=1

1 Background Research - due in 7/12/2020

1.1 Necessary Background Material

1.1.1 What is a compiler?

In short, a compiler is a computer program that converts a source program into machine code that can be executed by a CPU. It does this by taking its input, a program that is written in a human-readable programming language and performing multiple actions upon it. As a result of these actions, an executable program is produced that can be run by the computer.

1.1.2 Why do we use compilers?

Back in the 1950's, programmers wrote software in assembly language. This means that in order to tell the computer to do something, they would have to write instructions that would change the state of specific bits in order to do anything. This meant that even doing simple things takes a lot more lines of code than in higher level programming languages that we have today. For example, this C program;

```
#include <stdio.h>

int main()
{
    printf("Hello, World.");
}
```

becomes much more complicated in assembly language:

```
global _start

section .text

_start:
  mov rax, 1        ; write(
  mov rdi, 1        ;   STDOUT_FILENO,
  mov rsi, msg      ;   "Hello, world!\n",
  mov rdx, msglen   ;   sizeof("Hello, world!\n")
  syscall           ; );
```

```
  mov rax, 60       ; exit(
  mov rdi, 0        ;   EXIT_SUCCESS
  syscall           ; );

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

TODO: reference this website for the code: `https://jameshfisher.com/2018/03/10/linux-assembly-hello-world/`

This meant that programmers were a lot less productive, as they spent a lot of time doing what we now consider trivial operations, as well as having to create solutions to complex problems. As a consequence of this, the price of software exceeded that of the hardware available at the time due to how complicated and time consuming it was to make even a simple program.

Happily, this all changed when John Backus and his team developed FORTRAN. FORTRAN was the first widely used high-level language, and it greatly simplified writing software. It worked by taking input in the form of a simpler language which abstracted away many of the complications caused by writing directly in assembly, and then translated that input into assembly instructions which could then be run on a compatible computer.

This process of translating a high level language into assembly became known as compilation.

1.1.3 structure of compilers

TODO: reference the heck out of the dragon book for this section

The overall structure of compilers has not changed much since the creation of FORTRAN I, and the compiler I will create also sticks to the ideas introduced by it. The structure of a compiler is made up of several stages:

1. Lexical Analysis

   In this first stage, the source code is split into groups of characters which have meaning called lexemes. For example, this:

   ```
   example = 1 + 3
   ```

   Would be split into the following lexemes:

```
example
=
1
+
3
```

Each of these lexemes are then used to create a token. Each token has a value and a type. The variable example is stored in what is called a syntax table at index 1. The equals sign and the addition sign both have no value, but they are the type of an assignment operator and an addition operator respectively. Both of the numbers have the type integer and the value of 1 and 3 respectively. This leaves us with the following tokens:

```
(id, 1)
(assignment, =)
(integer, 1)
(addition, +)
(integer, 3)
```

2. Syntax Analysis

   After the source code has been successfully split into tokens, a syntax tree needs to be produced using the tokens from the previous phase.

1.1.4 what does the examiner need to read or know

To understand the source code aspect of my project, a reader would need to understand basic programming concepts such as what a statement is, how basic logic such as if statements and loop statements work, and be decently familiar with either Java or another mainly object oriented language (for example, C\). They would also need to understand object oriented concepts, such as classes, objects and inheritance. A basic understanding of assembly would also be useful for the later parts of the compiler where we are creating machine code, but I will be documenting these quite intensely and intend to make them as simple as possible.

For a simple definition of what a compiler is, I would reccomend the BBC Bitesize page on assemblers, compilers and interpreters.

`https://www.bbc.co.uk/bitesize/guides/zgmpr82/revision/2`

From there, I would recommend reading the first chapter of Compilers: Principles techniques and tools (AKA The Dragon Book). This chapter

gives an overview of the various components of a compiler and the different transformations that the code that is being compiled needs to undergo before it can be processed by the CPU. An especially useful resource to understand these concepts is figure 1.7, which can be found on page 7. This figure shows how the code to be compiled will look through the various stages of compilation.

Other topics of interest that are located within this chapter are the concepts of tokens, syntax trees and intermediate representation. These are what the source code of this project will be attempting to produce and then use in later parts of the compiler input's journey through the compiler.

1.2 A bit of compiled programming language history

talk about history here

1.3 Related Work

similar products that exist

`https://dl.acm.org/doi/10.1145/611892.611974` very similar project from 2003.

This project sounds like it has a comparable spirit to mine in that it espouses similar ideas regarding how the use of compiler creation tools effect educational benefits, but the above paper discusses a compiler that is designed in order to teach a course, whereas mine is simply a resource from which you can see how a compiler could be implemented without the use of compiler creation tools.

`https://www.researchgate.net/publication/220807902_A_set_of_tools_to_teach_compiler_construction` enhanced tools for compiler creation that are more suited to education from 2008. Based on GNU bison.

This paper introduces a set of tools to aid in the teaching of compilers, as the authors of the paper found that some of the tools commonly used in compiler construction were either obsolete or lacking in terms of educational features. One example of how they remedied this is by making use of a modified GNU bison, which outputs a detailed description of the various states the parser is in whilst parsing the input tokens. This information was lacking in the original bison, making it very difficult to find errors in either the input or the parser code.

My project differs from the tools described in the above paper quite significantly. In the paper, they still make use of tools to create code which skips over the gory details. These tools are better for education, which is an improvement, but I want to stick to just using a programming language in

my project. My intention with this is to reveal how a normal student could create a compiler with out the use of complicated tools and theories, therefore making the student totally understand the process of compilation.

1.4 Professional, Legal, Ethical & Social Issues

???