My Journey in Compiler Design

By Toby Duncan

December 8th, 2017

Compiler Design is rarely talked about in Computer Science, in fact it is almost shunned by the community for being complicated and painful to build, so against my better judgement, I decided to build one for my senior project. While it was not the most exciting piece of code I have ever built, the sheer rush of emotions I got when I printed “hi world” to the terminal screen was nothing short of incredible. However, it was a long journey to get to that final point. I slowly journeyed through each stage of compiler design, facing all the bugs and challenges in each stage, until I final reached the end and now I want to share that journey.

My journey started where all compilers start, which is picking a language to actually compile and implement. I juggled with whether it should be an esoteric language or a subset of Java for some time, weighing all the pro’s and con’s for both. It was extremely tempting to choose the subset of Java due to its familiarity and widely available sources, however, it was missing the heart of a senior project: to explore something new. After coding for almost seven years in Java, it was not the programming language that would let me explore new territory, so I settled on InSpace to be my language, and began my research.

My research started with how the InSpace language, developed by esolang user Zayne (Zayne 2017), actually functioned. After I looked over the website, I quickly realized that InSpace was best summed up as a bunch of different predefined functions. Each of these functions was meant to accomplish a task (print something, create variables, etc), and, like other programming languages, each function would only accept a particular input. On the site, each function had a short description for what it was meant to accomplish and do, but that was all. This was anticipated, as I specifically chose InSpace from the unimplemented section of the website. By taking an unimplemented language like InSpace and giving it an implementation, I would not only make the language usable, but provide other programmers a means to use the language. It was for this reason that I settled on InSpace as my language, as it would provide me with the opportunity to explore a new language that I was not familiar with, while also having a community impact and therefore leaving my mark on the coding community.

With the language researched, the next step was to begin researching the actual art of compiler design. The main parts of the research were how a compiler gets created and the best ways to take each strategy and put them to use in my own compiler. I enlisted the help of the infamous Dragon Book, that is widely considered the book on compiler design and theory. The Dragon Book, however, as I quickly realized, was not as clear as I had hoped, so I ended up pairing it up with other resources like. For example, an open source class from Berkeley, which like the Dragon Book, provided good foundation on the theory. From all these resources, I was then able to get a better understanding of compiler theory, and how the actual compiler should operate, which smoothly lead into laying out each stage of the compiler (Aho 2014 & Hilfinger 2009).

The stages of compiler design are fairly straightforward in that there are five stages (really six): lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, and finally code generation. All these steps combine to produce a target program, which typically is assembly code, that is then translated by an assembler and linked and loaded into the computer's memory. I will touch on these topics later in the paper, but I did want to discuss a few things now. Firstly, I wanted to say that I did not do any optimization to my compiler, and vouched to skip the step in its entirety. While optimization is for sure an important part in compiler design when trying to improve runtime, it is not necessary for a compiler to work. It also is one of the more complicated parts of compiler design, and it typically discussed in master level classes as a topic of its own. For this reason, my mentor and I both agreed that it was out of the scope of the project, and therefore I was allowed to skip the step (Aho 2014 & Hilfinger 2009).

Another part of the compiler that I skipped was that some compilers translate the language all the way down to machine language. This requires the construction of an assembler which takes generated assembly code and translates it to the 1’s and 0’s, so that the computer’s CPU can understand the code. Again, similar to optimization, this optional step is also complex and has its own class that it is taught in, which again led to it falling outside the scope of this project.

While optimization and building an assembler for the compiler fell out of the scope of the project, everything else was fair game. Therefore, the first step to building my compiler was the lexical analysis phase of compiler design. The goal of this stage was to take my source program of words and symbols that make up InSpace, and turn it all into various tokens. These tokens would then be used for processing the language and verifying that it was written correctly in later stages. Lexical analysis was also the stage of compiler design that the first version of the symbol table was generated.

A symbol table is what holds the data and information about all literals in the program and is typically stored in a hashmap. A literal in a programming language can represent many different things like numbers, strings, and variable names. My symbol table contained the information regarding if a literal was an integer, string, variable, or function name. All of these entries had fields for the line that the literal was found on, as well as its type along with other information that was relevant to the literal(the function’s start line, etc). The first pass of the symbol table, however, had most of these fields zeroed out as it only looked to see if something was a string, otherwise it would mark it as an integer to start. This is corrected during my semantic stage, and I will discuss it in more detail during that part of the paper.

Another major hurdle in the lexical analysis stage was dealing with any space or notes that should be ignored, as they had no relevance to the program. The two kind of these objects I dealt with in InSpace were whitespace and comments. For white space, I started processing it by replacing it with the token WHITESPACE. Very creative I know, but it was only used temporarily and would be cleared out at the end of the lexical analysis stage. For comments on the other hand, I simply had the lexical analysis engine ignore anything once it found the word that signal the start of the comment. In the language InSpace’s case, this was simply the word comment. When the engine found this, it would ignore everything after it, until the engine found the end comment symbol which was the symbol “::”. This means that if the end is not found the rest of the program is ignored, but because other programs would treat the issue the same way, I kept that functionality in there, even if it is not intended by the user.

The only other major hurdle with this process was to make edits to the language itself in order to make sure the lexical analysis engine could clearly differentiate things. For instance, I made it so all strings required quotation marks around them similar to what is found in Java. This way there was a clear differentiation between what could be considered a string and what could be considered a variable or function name. This also helped me later, when I was concerned with the language being unambiguous, which thankfully InSpace is extremely . I also took the semicolon, representing the end of the line in Java, and added it to InSpace. Looking back, it probably wasn’t entirely necessary, but it improved the clarity of the language, so I kept it in. Other then these two changes, the only changes I made were adding the occasional pair of parentheses and eliminating functions like “vacuum”. (puts the program into an infinite loop) These kind of functions, that I considered to be “fun” functions, weren’t really relevant to building a functional language and could be accomplished through other means, so I removed them.

With all the changes and the hurdles planned out, I implemented the lexical analysis portion by first using regular expression to break each line of the programming language into semi-tokens. These semi-tokens weren’t quite lexemes, which are what the tokens produced in this stage are called, but rather broken up parts of a line of words that represented the language, in the context InSpace, each part of the function being called on the line. While I am not the biggest fan of regular expression’s, and neither are most people I talk to, they are extremely helpful in this stage and simplified the breaking up of the function very nicely.

After I handled this, I first replaced any whitespace with the token I came up with to represent it (WHITESPACE), and then stored each line of semi-tokens into an ArrayList. This now “cleaned” ArrayList was parsed through using a switch statement, which found words that matched with the definition of a particular token and added that token to the ArrayList for that held all the lexemes. As part of this process, the symbol table was also populated, and a token called ID was added to the list of lexemes with a dash and a number. After the dash, there was a number that represented the symbol table key for the particular literal, so the information about the particular literal could be reached when needed.

When all this was complete, I was left with a list of lists with each list containing each line of tokens representing the function, however there were some challenges that I faced during this stage. One of the challenges of this stage was making sure there were no duplicates in the symbol table. For instance, when a variable is called multiple time throughout the program, it should only be represented in the symbol table once. I was able to solve this issue by simple making sure the “name” of the literal did not match any other literal in the symbol table before adding it. If it did, the ID of that literal was simply added to the ArrayList, instead of a whole new ID for the duplicate literal.

With the symbol table and ArrayList of lexemes complete, I was then able to move on to the next step of the compiler which was syntactical analysis. This stage is part of what is called the parser, which is simply responsible for making sure the grammar of the language makes sense, and that valid grammar is used in the correct context. In the syntactical analysis phase, the parser is first checking to make sure the grammar for the language is correctly written. For potentially ambiguous languages like Java and C(languages where two statements could mean the same thing), the first step of this stage would be to take the lexemes and put them into an Abstract Syntax Tree(AST) to eliminate the ambiguity of the language. For InSpace, however, I chose not to implement this step (Aho 2014 & Hilfinger 2009).

I am sure someone has just lost it, since I said did not use an AST and parser in the same sentence, so it is only right that I explain why. Like mentioned above AST’s are extremely useful for languages where the program can be written in many different ways. For instance, declaring a variable in Java ,where the variable could be a int, float, double, boolean, etc. Because there are so many different forms to a variable declaration in Java, AST’s are a very useful tool to use. It makes it so the compiler doesn’t have to check hundreds of different combinations to make sure the one it has found is valid with the grammar, however, with InSpace, this issue does not exist.

For InSpace’s case, the language is strictly unambiguous, meaning that something can only be taken one way. InSpace accomplishes this by just being represented and written through a bunch of predefined functions. Due to the functions also being so few in numbers, I found that building AST’s for the parser was overkill. I actually started building them only to realize that it was simpler to write out the ten different functions, or ways functions were allowed to be defined, rather than deal with AST’s and then have to break them down later. I would, however, like to note that I was only able to do this because InSpace was so unambiguous, and also because it had so few valid combinations for its function. This is typically very rare in languages, and I found myself extremely lucky to be able to utilize the “brute force” way of parsing with minimal work and runtime loss. It is for all these reasons, that for the parsing stage (syntactical and semantic analysis) I instead kept the ArrayList of lexemes the same and avoided AST’s.

Now that I have made my justification for not using AST’s in my project (don’t worry I still had to go through assembly), how I implemented the syntactical analysis phase was by simply making up the ten different rules or valid function declarations and checking each stream of lexemes to see if it matched correctly. There were a few ways I made this easier on myself. Firstly, I knew that each line must start with a function call, which was represented by a certain lexeme. I simply popped the lexeme from the list, and used it to find which rule set to check. The other quality of life check I made before actually parsing the rule set, was to make sure the the stream of lexemes, and the rule set matched lengthwise. If they did not, then there was no reason to loop through the rule set because there was already something wrong, and an error would then be thrown.

If the token stream made it through the two quality of life checks, it would then checked to make sure each token matched its respective rule token. Anytime it failed to match, an error would be thrown. Otherwise after the syntactic parser had made it through the each lexeme stream without any errors in them, it would move onto the next stage of the parser which was the semantic check. Before this step, however, the first lexemes of each token stream (the names of functions), which were removed in the syntactical check, were added back to the array so that the semantic check can use them. I actually ran into a few bugs here due to forgetting that Java ArrayList’s are passed by reference not copy and therefore, it took me a bit to realize why my arrays were not changing, even though I had declared another instance of them. It goes to show that even after seven years of Java, I still make mistakes and forget things. After I finally discovered the mistake in my code, I soon moved onto semantics, which ended up being type checking as well.

Before I could move onto the semantic check, as I mentioned above, the symbol table generated in the lexical analysis stage was still in a very basic form and needed to be updated. My solution to this problem was the creation of the type checker. This pre-stage of semantics is actually pretty common, as it makes more sense to implement it here rather than in lexical analysis. This is because up until this point the symbol table was acceptable and could be used without a problem. However, due to now having to check to make sure that the grammar was used in the correct fashion, the type checker needed to update the symbol table.

The main function of the type checker was to go through the symbol table and update the variable names and function names. Up to this point, the only thing that verified to be correctly logged in the symbol were any strings. Otherwise the ID had been logged as an integer, when it might have not necessarily have been one. Because of this, the type checker found all the values in the symbol table that were considered integers, and then tried to convert them into integers. If the conversion failed, the type checker then knew that it had to be a variable or function.

At this point the type checker would first see if the ID represented a valid variable or function name. Just like in Java, the names could not start with a number, otherwise an error will be thrown. Only once the name has been verified to be a valid name, did the type checker parse through each variable or function and determine which type it actually was. The way it determined this was using a similar strategy found in the syntactical check: using the first lexeme. The first lexeme in the lexeme stream, which represents the function that was called, is used to tell whether it is a variable or function. For a variable, it would be the variable declaration function, while for a function it would be its respective function. If it was a function, the functions end would be handle in the main semantic pass rather than in the type checker, otherwise the type checker had finished it job. When the symbol table has been updated with the correct information, it was returned back as this “improved” version and was ready to be used by the semantic parser.

Semantics was the one stage where the word tedious and compiler design really started to show itself a bit. While lexical analysis definitely had some tedious parts to it, when it came to the semantic analysis it really did not compare. The problem with semantics was there was just so many minor things to check. For instance, checking that things had been declared before use, and where it had been declared. It was a simple check to implement. What slowed me down was just the sheer amount of times I had to call it, and the slight tweaks I had to make for every function. There are tools like bison, that can help with semantic parsing but I choose to not utilize them for this particular language. While doing my research, I found bison to be a powerful tool, that was extremely confusing and also built around using AST’s. Both of these issues are what lead to me lean away from using a tool.

So because I chose not to use a tool, all the different checks had to be built manually. The best approach I found to this process was to plan out each check beforehand, sketch out where each check needed to be used, and any particular way that it needed to be implemented for each function. Multiple times, I found that even with all this prep work, I was still having to implement more checks later, or I realized I forgot something and needed to reimplement it. Again, just adding to the tediousness of the project and making it one of the more challenging portions of the compiler.

The most challenging portion of semantics was easily handling the loops of the program. There was just so many checks to run on them, and it was particularly hard when I started making checks with the concept of nested loops in mind. The problem was that most of my checks had been built around using only one loop, but I wanted to get nested loops in the functionality, so I had to build a complicated set of checks to verify that everything was correct. The main challenge was making sure there was not an end before a start or a start without an end. I ended up accomplishing this with counters and checking them once the parsing loop was complete. If the two loop variables didn’t match up, then I knew that something was off and threw an error. I found myself having to do many of these for loops, and I can safely say that they were the most challenging part of the semantic checks.

Nevertheless, I eventually managed to get all the checks implemented. Some of the more challenging checks, along with loops, ended up being functions and variables. Both were very involved by the sheer number of checks, and at points both had checks that crossed over to make sure that one wasn’t declared as the other when called. While I did not have very much fun in this step, it really showed me how complicated semantics can really be, and all the different ways a programmer essentially can mess up a program and crash the compiler. However, when all the checks were complete, the full parser (syntactic and semantic) was finally complete, and the source code was verified to be correct and sensible code. This meant that the code was ready for the next stage, which was intermediate code generation.

For me, intermediate code generation was the one stage I actually found to be overkill. While extremely helpful when there are AST’s or optimization to be done on the code, in my case, since there was none of this, it felt a bit like turning the lexeme streams into intermediate code was just for the sake of having the step. With that said though, at one point I do want to explore optimization in compiler design, which I will discuss later in this paper, and because of this ended up implementing inter code generation. It also gave me good practice with how intermediate code should be structured and built, so while it was not the most helpful stage in my case, I still learned a lot. It also was very useful for catching a few bugs in my code in the previous steps, so it for sure wasn’t a waste of time.

With my commitment to build intercode, I actually skipped ahead and started experimenting with x86-64 assembly before actually writing the inter code generation. My justification for this was that this project was the first time I ever had to deal with actual assembly code. There were a few times in the past that I was introduced to it, and had to write short instructions, but never to the scale I was getting into with compiler design. I will not touch too long on this topic right now, as an entire section of this paper is dedicated to it, but I did want to mention that I would recommend understanding a bit about the assembly that needs to be written before writing intercode. It helped me understand tremendously with what I needed to generate in regards to intercode.

After I had returned from my short adventure in the world of assembly with a better I idea of how it all worked, I got started on building the intercode. The Dragon Book used the term, three name addresses. Since assembly has a lot to do with moving things in out of registers, I took this term and tried to aim for only three word instruction. While it didn’t always happen, it did for the most part, and actually ended up helping make more concise intercode. Therefore, I would recommend taking the statement from the Dragon Book as sound advice, however, this does not make intercode any easier(Aho 2014).

The hardest part I found with it was after working weeks on the parser, I was in the parser mindset, not the intercode mindset. This made me constantly worried about whether the code was correct grammatically, even though it was. My recommendation for this stage is to only be concerned with sizes of characters and numbers like I was in end, as the rest has already been handled. With the sizes of characters and numbers, I just had to make sure things did not exceed the bounds of what the programming language was designed for. For instance, the language does not support negative numbers, so I had to make sure a number did not go less than zero or exceed a certain size (more of an assembly problem), so a negative number didn’t appear. It was in the intercode section that I implemented these checks into the code base. This another reason I recommend quickly exploring assembly to know the limitations of what kind of input will be accepted before generating intercode.

When this step was handled, I was then able to accurately produce the intercode that matched the assembly code that needed to be produced. One of the challenges I faced was that the loops need to be numbered throughout the intercode. For example, if I had two loops, they would need to be called loop1 and loop2 respectively. The end of the loop also needed to be represented this way, so just like in semantics I had to implement a variety of counters to make sure the intercode was generated correctly. Once this was accomplished, I was finally able to move on from intercode and to the final step of the compiler which is code generation.

Before I discuss the prospect of code generation and my approach that I took, I did want to briefly discuss optimization because in modern compiler design this is the optional step that comes after intermediate code generation. While I did not personally go through this step for the reason I mentioned above, it still is used fairly often in modern compiler design, so I wanted to quickly talk about the topics, and some of the types of optimizations that are used.

Optimization is the stage where the goal is to increase the runtime of the code, and eliminate any code that isn't necessary, like unreachable code. While it is not a major concern with relatively small programs, as the program grows in size it becomes a much bigger issue that needs to be addressed, hence the optimization. Things like loops and redundant code can all be optimized to run more efficiently through concepts like dynamic programming, things like greedy vs cheap operations, segments of code, etc. This also plays a part when there is parallel computing involved with the program, and making sure it’s running as efficiently as possible. I will not go into much more details in this particular paper, but there are numerous resources online discussing the topic (Aho 2014 & Hilfinger 2009).

Because I opted out of optimizing the code, the next natural and final step of the compiler was to generate the assembly code. This stage is what is considered the backend of the compiler as it handles things that the programmer is not usually responsible for. This stage is more translating the code into a workable form for the computer. I choose x86-64 as my assembly language that I was going to use for my code generation, as it was the most widely used by the target operating systems that I wanted to use. However, this did not mean this step was easy.

This step was in fact probably the hardest part of the project for two reasons: it was the most unfamiliar aspect of computer science to me and the complexity of assembly language and memory allocation. The biggest problem I found when I got down to assembly language was that it will not help the programmer out in the background like languages like Java and Python. Even in C, where someone has to do some memory management, it still doesn’t compare to the low level that assembly language has the programmer doing. It was by far one of the more challenging aspects of computer science I have ever done, and the best place to start with it is the assembly code itself.

Assembly is one of those things in computer science that no one likes to use or code with, or the person is crazy. It is particularly not fun because it doesn’t hold your hand with anything and instead the programmer is responsible for everything that needs to be accomplished with it. The other fun surprise I found with assembly was every single operating system likes to treat it differently than all the others. For instance, Windows, Macintosh OS, and Linux all accept x86-64 but in vastly different forms. Windows was so different then all the other forms that it was the first operating system to fall out of the scope of the compiler. It didn’t play well with any functions and everything that should take only a few commands in x86, in Windows took almost double or triple the commands.

The next operating system to fall out of the scope was Macintosh OS, and again it had to do with how complicated it was to do something in it. Mac OS compared to Windows was significantly easier, but still had its drawbacks. One of the biggest drawbacks was lack of resources on the topic. While there were a few videos here and there on the subject, it was very sparse with most sites recommending that I avoid both Mac OS and Windows and just settle with Linux. After many failed attempts at finding any useful material on the subject and spending two hours trying to figure it out for myself, I gave up on Mac OS, and, as pretty much everyone on the internet recommended, went with Linux as the OS of choice for my compiler.

With the OS chosen and the documentation on how Linux wanted to accept and see x86, I was able to begin on handling assembly. The timeline for this was a full day of exploring assembly, with two full days of implementing it for my compiler. The first day or exploration day, was the day I mentioned above that I took before the intercode generation step of the compiler design. I went through numerous examples along with numerous amounts of documentation to try and understand what I needed to build for each of the functions for InSpace.

During all these examples and testing, was when I decided upon using NASM, or the Netwide Assembler, for my compiler along with the linker and loader, clang. The reason I went with clang for my linker and loader was in order to have access to the printf method that is used in C. The Linux operating system does have a system function to print out data to the screen, however, it is very archaic and requires many different functions to be built in order to have the functionality to print data like integers. For this reason, I just simply opted to go with clang as my linker and loader instead of the generic version of the linker and loader that comes with Linux.

With the assembler and linker and loader figured out, I could final begin exploring assembly code. It was during this exploration that I learned why computer scientists abhore assembly code: it is evil. This is an exaggeration of course for assembly isn’t evil but rather simply listening to an inexperienced programmer. This results in things like infinite loops and crashes of the terminal. One of the more memorable moments of this stage was the discovery of why segmentation fault jokes are made so much. At long last, after having to deal with them for a solid 2-3 hours and wanting to throw my computer across the room for no less than 3 times, I finally understood the pain they cause programmers.

The best part of the segmentation faults was that the issue was actually pretty simple: a mix-up of caller and callee registers. The problem spanned from using a caller register as my loop counter, which was also used with the printf function. However, by nature, caller registers do not have to be restored to their original assigned values. This resulted in the loop counter, when used by another function, to be left filled with garbage data, hence leaving me with an infinite loop. It was when I fixed the issue, that I truly understood that assembly does exactly what the programmer wants whether it is correct or not and that I needed to be careful when using it.

With my almost required segmentation fault saga now over, I got down to the guts of what makes assembly language on Linux so powerful: the .bss section. Assembly, similar to C++ in some ways, has different section where different data is held. The three sections I used were the .data section, .bss section, and the .text section. The .data and .bss is where all the constants and variable work is written and handled. My .data section was pretty sparse, as InSpace really doesn’t have any constant or final variable functionality, so it mostly just contains the different formats needed by the printf method. These formats are used in conjunction with the printf method to allow for the various outputs I needed, and are selected by what function is called (NASM and IA-32 Information 2015).

Next after the .data section, comes the .bss section, that I found so useful when writing the blocks of assembly code I needed for my compiler. The .bss section, in essence, is where you can predefine any undefined variables that are known to be used throughout the program. The big plus to this section is that it eliminates some of the memory allocation struggles that come with assembly. For declaring local variables, space on the stack has to be created by moving the stack pointer up a certain amount. Then when the variables have served their use, the stack has to be restored by resetting the stack pointer to where it was beforehand. This is typically accomplished through the push and pop that is common with any stack data type, however, since with InSpace all variables are considered global, I didn’t have to worry about this step and instead utilized the .bss section (NASM and IA-32 Information 2015).

Besides the two data sections, come the final section I utilized which was the .text section, which is where the bulk amount of assembly code comes into play. This section is where all the commands like add, sub, mov, etc. are used. Just like a typical program, the .text section has a main section where the work outside of functions are performed, however, x86 does allow for functions as well. Functions in my case were defined prior to the main section, and look similar to any other programming language. The only main difference to a programming language structure comes in terms of loops, where different jumps have to be implemented and point to the correct loop.

However, with all the parts of assembly figured out, the implementation actually went pretty smooth. My implementation of assembly used a bunch of different ArrayLists to hold the information to be stored into each of the sections, that I outlined in the last paragraph. There was, of course, the translating of inter code to the assembly, but after having accomplished this type of translation numerous times throughout the compiler construction process, this went pretty smoothly. The final part was just to add the “extra” things to the assembly document, like the external call to the printf method and defining the global method. When all this was finished, it just became a matter of assembling and linking and loading.

In a previous paragraph, I mentioned that I chose NASM as my assembler, which I though actually worked out pretty well. The nice part about NASM is that it is open source, so it is easily accessible to everyone. It also is simple to use and breaks down into just using one command that builds the output file that is needed for the linker and loader. The linker and loader (clang in this case) then takes the output file and loads everything into the operating system. Once this is done, I just have to call one final command: the now constructed executable. The executable then produces the output that should appear via the code written, and, man, is it satisfying.

Probably the most exciting and rewarding part of this whole project was finally finishing code generation and seeing the output appear on the screen. I can think of very few things that compare to the feeling I felt rush over me when I saw the output for the first time. Yes, there was plenty of bug testing to be done after, but seeing a simple string printed and knowing how much work went into actually producing, it is extremely rewarding. It pretty much made the project worth it, and really that was what it was all about.

Whether the project was worth it or not can really be broken down into two different categories: was it fun and was it rewarding. I have already briefly touched on the rewarding aspect of it, and can firmly say that it was. As for fun, it is a little harder to answer. The problem with compiler design is that it requires the use of things like regular expression, lots of tedious work, and assembly. All of these topics, while they may be fun for some people, were not very fun to me. There were numerous times I had to either take a break from the sheer amount of tediousness, or because of the fact that I wanted to throw my computer across the room multiple times. Yet, I still did have fun completing each step because I knew I was getting closer to my final goal of finishing the compiler. Because of this, I did some reflection on the project itself once it was finished.

I first reflected on what went well with the project. I firmly believe that actually getting the compiler to work in the end is probably the highest on this list. This is because accomplishing the goal should be something on that list, however, there are other aspects that went well. For instance, I thought assembly, outside of the segmentation faults, infinite loops, and many hours of trying to figure out what I was doing, went pretty smoothly. I had mapped out beforehand how long I thought it would take to produce assembly. I am happy to say it took much less time than I originally thought. This may be from choosing the Linux version of x86 which had a crazy amount of resources and examples, but also from realizing that assembly is not the impossible language that people make it out to be.

Another aspect of the project I thought went very smoothly was the syntactic and semantic parser. I was worried that I would overlook many of the checks that needed to be performed, as well as intimidated by the concept of abstract syntax tree. Thankfully, through InSpace not requiring AST’s and lots of prior planning, the whole process actually went pretty smoothly. I, of course, missed a few tests but not as many as I thought I would, which made the later stages move much quicker because I wasn’t backtracking as much. Yet, not everything went smooth in the project and there were definitely things I could have improved on.

The first of these to come to mind has to be the lexical analyzer, which unlike the syntactic and semantic parser, I found myself constantly going back and either fixing or improving. I partly think it was because it was the very first stage of the compiler I ever built, so I really did not have a clear idea of what it needed to do, that I had in the later of stages of the compiler. Also, due to it being the first stage, I found lots of bugs with it down the line, which again added to time I spent in other stages. If I ever did this again, I would probably spend more time planning out the lexical analysis stage than I did with this compiler.

Another part of the compiler that I think I could improve on, would be the character limits and not allowing negative numbers. With assembly, it is possible to increase the character limits and allow for negative numbers, but the assembly code does become more complicated. I found myself spending too much time to figure it out, so I had to pass on trying to increase the limits. It still bothers me, however, and in the future I want to take another look at the assembly code, and see if I cannot get bigger limits to be allowed. However, this experience did teach me the importance of time management with projects like these, and I came away with a lot better idea on how to approach projects this size in the future.

Speaking of the future leads me to talking about where I want to take this compiler from here. For now, I am going to take a break from it for awhile, as a programmer can only take so much assembly in a short period of time. With that said, I still do want to return to it one day, as there are other features I would like to add to it. For instance, I want to add the support for other operating system. In a world with virtual machines, it is not a major concern that it runs on only Linux, however, it still bothers me that it does not work on other OS’s, and is something I am will look into in the future.

Another thing I want to add to my compiler is allowing for greater functionality with InSpace. Operations like multiplication and division are not currently supported with InSpace or the compiler. I may want to take a look at this in the future to provide greater flexibility with the language, and I also want to look at adding support for more of InSpace’s functions. While the functions that I chose to leave out were the “fun” functions I talked about earlier, I still want to add support for them in the future, which I believe is a reasonable goal.

Beyond any future additions to the compiler I built, I don’t see myself starting a new compiler from scratch, and have firmly realized after this project that it is not something I see myself doing as a career. When I first set out on this project, it was to gain a better understanding of how a language was made into something that could be processed, and I firmly believe that I accomplished this goal. Through it all, I feel like I learned a tremendous amount about compiler design and assembly language that I did not have before. To say it was nothing less than a journey is an understatement, but it is a journey that I am glad I went on.

To wrap up my senior project, there was once a time I wanted to try and design an ANSI compliant C compiler. Being the young computer scientist that I was back then, I had no idea what I was actually trying to attempt. After going through each step of compiler design and having to deal with all the issues and bugs that arose with each step, I think I finally understand why my professor looked at me crazy when I told him what I was trying to accomplish. In light of this and to set the record straight, I now have the adequate knowledge to say that I was indeed crazy, and that I will not be making an ANSI compliant C compiler. As one of my friends once said, it was the best thing ever, that I never want to do again.

Sources:

Aho, Alfred Vaino, et al. *Compilers: Principles, Techniques and Tools*. Pearson Education Ltd., 2014.

Hilfinger, Paul N. “CS164: Programming Languages and Compilers.” *CS164: Programming Languages and Compilers, Spring 2009*, UC Berkeley, 19 Jan. 2009, inst.eecs.berkeley.edu/~cs164/sp09/.

“NASM and IA-32 Information.” *Sample 64-Bit Nasm Programs*, UMBC, 27 Jan. 2015, www.csee.umbc.edu/portal/help/nasm/sample\_64.shtml.

Zayne. “InSpace.” *InSpace - Esolang*, 16 Aug. 2017, esolangs.org/wiki/InSpace.