Evil Hangman

Reid Anetsberger
Michael Parker
CMPS 112
3/21/14

Project Overview

The goal of our project was to create a game of hangman that would be very difficult, or in some cases impossible, to beat. Nobody wants to play a game that's blatantly unfair, so Evil Hangman had to cheat invisibly. To do this the computer adversary avoids the player's guesses, but in a way in which the player will not be able to notice. This is accomplished by utilizing the entire English dictionary, and progressively reducing the words which fit the required parameters: word length, containing certain letters, and matching the position of letters.

We implemented Evil Hangman in Haskell, C++, and Python. We chose Haskell because it is what we are learning in the class, and because it is purely functional. We chose C++ because we are familiar with the language and it is an imperative language. Lastly, we chose Python because it is somewhere between the functional and imperative paradigms of Haskell and C++.

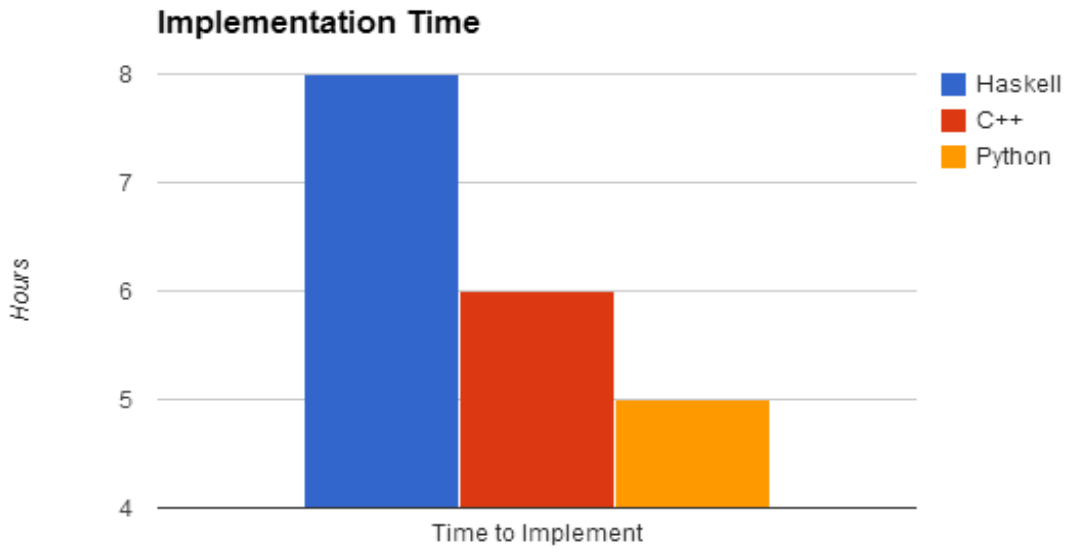
Design

Evil Hangman maintains a list of every word in the English dictionary; we will simply call this list "the dictionary," as it is used in every part of the program. The game begins by asking the player for the length of the word they will guess. The dictionary is then filtered to contain words only of this length. Next, the player is prompted to begin guessing letters. Every time the players guesses a new character, it removes words which contain that letter from the dictionary. The player loses a guess and it appears that he or she simply guessed the wrong character. Eventually the player may guess a letter that is in every word of the dictionary; therefore, a

random word is selected from the dictionary and the letters that were just guessed within that word are exposed. Since Evil Hangman does not cheat blatantly, we must do some additional work to maintain this illusion. The letters that are exposed to the player must be the same for every word remaining in the dictionary. For example, if the player has “f--bar”, then the computer must filter out all words that do not start with “f”, end in “bar”, and has two letters which have not been guessed between the “f” and “bar”. This process of filtering words by letters and position of letters continues until either the full word has been exposed and the player wins, or the player runs out of guesses and hangs!

Implementation in Different Languages

Since we were not very familiar with Haskell when starting the project, this language proved to be the most challenging to work with. There were some parts of the program that we knew would be simple, even in Haskell, such as filtering out the dictionary by length, letter, and letter position. However, the imperative methods of looping and mutable variables is not an option in Haskell. We weren’t familiar with file IO, the IO monad, and functional control logic at all when starting this project. These parts took the most time and effort to get right. The functional solution, of course, involves do blocks, binds, and a recursive programming model that isn’t obvious to an imperative programmer.



It was very convenient to be able to test our Haskell functions using the interactive interpreter in GHCI. Being able to immediately test code GHCI is certainly an attribute to the language, and accelerates the programming and debugging process greatly.

There were a few obstacles and drawbacks in the C++ implementation, which mostly had to do with performance while building and iterating through lists of around 125,000 words. First, we tried using the STL List to store the dictionary and iterate through it to remove words and was several times slower than the Haskell or Python implementation. Next, we attempted to use the STL Vector, but it was even slower than using List. After that, we decided to just use arrays to store the dictionary. With an array data structure, the C++ implementation was able to run at the same speeds as the Haskell and Python implementations. However, this also caused the C++ implementation to use up more memory than the others, as it needed to have several large arrays allocated to store different versions of the dictionary. This caused the C++ executable to use about 5MB of memory, while Python used 3MB and Haskell used less than 1MB. Haskell's surprisingly good memory efficiency can be attributed to it's lazy evaluation of the dictionary list.

The development of the C++ and Python implementations of Evil Hangman were rapid compared to the Haskell implementation for several reasons. First, we already had a working algorithm for the program from the Haskell implementation. Secondly, we were already very familiar with C++ and Python. Lastly, a side effect that we did not expect was that we found ourselves writing Haskell-like code in C++ and Python. We were able to conceptualize the different pieces of code making up the whole program in an efficient way (the Haskell way), that we wouldn't have been able to otherwise.

The Haskell Effect

One of Haskell's greatest attributes is its elegant and concise syntax. Haskell uses high level concepts to define what things are rather than tell the computer what to do. Functional programming allows the programmer to conceptualize the program

as a series of transformations on data. Each step in the series can usually be expressed succinctly as some composition of functions. This was especially true for the Haskell implementation of Evil Hangman. The language is fantastic for working with lists, which was especially useful for manipulating the dictionary in our game. Since we created Evil Hangman in Haskell first, we found ourselves unitentionally writing Haskell-like code in the C++ and Python implementations. For example, a common operation was filtering the dictionary for words not containing a certain letter.

```
filterLetter :: Char -> [String] -> [String]
filterLetter c strs = filter (c `notElem`) strs
```

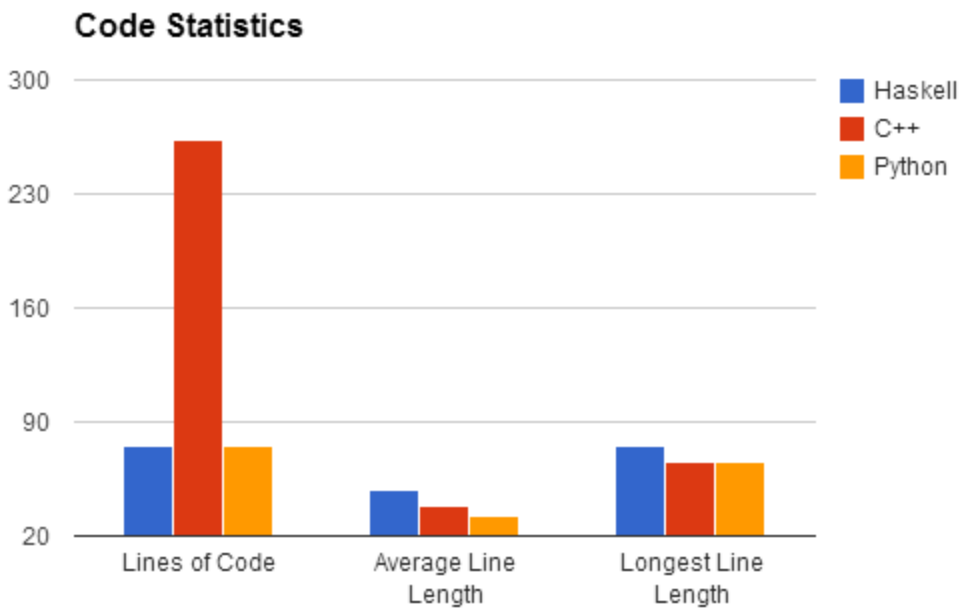
This task is completely trivial in Haskell. However, the same code in Python, a language which emphasizes short and clear syntax, is comparatively very verbose.

```
def filterLetter (c, strs):
    new = []
    for s in strs:
        if not elem(c, s):
            new.append(s)
    return new
```

The C++ implementation of the same function just feels unnecessarily verbose.

```
void hangman::filterByLetter(char l) {
    list<string> *tempList = new list<string>;
    for(list<string>::iterator it = cur->begin(); it != cur->end(); it++) {
        if(!elem(*it,l)) {
            tempList->push_back(*it);
        }
    }
    if(!tempList->empty()) {
        delete cur;
        cur = tempList;
    }
    curWord = cur->front();
}
```

So, does all that extra verbosity in imperative languages have some huge benefit over functional languages? This is not an easy question to answer...



C/C++ allows direct control over memory and is extremely efficient in the hands of a skilled programmer, but is very prone to mistakes and bugs. Python is extremely easy to develop, fairly portable, and is almost universally supported, but has high performance penalty. Haskell has the most succinct syntax, is extremely fast and efficient, but requires a more skilled programmer. Haskell has the additional benefit of practically eliminating the debugging process with pure functions and the interactive GHCI.

Conclusions

Looking back on our project, there were some things that we did well and some other things that we could have improved upon. We felt our choice of the project was good because it was not too large of a project, and we were able to explore the differences between imperative and functional programming languages. However, once we had finished our Haskell implementation, which was the largest unknown in the project, we did notice that the project was quite a bit easier than we had first anticipated. We overcame the challenge of creating a full program in Haskell, and in

the process we improved our usage of other languages by conceiving the program in the functional way. Finally, we were able to illustrate the meaningful advantages and disadvantages of each language in the software development process.