Kevin Zhang

Honors Project I Report

Description of Swift

Swift is a programming language developed by Apple Inc mainly for the use of software applications in Apple products. It is designed to be an easy-to-use and simple programming language.

Swift is a language that is descended from Objective C. Before Swift, Apple used Objective C for its iOS applications and other software applications. However, Swift eventually replaced Objective C as the main language for Apple applications, although a lot of old software still runs on Objective C.

Swift fits in with Apple Inc's ideology of simplicity. Because of this, Swift is good as a programming language for beginners who are new to programming. It is easy to learn, simple to use and makes its functionalities as simple as possible. I have a good personal experience of learning Swift. Throughout my programming career, I have had to deal with many hard-to-learn programming languages. However, Swift proved to be one of the easiest to learn.

Despite all the advantages of Swift, it does have its disadvantages. One of its biggest disadvantages is that it is still a relatively new programming language that is not in widespread use outside of Apple products. Swift still lacks a lot of libraries to provide extra functionality. For example, you can definitely create a hello world program on Swift, but you can't open or write a text file in Swift the same way you could in Java or C++. This makes it difficult to implement Swift in a lot of applications like web applications. Therefore at least for now, Swift is mainly useful only if you are working on Apple technology.

Description of Racket

Racket is a very interesting programming language. The name Racket actually has multiple meanings to it. It can imply the programming language itself, or it can refer to an entire family of programming languages descended from the Racket language. Similar to how the coronavirus evolved into many different variants over time, the Racket language also changed into many different variants of the same language with similar properties. The Racket name can also refer to the tools for the programming language like the compiler rather than just specific to the language itself.

Racket is very different in syntax from Swift and other languages like Java and C++. If you master Java, you can easily master other similar languages like Swift. This is because they all share a similar syntax format which makes it easier to transition to learning each other. However, Racket has a completely different syntax format compared to these programming languages, therefore it is harder to learn. It's just like how if you had already learned English, it is a lot easier to transition to learning German instead of learning Japanese since German uses the same alphabet but Japanese is completely different.

Unlike Swift which was developed by a company for its own profits and benefits, Racket was not developed by a company. It was created by a research group. It was created for the purpose of being a programmer for programming language. That is, it's supposed to be a programming language which you can create all the programming languages from it. This is why they are so many different variants of Racket.

There is very little commercial use of Racket. It is not designed to be a commercially viable language and therefore very few if any companies widely adopt it. Its main purpose is to have a place in programming language theory. Despite having a large arsenal of libraries that allows it to cover everything from GUI to web development, it is not even close to being as popular as the famous languages like Java and C++. However, there is a lot of potential for Racket and I expect that it should grow in popularity eventually.

Timeline of Project

At first, this project was slow going for me. The hardest step was the first step. The first step of course is figuring out what is the best place to start. Even though I had already proposed a Swift to Racket compiler before enrolling in this honors project course, one of my biggest questions is how exactly am I supposed to do this? What operating system is this supposed to run on? What kind of integrated development environment do I need to use to do something like this? These questions held me up for a long time but eventually, I was able to figure it out.

There was also the hurdle of getting this course approved as an RHED entrepreneurship course. This was not an easy and straightforward process, because they had to go through the approval of many different people. First of all, the RHED leader Vanessa Farzner had to approve it, and then the RHED committee needed to approve it in order to finalize the approval. However, thanks to the help of Professor Kevin Willet, I was able to create an entrepreneurship write-up and then send it to Vanessa. Vanessa was then able to get it approved by the committee. Thanks to Kevin W., when Vanessa left UMass Lowell in Spring 2022, she was able to finalize the approval for the Honors Project I. The only remaining objective I needed to do was to get a passing grade for this class.

Eventually, I was able to overcome the difficulties of thinking about how to implement this program. I eventually settled on Visual Studio as the integrated development platform to use for this project. I decided to use C++ for this project because it is the programming language that I am most comfortable working with. It is also the programming language that is most suitable for this project due to its unique nature.

I also had difficulties figuring out the Swift and Racket compilers. For the Swift compiler, it was particularly difficult. Initially, I wasn't even sure how to download it and where to download it from. I eventually decided to use a compiler that someone else posted on GitHub that is easy and convenient to use, rather than use the compiler from the official Apple website. This proved to be the right choice since it was easier to use the GitHub compiler. I actually tried to tweak the Swift compiler a bit in order to fix an issue related to command-line running a Swift program, however, it turned out that I would risk damaging the compiler and rendering it unusable on my computer. Not only will this ruin the Swift compiler, but even if I uninstall and reinstall it, it still wouldn't work due to remnants of the old program on my computer, which would make this computer forever unsuitable for the Swift compiler. I made that mistake once and it took a lot of time and effort to fix the issue. I made myself promise that I would never do anything that would risk damaging the Swift compiler. I have a feeling that Apple makes it difficult to run Swift on Windows because the company does not like people to use non-Apple products. After all, they are a closed company with a closed ecosystem.

Installing the Racket compiler went smoothly. Windows seems to be a friendly operating system to Racket, therefore the installation happened cleanly and everything went as smoothly as it was supposed to be. Figuring out how to use the Swift and Racket compilers took some time, but I was eventually able to master them.

Then there was also the process of figuring out how exactly to understand the Sil code that is translated from Swift as the first step of compilation. When I first looked at the code, it appeared to be very hard to read and very hard to understand. The first thing I noticed when I looked at the Sil code is that it looks a lot like the low-level language code that I've learned in Assembly Language class. Since I do have some background experience in the Assembly Language, it took a lot less time for me to understand it than if I had never learned assembly in the first place. Once I understood it, I am able to simplify it and make use of it to help me in the translation.

There are two main parts to the program, one part is the command, the other is the translator. I started out with implementing the command part of the program because it was the easier and more straightforward part to implement. Doing this required more creativity than skill. Since the command part of the program is the part that interacts with the user and allows the user to control the program, it was more enjoyable to implement this part of the program than implement the translator.

The translator part of the program is the more difficult part to implement. This is mainly due to the fact that it involves close the examining and working with a lower-level part of Swift. The more low-level you get with programming languages, the more difficult it becomes for the human eyes to read. At first, I was at a complete loss at what am I even supposed to do to get started. Nevertheless, through my perseverance and hard work, eventually, I was able to successfully implement the translator, despite lacking some functionalities.

Compare and Contrast Swift, Sil, and Racket

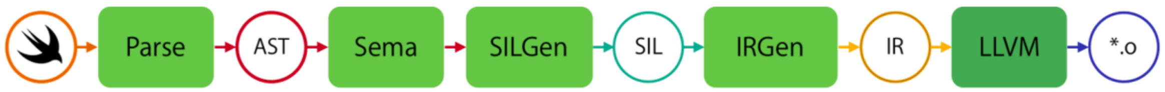
I show it on my presentation PowerPoint. I've decided not to show it on my report since it is harder to display the comparison here.

NOTE: The Sil language is only displayed in its spliced form after removing all but the main function of the Sil file.

Closeup of Swift Compiler

The Swift compiler is not just a piece of software that converts the code you have written into what it is supposed to do. Instead, it goes through a series of steps and layers before it successfully does what it's supposed to do.

Below is a diagram of the stages the Swift compiler goes through to produce your output.



The first and most important step of running the Swift code is to parse it. This will generate an abstract syntax tree or AST. It is a tree that represents the programming language syntax as a hierarchy structure.

Here is a link to a website that allows you to easily see the difference between Swift code and AST. https://swift-ast-explorer.com/

After the abstract syntax tree is generated, it is time to perform semantic analysis on the tree which will convert it into a Swift Intermediate Language. or SIL. It is a representation of the Swift program that shows how the data is stored in registers as the program is run. It somewhat resembles MIPS in assembly language, but the function syntax is completely different.

The next step to do for the Swift Intermediate Language is to transform it into Intermediate Representation. At this point, the code will become truly unreadable. It will be difficult, if not impossible to reconstruct the original program unless you have a decompiler. The intermediate representation is meant to represent a program in between the source code and the target machine code.

The LLVM will transform Intermediate Representation into binary machine code. This is the point where the machine becomes universal. What I mean by that is that the LLVM is no longer exclusively created by Apple and instead is specific to each individual machine on which the program is run. The LLVM originated from a research project at the University of Illinois and is not exclusive to Swift itself.

The more low-level we go in the translation to more it is hard for humans to read it. By the time we get to the machine code, it's basically all just numbers and binary. It is not meant for the human eyes to read, it's only meant for the machine to use in order to run the program.

Closeup of Racket Compiler

There is less information available about the Racket compiler than for Swift. Racket translates the .rkt file into Racket bytecode in .zo file format. Bytecode is usually processed by a virtual machine instead of by the computer hardware directly as machine code. It is the equivalent of Intermediate Representation in Swift. The code is machine independent and is then processed into machine-dependent machine code for the hardware.

Description of Program

The program that I have created is designed to be easy to use and practical in the conversion of Swift to Racket. There are two main classes in the program. There is the command class and the translator class.

Command class

The command class is the program that is meant to directly interact with the user if they choose to run my program. It represents the business aspect of this project regarding user experience and how to make the program appealing to potential users. It is supposed to allow users to enter their input into the program and the program will respond depending on the input. If the user entered a valid command into the input, the program will follow the command, otherwise, it will return an error message and prompt the user to type their command again.

There are many options that you can select from in the command interface. For example, you can enter a command to see the help information or you can enter the command to compile and splice a Swift file.

Below I give a summary of the command features and their functions.

Command class variables

The string path is the only variable in the command that is meant to store the file path when you start the class up.

Command class functions

- command() - This is the class constructor. It first saves the file path from FilePath.txt into a string, then it keeps looping the main menu repeatedly until the user quits.
- void main_menu(int* q) - The main menu function keeps giving the user a prompt to enter a command. It executes the respective function depending on the command and keeps repeating the same thing until the user quits.
- void set_path() - this function allows you to set and change the file path in the FilePath.txt file.
- void splice(const char* file_name) - this function takes a Sil text file name as a parameter and performs splicing on the file to simplify it and only leave behind the Sil main function
- void compile_splice() - this function prompts a user to enter a swift file name, and the function will first partially compile it into Sil format, then splice it so only the useful part of Sil remains
- void racket_run() - this function prompts a user to enter a racket file name, and will run it and show its output
- void translate() - this function prompts a user to enter a Sil text file name, which it will then translate to Racket
- void help() - this function prints out the possible commands and their respective purposes
- void view_path_folder() - this function allows you to view the current file path
- void info() - prints out some information about the translator.

Translator class

The translator is a low-level translator that translates the lower-level Sil code into Racket.

The translator features different functions. These include the interpreter and the translator functions.

Translator class variables

Variables for the translator include the string vector output which is for storing the output, the Racket code. the string Path stores the file path that is being worked on.

The translator class also contains enum variables that represent the state that the program is in. Enums are also used for pinpointing the datatype of the variable.

There is a structure called swift_var that is meant to store the properties of the Swift variable like the variable name, datatype, and the value. The variable value is always stored in string format.

Translator class functions

- low_translator(string path, ifstream* file) - the constructor for the class stores the file path in the path string, and then it uses a while loop to go to the end of the ifstream file. For each line of the file, it goes into the main interpreter function
- void main_interpretor(string line) - the main interpreter is pretty simple. All right has to do is activate the variable states if it detects the declaration of a new variable. If the states are already activated, it invokes the swift_variable_interpretor function.
- void swift_variable_interpretor(var_state state, string line) - this function serves a very big role in the translator. If the translator is getting a variable name, it finds the length and location where it can read the variable name and reads it. It also finds out about the data type by checking in a specific location for a letter that changes based on the datatype. If the translator is getting the variable value, it uses a switch statement where depending on the data type, it has its own way of finding the data value.

- string racket_variable_translator(swift_var var) - it uses the data about the variable collected from the Sil file and generates a line of Racket code for variable declaration.
- void write_racket_file() - it writes the strings from the vector output into the Racket file.

Limitations of this Project

The original ambition of this project was to create a programming language translator between Swift and Racket that can translate any Swift program into Racket. However, on a closer look, it appears that it will be a lot more complicated and take a lot longer than I initially thought, therefore requiring slight amendments to the original plan. It's not impossible to create a perfect programming language translator, however, it will take a lot more than just me to be able to do that. That would be a project of such magnitude that it requires a team of programmers working full-time on this instead of just one person like me. Even with all that, it will likely take years before a complete translator can be developed.

The first thing I noticed is the complexity of the Swift Intermediate Language. Even if you only partially compile one line of Swift code, it will still produce thousands of lines of Sil. This was why I had to simplify it by splicing the Sil so that my program can only focus on its useful components. This may simplify the reading and analysis of the Sil code, but it also means you will be missing out on the other components that may be useful in certain circumstances. After all, there's always a reason for all those lines of code, which likely represent libraries or something else important. What I show in the presentation PowerPoint is merely a very small part of the Sil program. If I am doing something simple like declaring a variable, I only need the main part of the Sil, but something that is significantly more complicated like a function likely contains required information outside of the Sil main function. Given that along with the high complexity of the Sil files, it wasn't possible for me to implement all of this on my own within the timespan of the project.

Usually, once the Sil is generated, the code would then go into an intermediate representation generator or IRGen. This generator would generate the intermediate representation of the Swift code. I was able to look into the source code for the intermediate representation generator at this GitHub link https://github.com/apple/swift/tree/main/lib/IRGen and I saw just how complex and long the whole program is. It was an endless span of cpp and header files, each file hundreds or even thousands of lines long. It would have been completely impossible for a single student working on this as a school project to implement something like this on this magnitude that can completely translate Sil to Racket. Given that, the best path forward for me in this project was to create something nice and simple like a standard cpp program that translates some degree of Swift instead of using my time to try to attempt something impossible.

Business Aspect

One of the biggest problems that computer programmers have when it comes to programming is a lack of functionality for certain programming languages. Oftentimes, programming languages have aspects that they can do that other programming languages do not. This is a problem, because there is only so much that an average person can learn, they can't possibly learn the whole world's computer programming languages. At the same time though, they need all the functionality that they can get from their programming languages in order to maximize their usefulness. It is important in order to create a more technologically advanced society. Programming languages are different because they each have a different advantage. For example, the Swift programming language has the advantage of being more versatile and easy to learn than most programming languages. However, it is not a multi-paradigm and scripting programming language like Racket is.

The solution to this problem is to create a compiler that treats Swift code like Racket and outputs something like Racket does. So basically, the Swift and Racket programming language each have different syntaxes. Each of these programming languages has a different syntax and a different functionality depending on each of their individual syntaxes. The goal of the Swift to Racket compiler is that you can get the functionality of Racket by using Swift code. In other words, you can get the benefit of using Racket without having to know Racket syntax, as long as you know the Swift syntax. Think of it as a translator that translates Swift to Racket so that you don't have to know Racket in order to use it just like you can use a translator to speak English to French if you don't know French. Of course, I will need to figure out how to implement it in order to have it be of any use in the future.

In order to be able to successfully create a product that will be successful in popularity like the Swift to Racket compiler, it is important that I address certain aspects like the demand and the ways to have it take off. The most important question I need to answer is whether or not people are going to need it. For that, I am going to look at how popular the Swift programming language is and how useful Racket is. The Swift programming language is created by Apple Inc. It is meant mainly for Apple products like iOS devices, iOS app development, and other Apple development. This means that it is a popular programming language, but only specifically for Apple. Even though it is a multi-purpose programming language, for now people use it mainly for Apple products. This doesn't mean that Swift has no potential for applications outside of Apple Inc though, it has potential for being used for back-end websites and other applications. So this means that even though Swift is still a relatively new and obscure language, it still has a lot of room for potential. Another way to make this product successful in the market is to make it easy to use. Often times, people who are new to computer programming or I have never done something like this before need a beginner friendly platform to use. The compiler will also have to be available to use on multiple operating systems, most importantly Windows, Linux, and MacOS. Because in order to convince people to pay money for my software, they will have to think that it is worth their money, therefore, by allowing the compiler on multiple operating systems, it will help in making people think that their investment is worthwhile. It will also need a website for download so that people can navigate to that website and download my compiler from there. The website will need cool graphics that both catch people's interest and serve as a great first description and impression to the purpose of the compiler software. The website will also need catchy phrases and descriptions to supplement the images in order to give people something to read so that they can have a more solid understanding of what this website is about. These are just some of the innovative ideas for my future business. I will need to keep up with this innovative thinking when I actually get into starting this company.

Of course, there will be risks to launching the compiler as a marketable product. I will have to be careful not to make the wrong decisions or else I could end up wasting my entire investment and time for nothing. The first most important thing to do for creating this business is to start out small and take my time, don't invest too much of my money all at once without good research and judgment. I'll also need to make sure that all the money I invest is disposable so that even if everything completely fails, it wouldn't have a big impact to my life in terms of finance. To minimize the risk of my product failing after having invested so much time and money, I would need to do research and get feedback from people in order to see and verify whether or not this product really is going to be appealing to them. I could find some people and companies to call and ask to see whether or not they would be interested in paying for a Swift to Racket compiler, or I could use online surveys and find a way to send them out to people so that they can take the survey and give me data on how much is the compiler is appealing to them. Talking on computer science online forums like on StackOverflow is a particularly good way to get feedback. These are places on the internet where all kinds of computer science people ranging from beginners and experts congregate online to talk about pretty much anything that is related to computer science. I'm sure that I would get very good feedback if I ask people on these forums to tell me what they think. Hopefully, the results would turn out that at least some people support the idea of a Swift to Racket compiler.

I believe that even though my Swift to Racket compiler idea is a very new and uncertain idea, it has a lot of potential to grow and eventually become widespread in the computer science community. In the end, only time and effort will tell if I really am going to be successful in getting people to use it. However, developing an innovative plan and coming up with as many good ideas as possible for this project will boost the chances of it succeeding.

Future Development

Despite not having enough time to implement the whole translation before the end of the course, that doesn't mean that it is the end of this project. I believe that with enough work, this project can have more potential and be a more potential commercial success.

The first and most important thing that it needs is a graphic interface. The graphic interface is very important if I am to try to make this project a commercial success because prospective customers will not be interested in using my project if it doesn't have a user-friendly interface. Unfortunately, I do not have a lot of experience in graphic interface design. I've only done graphic design for programming in Java so it will take some time for me to learn how to do graphic design in C++ and apply it to this project.

It also has to have more functionalities that it can translate. So far, this program is only limited to translating simple stuff like variable declaration and some mathematical operations. In order for this project to attract any significant attention, it has to have the ability to translate more aspects of Swift into Racket. Therefore, I will likely look into this. I believe that it will be very hard and complicated to translate to more complex parts of Swift into Racket.

Last but not least, it needs a business plan and investment. Even if I determine that it is commercially viable in the future, it still needs work related to business development. I will need to do things like identify a target market and segment the market in order to figure out which population of the target market would be most likely to pay for this software. Of course, all of that is assuming that I determine that it's possible for this project to be profitable in the first place.

In the event that I determine that this project is not going to be commercially profitable in the future, I will probably make this into a nonprofit project. If that happens, it is going to be an open source project where anyone can contribute to this project to make it better. Even if it is not a commercial project, that doesn't mean that I don't need to raise money for it. Nonprofit projects need money and people in order to run, therefore, I will still need to figure out how to recruit volunteers and raise donations to fund this project.

What I Learned

From this project, I have been able to learn a lot about programming languages and business at the same time. Before I did this project, programming languages were in my point of view mostly just something where I had to follow a series of syntax rules and learn about different functionalities and they can do amazing stuff. Beyond that, they weren't any more special to me. This project had a huge impact on changing this perspective. I now understand that programming languages have a larger story.

I also learned more about Swift and Racket and became more familiar with low-level languages. Swift is a very useful programming language to learn if you are to go into iOS development. In fact, I would say it is even required because all iOS apps run on Swift. Racket may not be a language that will allow me to earn a lot of money once I master it, but it is a good foundation for useful knowledge in other areas. Computer programmers do not usually program in low-level languages but it is still useful for me to learn about it from this project to get a good sense and understanding of computer architecture and machine code and how exactly a computer program runs behind the scenes.

I also learned more about Visual Studio and how to use it. Before this project, I used Visual Studio earlier before in Computing I and II, however, that was because my professor was using it and wanted us to try it out, not because I actually found it useful. This project is the first time that I found myself enjoying using Visual Studio for my projects. Getting used to Visual Studio was not that easy at first because I first needed to install some libraries and add-ons to Visual Studio before I could truly start compiling and running C++ on it.

I also learned a lot about business. This project was not just a project where I create a working computer program, it was also an opportunity to explore its commercial viability. I used to attend a lot of events and competitions in UMass Lowell DifferenceMaker, and I was able to use the experience and skills gained from these events and apply it to this project. This project was a good exercise in business thinking for me that will greatly prepare me for future entrepreneurship.

This project was also significant in improving my abilities in project design. When I started a new project like this, I had to figure out how to design and implement it. This was a hurdle for me because in the past I would always get specific instructions on how to implement a computer program. However, this project is a situation where I have to both design and implement it. This takes the idea of software development to a whole new level.

Surprisingly, this project also taught me to be cautious when working with machines on a delicate level. The incident when I almost ruined and messed up the Swift compiler was extremely scary and convinced me to never try something like that again. Now I know how delicate the process of computer systems is and how it is like the human body, where a slight change to the human body in the wrong way can drastically alter its basic functions just like how a small change in the wrong way to a computer system can negatively affect it greatly.

Bibliography

Inc., Apple. “About Swift.” Swift.org, https://www.swift.org/about/.

Miller, Stephan. “What Is Swift Used for?” Codecademy News, Codecademy News, 13 July 2021, https://www.codecademy.com/resources/blog/what-is-swift-used-for/.

Pankaj, Pankaj, et al. “Low Level Languages - Advantages and Disadvantages.” Codeforwin, Codeforwin.org, 17 May 2017, https://codeforwin.org/2017/05/low-level-languages-advantages-disadvantages.html.

Racket, https://racket-lang.org/

Aggarwal, Ankit. “Swift Abstract Syntax Tree.” *Ankit.im*, 29 Feb. 2016, http://ankit.im/swift/2016/02/29/swift-abstract-syntax-tree/.

Pourhadi, Ali. “Swift Compiler: What We Can Learn.” *Medium*, Medium, 4 Nov. 2019, https://medium.com/@ali.pourhadi/swift-compiler-what-we-can-learn-96872ea4b1b8.

Mattt. “Swiftsyntax.” *NSHipster*, 22 Oct. 2018, https://nshipster.com/swiftsyntax/.

“What Does Racket Compile to?” *Racket Source*, 8 Mar. 2022, https://www.racketsource.com/articles/what-does-racket-compile-to-6c3ded17/.

https://llvm.org/

https://www.swift.org/swift-compiler/

https://developer.apple.com/swift/

https://docs.swift.org/swift-book/

https://www.swift.org/about/

https://docs.racket-lang.org/guide/intro.html

https://codedocs.org/what-is/racket-programming-language

https://docs.racket-lang.org/raco/Bytecode_Files.html

https://github.com/racket/racket