FlashGen: The Flashcard Generator

Brendan Cicchi

Remington Maxwell

Sara Woods

# motivation

The idea for FlashGen, the flashcard generator, arose from one of our teammates, Remington, struggles using different flash card creators throughout high school and college. She originally used Microsoft Word to type her flash cards and then would have to change settings on both the PC and the printer to print them out. After seeing the inefficiency of this, she moved to online flash card creators. It was still very frustrating for her to spend countless hours typing up and formatting her notes so that she could create virtual flashcards. This led us to the idea of a time-saving way to create flashcards without any real additional effort: FlashGen.

Since one of the frustrations was needing to type up the flashcards whether the notes were taken by hand or typed, we thought it would be great if there was a way to be able to type up notes once and then be able to instantly have those notes become flashcards. Since we had all taken CMPS 104A, compilers, 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 there notes using a syntax that our program would understand and interpret, 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. These symbols were not typically used symbols, so they were not likely to confuse someone reading the notes and were also easily accessible on the keyboard. Allowing the user to also type notes that they didn’t want on a flashcard was also a major requirement that needed to be satisfied. We handled this by allowing any sentence not starting with one of the reserved symbols to not be handled by the parser. The parser will be further discussed in Section 3.

By allowing users to write notes and flashcards at the same time, we believe that users are more likely to type notes, since they will be able to generate a study tool while doing a task they would already be doing. We hope that FlashGen will be a tool that inspires students to become more organized with notes as well as make studying for exams more efficient.

# 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. Despite how ideal these two languages for their tasks, arranging 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 e 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 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. 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 need 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. But, 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 desirable.

After that attempt failed we toyed with the idea of having our main code be in the C program from which we could call the Haskell and Python functions as necessary to run our programs without having to create the DLL. But we decided not to go this route from the project mainly because it seemed to create unnecessary complication. The main result of our program is the GUI generated from the Python program and attempting to create this through C seemed like an excessive step. This approach was beginning to stray too far away from our original concept of trying to merge the strengths of each of these languages.

We felt the cleanest way to go about our project was to have our main program be located in the Python file since this is where the GUI generation occurs. But following this train of thought we were back to our original problem of not being able to call Haskell functions from Python. After some 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 fashion closest to what we had originally envisioned. It is a bit messier than calling individual functions though. 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. This approach successfully allowed our two programs to talk to each other. Again, this merge isn’t exactly how we had hoped to combine the languages but this may have been due to the fact the Python and Haskell just aren’t the most compatible of languages. We are content though that we didn’t have to resort to coding the entire project twice, once in Python and once in Haskell, as was our fallback plan.

# 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 takes a text file path as an argument, and is

called using “runHaskell” from the Python code. 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 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 to be the symbol for the notecard. If the user types ~, |, or > anywhere else on the line, the face of the card in the GUI will not be cut short by our parser. We wanted our notes to accommodate text such as, { a^n b^n | n >= 0 }.

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 as a 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. In the future, if we had taken the time to rewrite the parser, we would have implemented monads.

# gui design

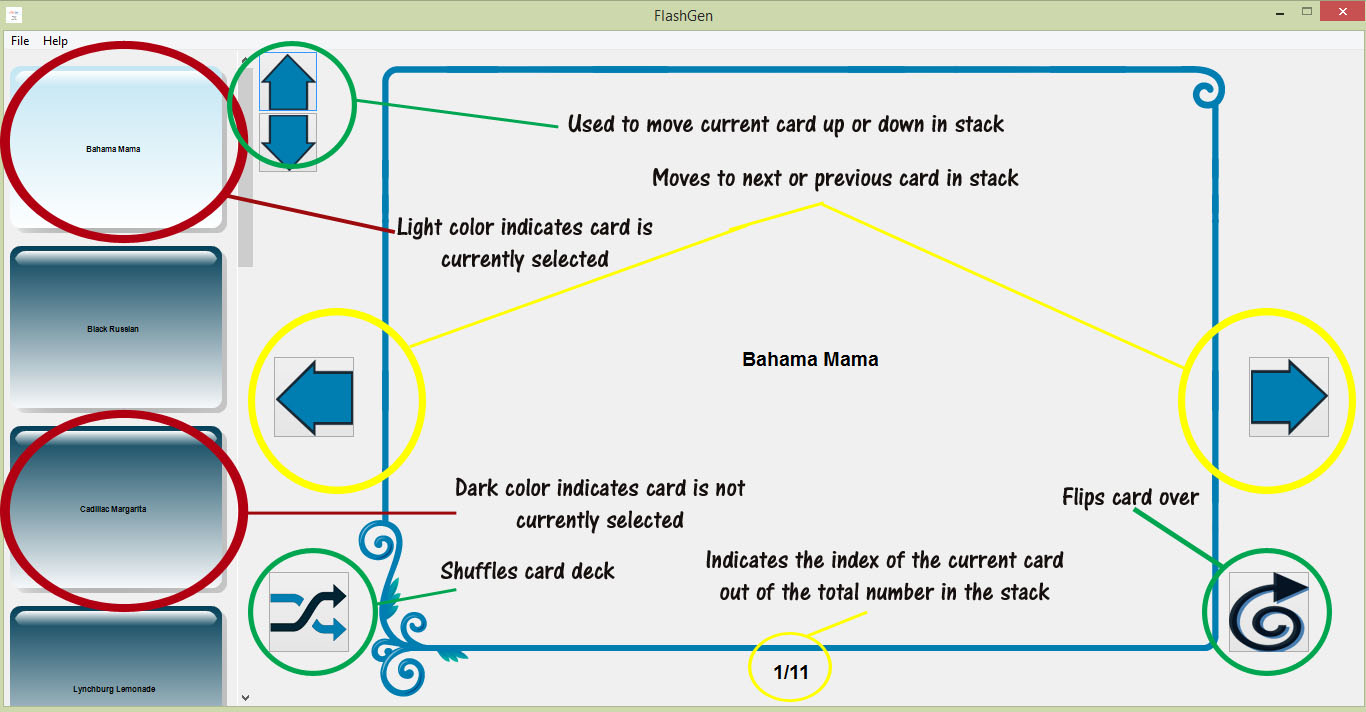


Figure 1: FlashGen Main Screen Diagram

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 asthetics of our project, we came down to two options: WxPython and Tkinter. Originally we thought that Tkinter was the better option because there was a lot of documentation and example found online, but upon working with it further, the looks of the windows and text created by Tkinter looked outdated. We wanted the appearance of our flash card generator to look fresh and new, not like the current flash card creators currently present on the web. This caused us to turn to WxPython, which uses the current window settings of the computer’s operating system to display dialog boxes and window. The difference between these two GUI frameworks was essentially the difference between Windows 95 and Windows 8. After starting to work with WxPython, our project was well underway.

When first designing what the program would look like, we thought about what widely-used programs were similar to ours. The closest that came to mind was Microsoft PowerPoint, so we incorporated aspects of this program into our design process. We liked the idea of making the program appear more like an app, with a cleaner appearance and less clutter on the screen. This would make the program seem more user friendly. On the left hand side, we also wanted to show “previews” of what flashcards were currently in the deck. We thought that this would be a great feature to add because if someone was looking for a particular card, they could easily find it whereas in other flash card creators, one must click “next” multiple times to find the card. With printed out flashcards, it can be very tedious looking for one specific flashcard. With this planned setup of having previewed flashcards on the left and the current flashcard largely displayed on the right, we determined it would be an intuitive and familiar user interface. The blue color scheme was inspired by Microsoft Word and a consciousness to color blind users. By using darker and lighter shades of blue to clearly distinguish between active cards (see red circles on Figure 1), a color blind user can also take advantage of this feature.

For the flashcard, functions were added to display the text from the JSON object. These functions included a “previous” function for pressing the previous button, a “next” function for pressing the next button, a “shuffle” function for pressing the shuffle button, and a “flip” function for pressing the flip button (see yellow and green circles, respectively, on Figure 1). For the previous and next functions, the index into the JSON object was shifted left or right, respectively. For the shuffle function, the Python shuffle function was used to shuffle the items in the JSON object. For the flip function, the index was changed to access the “answer” string of the JSON object. We also display the card’s index in the stack and the total number of cards in the stack under the card (see yellow circle on bottom of Figure 1).

The variety of buttons present in WxPython, especially buttons with images, was an important aspect when we decided to move from Tkinter to WxPython. Not only did WxPython have more button types, but it also had the added ability of button tips, displaying information on what a button does when the user hovers over it. This was essential to our “clean” design by allowing us to not need to add labels to the buttons on the design. For the button images, images were created in PhotoShop Elements and then colored to match the theme of the program design.

One of the most difficult parts of the GUI was figuring out how to implement the preview feature. We originally thought that displaying a text box over a static image, the same image as the main flashcard, would suffice. After working with this implementation, however, we found that there was not any easy way to trigger an event after clicking on the previewed flashcard. We wanted to be able to click a flashcard preview and to have the main flashcard displayed update to show the new text. This led us to thinking about using buttons for the previews. 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 WxPython has problems with layering objects on the GUI design. Thus, even though we were able to get text to appear over the button, the button was not clickable. This was extremely frustrating so we looked into other possibilities and found the Aqua Button, which was a button import. We decided on this particular button because the gradient it provided on the button was customizable and flashy.

After implementing the previews using buttons, we added a for loop that would iterate through the JSON strings, trim the text of the question so that it would fit properly on the button, create a button for each element and add the trimmed text, and then added all of these buttons to the left side. This left side consisted of a WxPython Panel, which had the added ability of adding a scrollbar if there were more flashcards than could be displayed in the provided space.

Another major challenge in the GUI design was figuring out how to properly implement the text to display on the main flashcard. Since the size of the text of the questions and answers could greatly vary, we created a cutoff point for the amount of text that could be added to the notecard. Despite adding this cutoff point, we still needed to figure out how to center the displayed text on the notecard with a variety of sizes. This led use to WxPython boxsizers, which allow centering of items by putting them into boxsizers. Boxsizers can be horizontal or vertical, can be nested, and allow organization of objects on a page. At first these seemed to be very useful and simple to understand, but they were very difficult to work with. Because WxPython doesn’t like layering objects, it doesn’t like to layer anything with boxsizers either. This meant that we couldn’t add our notecard image and notecard text to the same boxsizer otherwise the notecard text would not appear over the notecard image. After trying different combinations of vertical and horizontal boxsizers to center the text, we eventually settle on the text being added to its own boxsizer, which was a vertical boxsizer nested in a horizontal boxsizer. This boxsizer dynamically adjusted the location of the text based on the amount of text, centering it over the notecard image.

For reading in the text file, upon running our Python program, a dialog box is opened where the user can select a .txt file to read in. We decided to just have the type of file that FlashGen recognizes by a .txt because it is recognizable by any system, allowing users to easily share flashcard files with one another. Also, the flashcard files can be easily edited in any text editor. 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.

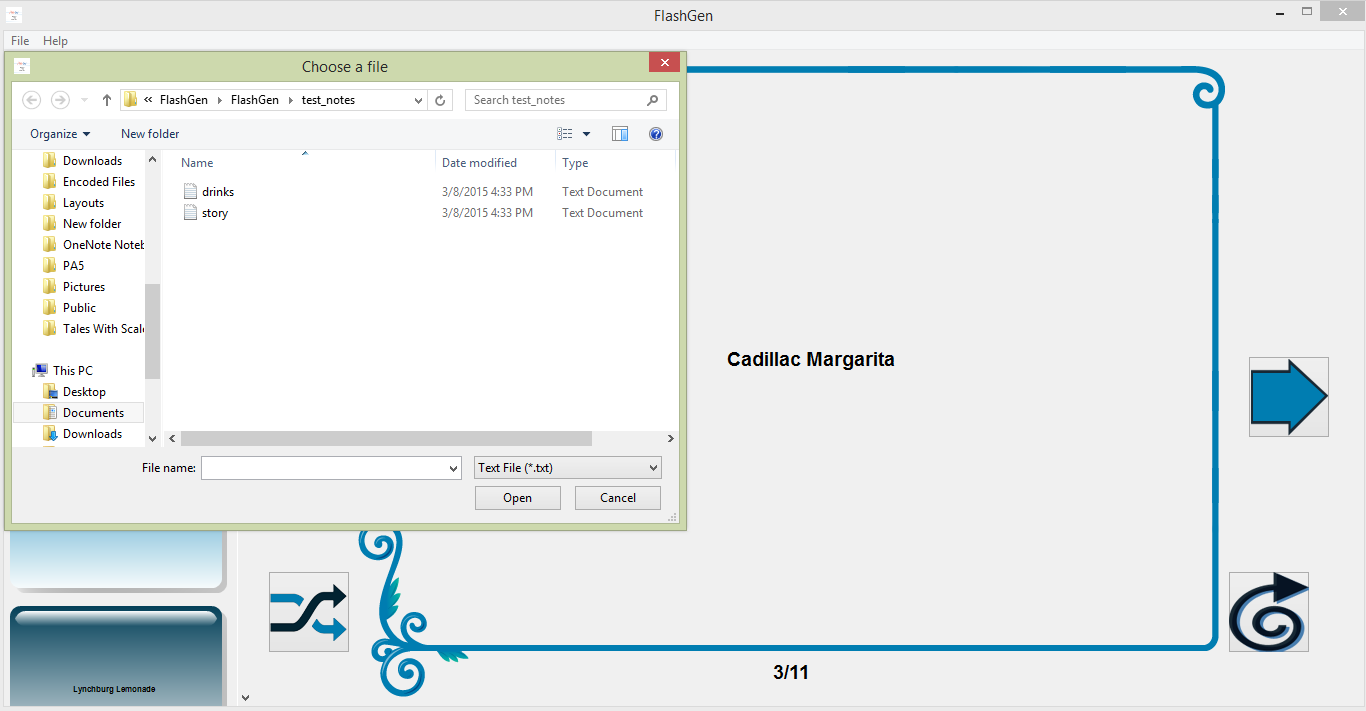
Once we finished all of the aforementioned aspects of our design, we decided to add a feature for organizing the cards in the stack. We added the up and down buttons to move the currently view flashcard up or down in the stack. In this way, the order of the flashcards could be determined. We also added a File Menu with the options Open, Help, About, and Quit.

Figure 2: Open Dialog for Selecting Text Input File

The Open option opens a dialog box where a different text file can be selected, changing the flashcard display to a different set of flashcards (see Figure 2).

The About option displays information about our team (see Figure 3).

The Help option displays information on how the buttons work in the program and how to properly write a text file for input into the program (see Figure 4).



Figure 3: FlashGen About Us Dialog

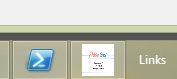


Figure 5: FlashGen Icon in Taskbar

Figure 4: FlashGen Help Dialog

The Quit option exits the program. An icon representing our program is featured in the upper left-hand corner of the window screen and also appears in the task bar (see Figure 5).

# conclusion

