8 extraneous input 'noCache' expecting {<EOF>, 'add', 'var ', 'requestVideoCredentials', 'config', 'CoMM '}
line 40:22 mismatched input ';' expecting '('
line 3 - CoMM noContent;
  This CoMM definition is empty!
line 10 -  CoMM myFirstVid;
  The filename 'myFirstVid' has already been used in this file!
line 17 - var fail=variableDoesNotExist;
  The variable 'variableDoesNotExist' does not exist!
line 31 - add(validUrl,nonExistentVariable,"0:30");
  The variable 'nonExistentVariable' does not exist!
line 32 - add(validUrl,"Invalid Time","1:00")
  Time strings must be in the format 'minutes:seconds'.
line 33 - add(validUrl,"1:00",135)
  A time string was missing or invalid.
  Time strings must be in the format 'minutes:seconds'.
line 34 - add(validUrl,"1:00")
  A time string was missing or invalid.
  Time strings must be in the format 'minutes:seconds'.

nkarasch@nkarasch-HP-EliteBook-8570w:~/GitStuff/codable_media_mashup$
```

This image shows some of the various error handling capabilities of the program.