FlashGen

The Flashcard Generator

Brendan Cicchi

Remington Maxwell

Sara Woods

CMPS 112 – Comparative Programming Languages

Winter 2015

Cormac Flanagan

Table of Contents

Title Page ………………………………………………………………..…………………........ 1  
Table of Contents ……………………………………………………………………………… 2

Motivation ………………………………………………………………………………………… 3

Comparative Programming Aspect …………………………………………….……… 3

Parser ……………………………………………………………………………………………..… 5

GUI Design ………………………………………………………………………………………… 6

Conclusion ………………………………………………………………………………………… 9

# motivation

The idea for FlashGen, the flashcard generator, arose from our struggles using different flash card creators throughout high school and college. Originally Microsoft Word was used to type flash cards and then the settings on both the computer and the printer would have to be changed to print them out. After seeing the inefficiency of this, online flash card creators became the obvious choice. It was still very frustrating to spend hours typing up and formatting notes just to create virtual flashcards. This led us to the idea of a time-saving way to create flashcards without any additional effort: FlashGen.

We thought it would be great if there was a way to be able to type up notes once and then to instantly have those notes become flashcards. Since we had all taken CMPS 104A, Compiler Design, and had written a parser for a made-up language, we decided to apply this same mentality to our flashcard generator. Users would type up their notes using syntax that our program could parse, producing flashcards. We wanted this syntax to be easy and intuitive so that it wouldn’t interfere with the student’s typical note-taking process. We ended up using just three symbols: ~ to indicate a question, | to indicate options for the question, and > for the answer. The parser will be further discussed in Section 3.

Technology is constantly evolving and we are now living in a world where a student can use a laptop or a tablet to take notes. Our application enables students to generate flash cards directly from their notes drastically cutting down on preparation time that could be used for studying. We hope this encourages students to create flash cards throughout the semester compelling them to actively engage in their classes.

# comparative programming aspect

Instead of writing our code in multiple languages we decided to take a different approach and attempt to combine languages to see how well they work together. Using Haskell and Python allowed us to capture both the functional and imperative styles of programming in our project. The reasons behind choosing these languages for their specific roles in our project is further explained in their corresponding sections. An added bonus of using Python allowed for two of the group members to learn a new language. While Haskell is an excellent language to use for parsing, and Python is great for GUI’s, configuring the two programs to talk to each other proved to be difficult. No one on our team had any experience “combining” two languages so lots of research was required to decide how to approach this problem.

Originally we decided the best method was to create a dynamic-link library (DLL). The DLL would use c-types, a form of compatible data types, to interface between Haskell and Python. The main code would be in Python where the GUI was generated which would call Haskell functions as needed to parse the input. To achieve this, the Haskell program would have to import the Foreign.C module. Then every function that we desired to be callable from Python would have to be defined to have a type “foreign export stdcall funcName :: type -> IO type”. This would have provided an easy way to have our helper functions be “private” and not accessible from Python. After the correct type was specified in Haskell, we needed to create a C file which would act as the entry point for the DLL. In order to compile Haskell with the C code to create the c-types, we needed to use the –c flag for GHC. This would generate \_stub.o files which needed to be accessed to create the DLL. From here, we just needed to import the c-types into the main Python program for use. This would have been a clean way to link our two programs if the Haskell compilation produced the \_stub.o files; unfortunately, the latest version of Haskell compilation decided to merge the \_stub.o files into the main .o files to reduce generated clutter. There is no way to access these files needed for the DLL creation without reverting to an older version of Haskell, which was not a practical choice for a public use application.

Having failed to use DLL’s, we considered using a C program for main(), from which we could call Haskell and Python respectively without creating a DLL. However, the use of C to integrate Python and Haskell appeared to be an overly complicated method. Due to the GUI generation originating from the Python program, it makes more sense for our main() to exist in the Python code. We wanted to be consistent with our original goal, merging the strengths of each of these languages.

After further research, we found that we could simply call the entire Haskell program from the Python program. This approach allowed us to blend our languages in a manner closer to what we had originally envisioned. The main program must tell the operating system to run the Haskell code once the user has selected a text file. The parser must then print the JSON flash card object to the standard output. The Python code will capture this output and save it to a variable which is accessed as needed for the GUI. Using this approach, our two programs could successfully communicate with each other.

# parser

Since we have worked so much with Haskell this quarter, our

group decided that we had to implement a portion of our project

using Haskell. The choices were between the GUI and the parser,

of which Haskell is clearly more suited towards the latter.

The parser is called using “runHaskell” from the Python

code passing it a text file path as an argument. The parser uses

the file path to pull all of the contents of the file. We implemented

a two-pass system. First we remove any extraneous text from the

notes which will not impact the flashcards. Using the Haskell

module Text.Regex, we then processed the remaining text to

retain all syntactically correct notecards. If a portion of the text

file does not satisfy the required format, our parser does not fail.

Rather, it ignores the input until it comes across valid input again

or the EOF. We intentionally designed the code this way since *Figure 1* – Sample Input

users are prone to errors, and crashing the program due to an option or answer being out of place seemed to defeat the idea of a simple flash card generator application.

We chose the symbols, ( ~ | > ), to represent a question, question option, and answer respectively due to their general scarcity in most documents. We accounted for any potential users who would like to use these symbols for Mathematics or Computer Science notes, by only acknowledging the first non-white space character of the line to be the symbol for the notecard. If the user types ~, |, or > anywhere else on the line, such as { a^n b^n | n >= 0 }, the parser will not confuse the symbol with an indicator for question, option, or answer.

After having parsed the text file, all that remained was to send the accumulation of questions, options, and answers back to the Python code to be utilized in the generation of the flash cards. We decided to send all of the gathered data back to Python in JSON, since it is widely used and Python contains libraries to decode the JSON string. The format of the JSON was the following:

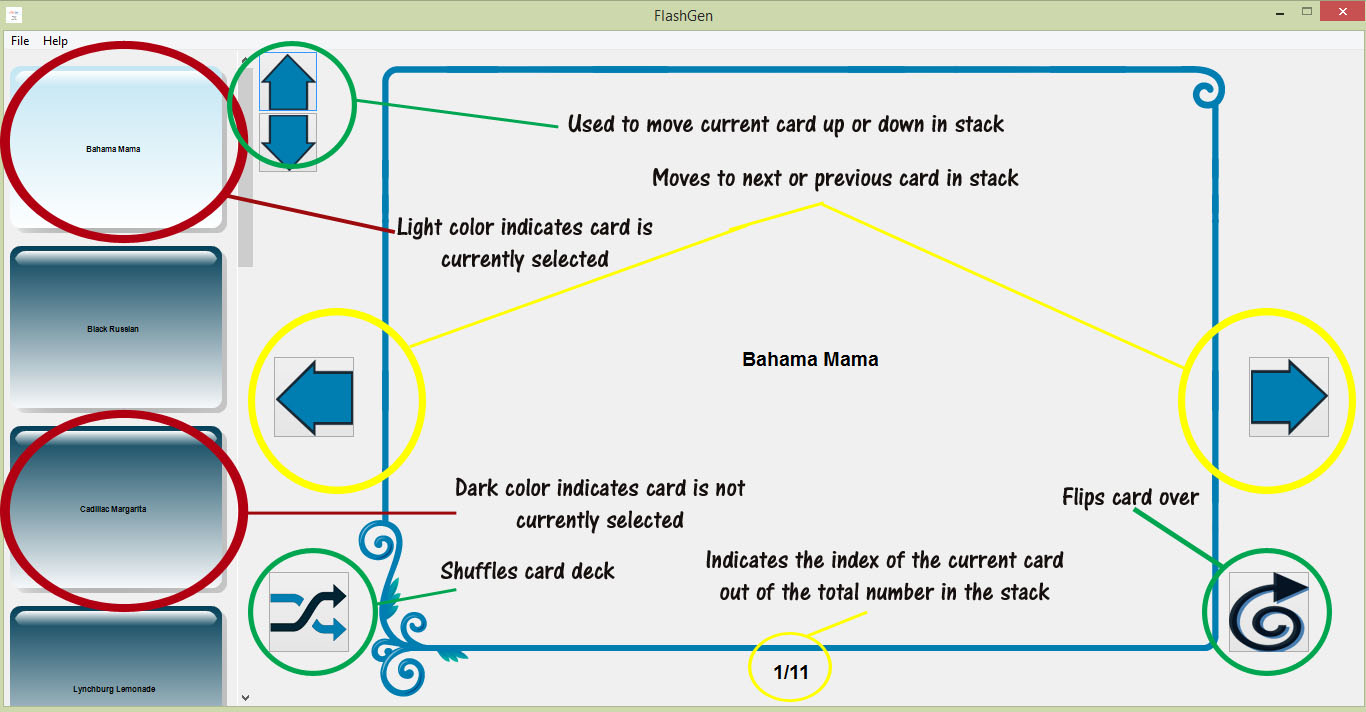
[ { “question” : "...",

"options" : ["option1", "option2",....],

"answers" : ["answer1", "answer2", ....] }, … ]

The parser code was only around 50 lines in Haskell to accomplish all of the above. Unfortunately, the parser was written five weeks before the project was due, in other words, prior to any knowledge of monads, other than IO monads. If we were to redo this project, we would have implemented a more monadic approach.

# gui design



*Figure 2* - FlashGen Main Screen Diagram

When we first started looking into Python GUIs, there seemed to be a wide variety of options to choose from. After considering the importance of the looks and aesthetics of our project, we narrowed our search to two options: WxPython and Tkinter. While Tkinter has a plethora of documentation and examples online, it used outdated graphics for the GUI. We wanted the appearance of our flash card generator to look fresh and new, not like the flash card creators currently present on the web. Unlike Tkinter, WxPython uses the current window settings of the computer’s operating system to display dialog boxes and windows. As such, WxPython was the optimal choice for our application.

We designed buttons for the next, previous, flip, shuffle, move up and move down features of our GUI (the yellow and green circles, respectively, on Figure 2). The previous and next buttons decrement or increment the index of the JSON object accordingly to display the correct question. The shuffle button imports Python’s random library to call random.shuffle on the list of cards. The flip button simply changes the face attribute of the cards object to the question or answer depending on what the current face of the card shows. To help the user keep track of what card they are currently on, the index is displayed as seen in the yellow circle at the bottom of Figure 2. To further improve the user experience, we added a feature for organizing the cards in the left panel. The up and down buttons, which can also be seen on Figure 2, relocate the current card so the user can implement a desired study order.

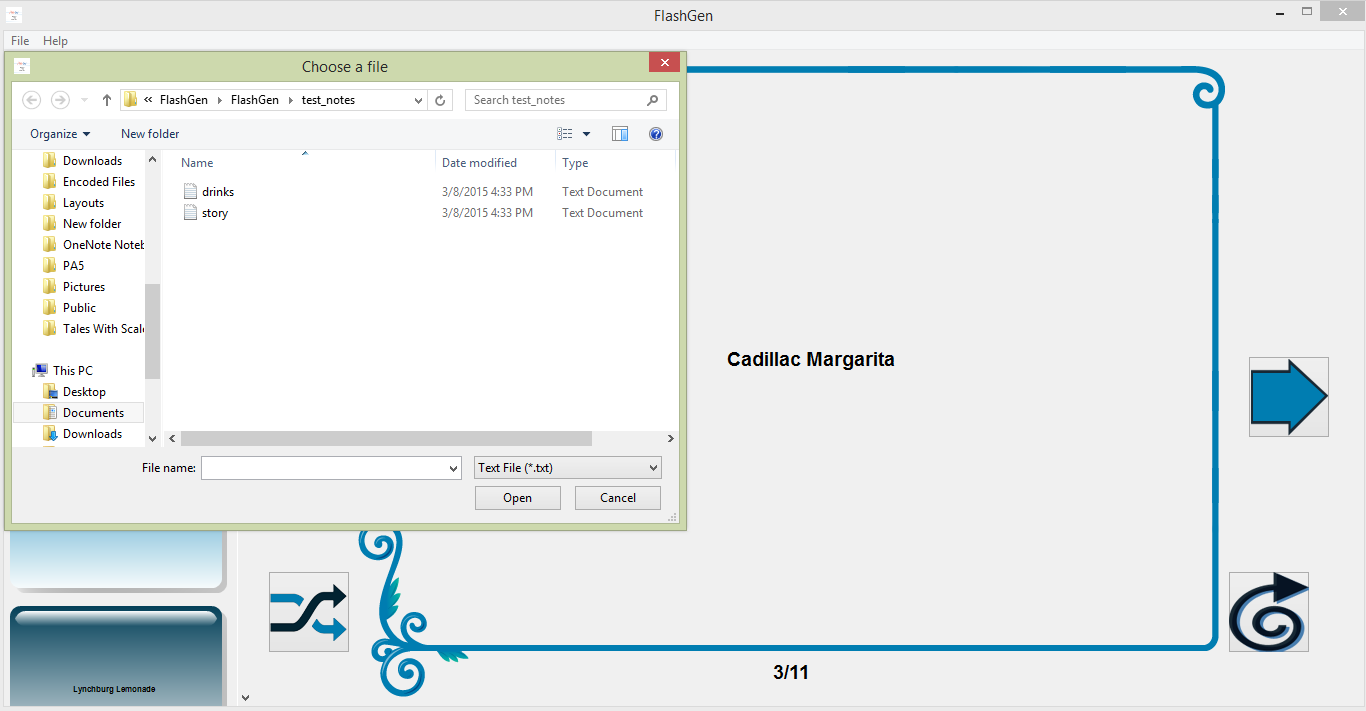
WxPython provides the ability to add images and tooltips to buttons to improve the user experience. The images for the buttons were created in PhotoShop Elements. Only informing the user of what the buttons do on hover instead of always displaying this information in the form of labels removes additional clutter from the display.

One of the most difficult parts of the GUI was figuring out how to implement the preview feature for the left panel. When designing an initial layout, we wanted to have more than just the current flash card be displayed. To give the user a sense of their progress when working through their flashcards, we decided to add a preview pane on the left hand side, similar to that of Microsoft PowerPoint. Our prototype tried to create the previews pane by displaying text boxes over the same static image that was used for the main flash card. This would have added flare to our GUI, but unfortunately, we weren’t able to determine a way to trigger a click event for these static images. As a result, we had to use buttons built into WxPython for the previews pane.

Although WxPython has many buttons to choose from, it does not have buttons with both an image and a label. This was very disappointing because we had wanted to match the flashcard previews’ backgrounds with the main flashcard background. We tried to manipulate text to appear over a button with an image, but the buttons became unclickable when a static text was layered over it. Luckily, WxPython contains Aqua Buttons, which allowed us to compromise with an aesthetically pleasing button in place of the desired background image. To add the preview buttons to the left panel we iterated through the cards object, formatting each question to appear on the appropriate button. This panel automatically creates a scrollbar if the number of flashcards displayed exceeds the allocated space.

A major challenge in the GUI design was figuring out how to properly format the text to display on the main flashcard. Since the length of the questions and answers varied significantly, we created a maximum line number to limit the amount of text on the notecard. To center the text on the notecards we used the WxPython boxsizers which enable us to manipulate the content of the page. Layering with boxsizers wasn’t any easier than the layering we had attempted before preventing us from being able to place the notecard image and the text in the same sizer. The solution was to have the image be statically placed on the page and only have the text dynamically change in the boxsizer.

On the application startup, a file dialog box appears requiring the user to select a .txt file. We chose .txt extension because it is universal to every platform. Once the file has been selected, Haskell handles the parsing and creates a JSON object, which is then used by the Python GUI to create the flashcards.

*Figure 3* - Open Dialog for Selecting Text Input File

To simulate a real world application, we added a tool bar with dropdown menus for File and Help. The File dropdown presents the user with Open and Close options. The Open option triggers a dialog box where a different text file can be selected, changing the flashcard display to a different set of flashcards (Figure 3). The Help dropdown provides the user with options for About and Help. The About option displays information about our team (Figure 4). The Help option demonstrates how to format the .txt file for FlashGen (Figure 1). The toolbar options all have an associated hotkey as well as icon to provide a more responsive interface. An icon representing our program is featured in the upper left-hand corner of the window screen and also appears in the task bar.



*Figure 4*: FlashGen About Us Dialog

# conclusion

While FlashGen does not demonstrate a true comparative nature between languages, our team wanted to create a usable application, which we could spend a lot of time making practical, yet fun. Originally we had planned to do our project twice, switching which language was used for the parser and which language was used for the GUI. The feedback from our project proposal suggested we approach the project using pair programming instead of delegating specific languages to different team members. Taking this into consideration we re-evaluated what we were trying to accomplish with our project. We decided that doing the entire application twice was a little redundant and impractical for the amount of time we had allocated, especially since each team member averaged 65 hours’ worth of work. Our GitHub totaled over 100 commits, with a consistent effort beginning 5 weeks from the due date. In the end, we were happy with how our project demonstrated the strengths of different languages and that we have a fully functional study tool for the final.