Compiler Construction Final Report

Brindyn Schultz, 12/10/2022

An Overview of the Compilation Process

A compiler is a program that translates code written in one programming language, the source language, into another programming language, the target language. This process of translating code is known as compilation. The compiled code can then be run to execute the source program on the desired machine as many times as desired. Compilers are a vital part of software development, as they enable programmers to write code in high-level languages that is more easily understood by humans, and then easily translate that code into a low-level language that can be understood and executed by a computer. Before a compiler can translate any source code, it must be guaranteed that the source code has no syntax errors. To check for syntax errors, another tool called a parser must be used.

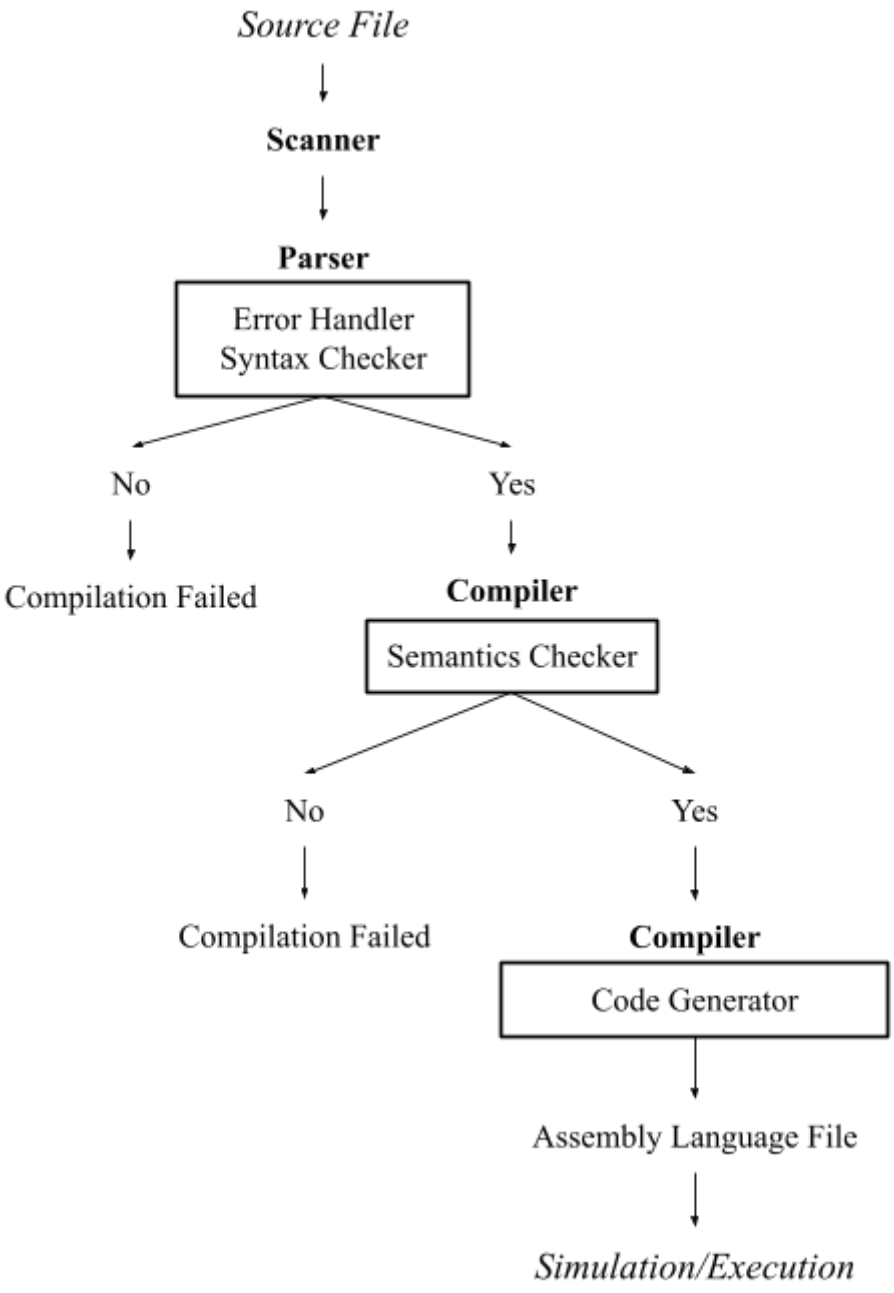
A parser is a program that takes source code as input and produces a syntax tree as output. The syntax tree represents the syntactic structure of the code and how the code's elements are related to each other. When the parser checks the source code for syntax errors, it must ensure that it follows all of the syntactical rules of the source programming language. With a parser implemented, a compiler no longer has to take code as an input directly, but it can take the syntax tree

produced from the parser instead. The compiler only needs to know how the code's elements are related to each other to be able to generate executable code in the target language that functions identically to the high-level source code.

There still exists one issue with the compiling process; how can the parser actually read and interpret the source code? To solve this problem, a lexical analyzer, also known as a scanner, must be implemented. A lexical analyzer, is a program that takes characters as input and divides that input into a sequence of tokens. These tokens represent the basic building blocks of the language, which include keywords, identifiers, punctuation, and operators that are explicitly outlined by the source language. This sequence, or stream, of tokens, can then be used by the parser, since the parser does not need to know how the code is written linguistically, but rather, it must know how the code is written syntactically.

With these three parts, compilation can be done. With this information, the full description of the simplified compilation process is the source file gets read by the scanner which sends tokens to the parser, and the parser checks the syntax and does error handling, and if the program is syntactically correct, the parser sends a syntax tree to the compiler, which checks the semantics of the syntax tree, and if the syntax tree is semantically correct, the compiler generates code from the syntax tree for the target language, which gets saved as an executable assembly language

file, which is then simulated or executed. This can be better visualized in a graphical format.



Lexical Analysis and the Fundamental Building Blocks of Programming Languages

As stated previously, the purpose of a scanner is to read streams of characters and interpret them as tokens to send to the parser to be evaluated syntactically. Before a scanner can do this, tokens must be defined. Tokens represent the building blocks of a programming language and include identifiers, constants, atomic data types, keywords, labels, operators, and punctuation. These building blocks have unique definitions from language to language, but they are typically very similar. The patterns for each of these building blocks are described using regular expressions, which are simply sequences of characters that define specific patterns that something will have.

An identifier is a name given to a unique item such as a variable, function, constant, class, or subroutine. Identifiers are used to explicitly refer to these items in code, so that they can be accessed and used. Some common rules for what can be used as an identifier include certain characters being allowed or disallowed, and limitations in the length of the identifier. Identifiers in the C programming language can be defined with the regular expression [_a-zA-Z][_a-zA-Z0-9]{0,30}, which means that they can start with any alphabetic character or an underscore, followed by any combination of letters and numbers or underscores, but are limited to 31 characters in length.

Constants are used to represent fixed values that are used frequently in a program. They are typically named using all uppercase letters to help differentiate them from other variables. Once a constant has been declared and assigned a value, that value cannot be changed, or an error will be returned by the parser. Constants are important for improving the readability and maintainability of a program. Constants have four different types in the C programming language; integers, octal integers, hexadecimal integers, and character integers. Each of these have their own regular expressions.

Atomic data types are the most basic data types that cannot be divided into smaller components. Atomic data types are typically the simplest data types to work with, and can be used to make more complex structures. Atomic data types in the C programming language are Booleans, integers, characters, and floats. Multiple atomic data types can be used to form things such as arrays and objects.

Keywords are reserved words that have a special meaning in a programming language. Keywords are used to declare variables, create loops, and control the flow of a program. Keywords can never be used as identifiers, because they are reserved for a specific use in the language. Using a keyword as an identifier will result in a syntax error that will get caught by the parser. There are 32 keywords in the C programming language, and some examples of them are ‘int’, ‘for’, ‘while’, and ‘if’.

A label is a name given to a specific point in the code. Labels are typically used with branching statements, such as ‘goto’ in C, to allow the program to jump to a different point in the code. For example, a label could be used to mark the beginning of a loop, so that the program can return to that point while iterating through the loop. Labels use the same regular expression as identifiers, but with the addition of a colon at the end of their name, for example, ‘stop:’ or ‘start:’.

Operators are special symbols or words that are used to perform operations on data. These operations typically include mathematical operators, such as addition and multiplication, as well as more complex operators such as comparison, assignment, and logical operators. Operators are an essential part of any programming language, and are used to manipulate data and control the flow of a program. It is difficult to give an exact number of operators supported by the C programming language, because the language is constantly being updated and reimplemented, which means that new operators are being added and some are removed over time.

Punctuation refers to the special characters and symbols that are used to structure and organize the code. These punctuation marks are an important part of the syntax of a programming language, and are used to define the structure and meaning of the code. Some common examples of punctuation in programming languages include brackets, braces, parentheses, semi-colons, colons, and commas.

These punctuation marks are used to separate statements and expressions, indicate the scope of variables, and more.

The scanner will use many rules made from regular expressions to read the source program and look for each of the distinct building blocks to translate into tokens to send to the parser, which knows how each of the building blocks work syntactically. A summary of these building blocks can be easily visualized in a table.

Identifiers	Constants	Atomic Data Types	Keywords	Labels	Operators	Punctuation
• Name given to pieces of code the program will access.	• Names that represent fixed variables.	• The simplest form of data that can be used.	• Words that represent special reserved functions.	• Name given to a specific point in the code.	• Symbols or words used to perform operations.	• Symbols that define the structure of code.
Ex: Functions and data variables.	Ex: #define PI 3.14	Ex: int timer = 60;	Ex: if (i==0)	Ex: stop:	Ex: y=3+4;	Ex: {},[],””,’’,.

Context-Free Grammar and Parsing

When the parser receives the tokens output by the scanner, they must be evaluated syntactically. This is done using a context-free grammar, abbreviated as CFG, which is a set of symbols and a set of production rules used to generate patterns of sequences composed of underlying symbols. They are called "context-free" because the rules of the grammar do not depend on the context in which the symbols appear, only on the symbols themselves. The symbols used to make a context-free grammar are classified as either being terminal or non-terminal.

Terminal symbols are the basic building blocks of the programming language the CFG defines. They are called ‘terminal’, because they cannot be further split into simpler symbols. These are the symbols that actually appear in the strings of the language read by the scanner. The overall purpose of a CFG is to specify how the non-terminal symbols interact with the terminal symbols in order to assess the validity of the source code.

Non-terminal symbols are the higher-level concepts and structure of a program that are defined by the CFG. They are the intermediate symbols of the grammar that can be expanded into much simpler terminal symbols using the rules of the CFG. This method of expansion is formally known as a derivation. Non-terminal symbols are only concepts, so they are not actually present in the

lines of code from the source program. Instead of representing the building blocks of the language, they may represent the ways that the building blocks interact with each other.

When a CFG is created, the person or group defining the grammar must be very careful with their design to ensure that the rules capture the syntax of the language to the best accuracy possible. If a parser using a CFG passes an invalid string as valid, very bad things can result when the result is passed to the compiler. There are four steps that are normally used when creating a CFG, to ensure the best accuracy. First, you analyze examples of the language to identify common patterns and structures, then you define a set of terminal and non-terminal symbols that can be used to represent the elements of the language, then you define a set of rules that specify how the non-terminal symbols can be combined with terminal symbols, and finally, you test the sample on a set of sample strings to ensure it passes the correct ones and rejects the invalid ones.

With this, a parser can take the strings of tokens generated by the scanner and apply the rules of the CFG to them to produce valid strings which will make up the syntax tree to be used with the compiler. The compiler will then be able to check if the syntax tree made by the parser makes sense semantically, and if so, it will generate the executable assembly language code.

Semantics, Compiling, and Assembly Generation

While the parser can catch syntax errors, it cannot catch semantic errors. That is where the compiler comes in. Semantic errors specifically refer to issues with the meaning or interpretation of the source code. Semantic errors happen when a programmer writes code that is syntactically correct, but incorrectly conveys the intended meaning of the program. In this case, the parser will pass the code as correct to the compiler, but the compiler must be able to detect the code is incorrect, and prevent it from moving on in the compilation process.

A semantics checker is implemented in compilers to handle these semantic errors. When it comes to syntax errors, they are very limited in the number of ways they can occur, however, semantic errors can occur in an unlimited number of ways, so they cannot all be caught. If they are not caught, unexpected results can occur, which are difficult for a programmer to fix. This is why it is useful to address many common semantic errors.

A semantics checker will analyze code to ensure that it uses the correct variables, functions, and more, as well as, it will check to make sure the code is structured correctly to represent the desired logic. An example is when a programmer tries using a variable without declaring it. In this case, there was no issues with syntax, so the parser will pass the code as valid to the compiler, but the semantics checker of the compiler will see that the variable was used but never

declared, so it will return the code as invalid along with an error message to say the variable was undeclared.

If the source code is semantically correct, the compiler can move onto its next function, which is generating the code for the target language. The compiler will first translate the syntax tree from the parser into assembly language instructions with no regard to efficiency or optimization. It will do this using a built in ruleset that will attempt to recreate the program with as much accuracy as possible, however it will likely generate far more instructions than necessary. This is an unwanted effect of compilers, so people who design compilers will want to minimize this. To do this, compilers will use a built-in code optimizer that will condense the instructions from the intermediate assembly code into something that is still fully accurate to the source code, but has more efficiency than the intermediate assembly code.

What Did I Gain From This Research?

Prior to my research I had the idea to make a 64-bit ARMv8 computer inside of Minecraft, which is not only an interesting and unique way to build a computer, but would save me a lot of money compared to building such a machine. On top of building this computer, I also wanted to make a C compiler that would compile C code into ARMv8 assembly. Lastly, I planned on making a program that would allow my Minecraft computer to read the ARMv8 assembly file output by the compiler, then execute it on the in-game computer. I had a lot of knowledge on computer architecture and ARMv8 assembly, but I heavily lacked knowledge on creating compilers. This is what sparked my interest in doing this research on compiler construction. While I knew this project would be ambitious, I quickly found out that it was even more ambitious than I originally thought as I was conducting my research. I learned everything I wanted to know about compilers which took lots of studying and note taking, and a bit of occasional googling, which I believe has well prepared me for this project. Unfortunately, over the course of this semester, I was unable to make much progress with the project, however, I have no intention of abandoning it.

This research has not only helped me gain the knowledge necessary to be able to complete the project, but it has helped me grow as a computer science student. Through learning all of the smallest intricacies and details of how a

computer can transform a high-level program into a machine-readable one, I have gained a much better understanding of both high-level and low-level languages. I have a much better understanding of why a program may return different types of errors, and how it does so. I also believe that this will greatly benefit me in the future whenever I need to debug my programs. Overall, after studying the topics of this research very thoroughly, I highly recommend that any computer science student or computer science professional take that time to study compiler design and get a true understanding of how they work.