# The development of DOG

## Part (A) – Introduction

### About DOG

My EPQ project was to create an interpreted programming language, capable of creating common programs often associated with interpreted languages, such as the basic “Hello World” program commonly used as the first program written by a developer when learning a new language with zero experience, and more complex programs that make use of conditionals, user input and variables.

The main specifications that I outlined for myself when I was planning the development of DOG are as follows:

* Can output data onto the screen
* Can store, manipulate, retain and output data in the memory
* Can prompt the user for input
* Can make logical and arithmetical decisions and comparisons both on stored data and data given by the user
* Can perform mathematical operations on both integers and floating-point numbers
* Must gracefully handle errors both in interpreting and at runtime

As well as the following which are not required but would provide additional functionality:

* Basic networking in the form of HTTP GET and POST requests
* Basic file handling such as read, write, and read into memory
* Functions and a more Object-Oriented approach
* Basic set of loops (for, while, foreach)

### What is an interpreted programming language?

This question can be broken down into two main parts. What is a programming language? and What is an interpreter?

#### What is a programming language?

Firstly, a programming language is a series of pre-defines rules and grammar that can be used to produce a result of some kind by either an interpreter or a compiler. There are different levels of programming language, ranging from low to high level. The lower the level, the closer it is to computer readable instructions. The lowest level code is binary instructions which are decoded and executed directly by the processor. Everything higher level than binary has been designed to be easier to read, write and understand by humans.

The next lowest level of programming language is called an Assembly Language and are unique to each CPU architecture. Assembly languages are in the form of three letter mnemonics called opcodes, and values used by the opcode called operands. For example, the assembly instruction MOV EAX EBX consists of the operand, MOV, and two opcodes, EAX and EBX. In most assembly languages, MOV is a mnemonic for MOVE and the command is used to move data from one place to another. In this example, EAX and EBX are registers, very fast short-term memory locations inside the CPU, used for storing data just before it’s executed. This command would move the data from register EAX to register EBX. Even a very simple program such as the infamous “Hello World” becomes long, complex and very hard to read and understand when written in an assembly language, so programmers developed higher level languages.

The lowest level mainstream languages are C and C++, with C being slightly lower level giving more access to memory management. C is commonly used in systems programming, writing software and drivers for electronics, while C++ is mainly used for applications software. DOG is written in C++ as it is fast and because I don’t normally get a chance to develop in C++ due to its complexity and slightly more modern equivalents which are more commonly used.

As you go up to higher level languages, they become easier to read and write, often at the expense of control over the lower level aspects of the program, such as memory management. Java is a language located just above the C family in terms of complexity. Java has an unusual approach of compiling in bytecode and running in the JVM (Java Virtual Machine). This means that Java code will run on any machine or operating system capable of running the JVM, meaning that instead of writing a different version of the language for each operating system, it simply requires porting the JVM so it can run on the device which is significantly less work. There is lots of debate as to whether this is a good idea or not. C# is another language on a similar level to Java.

Moving up from Java and C# you come to languages such as Python, JavaScript (no relation to Java), Perl, Ruby. These are all interpreted scripting languages. Scripting languages are languages that are designed to run a series of commands making up a script, instead of creating a program. They are often used for automation and programs that don’t require a user interface (UI). DOG falls into this category.

#### What is an interpreter?

There are two different ways that programming languages can be converted from high level languages (see above) to low level languages and ultimately assembly languages and binary instructions, compiling and interpreting. Compiling is when a high-level language such as C++ is converted into an assembly language by a compiler. The advantages of compiling are that the program will often run faster than an interpreted program, but compilation can take a long time, depending on the size of the program and the speed of the programmer’s computer.

An interpreted language works by having an interpreter scan the program, executing each line as it gets to it. This is faster to run but large programs can be slower than compiled programs. DOG is an interpreted language as an interpreter is far simpler to write than a compiler and given the time constraints an interpreter is the logical choice, despite my interest in writing a compiled language.

## Part (B) – Research, planning and tools

### Research

The majority of my research into this project was used to make an informed decision on whether to make an interpreted or compiled language, and the basic structure of an interpreter. This is because DOG is a large project and if the initial structure was wrong, there could be serious consequences in the latter stages of the project. In the end, the decision to make an interpreted language as opposed to a compiled language was a fairly straight-forward one, after less than 5 hours of research. I was already leaning towards an interpreted language before I started the project, and all the research backed up my existing viewpoints. An interpreter consists of three main components, the parser, the interpreter itself, and some way of executing instructions, whereas a compiler consists of several more, and requires a lot more planning and design. The research into the structure of an interpreter was vital to the development of the finished program and gave me a great place to start. The rest of my research was during development, checking details on specific C++ functions and basic algorithms and procedures, such as a quick and simple function to check if a file at a given path exists.

### Planning

Most of the planning I did was designing the structure and control flow of the interpreter and planning the kind of grammar for DOG. In the end I settled for a mnemonic style scripting language, similar to assembly languages but easier to read and with a more line by line scripting style to it. The basic structure I came up with was one opcode indicating followed by a series of operands, and optional characters after each statement indicating conditionals or piping returned values to variables. This style, while at first unnatural to an experienced programmer (inline conditionals especially), allowed me to spend time writing new features for the language rather than spending a long time implementing a complicated multiline source file parser. DOG is designed for rapid prototyping rather than large application development in a production environment. Variable piping however might be familiar to Linux users who use the terminal a lot, as piping return values occurs in several Linux commands and some developers utilise it on a day to day basis.

The next stage of my planning was to outline my aims and success criteria, as shown in the above section. These included must-haves that my project would need to include in order to be a useable solution, and a list of additional “nice-to-haves” that would increase usability and functionality but that I do not consider essential to making DOG a useful piece of software. This allowed me to organise my time better as I knew what order to do things in and what things I could save until last to ensure I had enough time to complete my project. An additional and unexpected benefit of organising my time and prioritising this way was that the “nice-to-haves” ended up being much more interesting to make than some of the more mundane tasks such as the initial parsing functions.

After planning the basic structure and how I was going to prioritise my time, I started to write prototypes. The first was a very simple 114-line Python script as a kind of Proof of Concept (PoC). This allowed me to break down the complex task of creating a new programming language into several core functions or units. It also allowed me to start thinking about how DOG as a language would work on a very basic level. It was at this stage where I decided the three primitive data types, although I didn’t implement the BOOLEAN type and integers and floating-point decimals where treated separately (whereas the final version treats all numbers as floating-point decimals under the NUMBER type).

After the simple PoC script, I began work on a more robust prototype. This was to experiment with different methods of parsing a source file, as well as some simple interpreter design. Other than scale, the main difference between the initial PoC script and this prototype was that I built it using my target language: C++. This prototype ended up being developed way more than I first anticipated, and almost ended up being the final build, however the lack of thought into the initial structure ended up causing massive and unforeseen limitations as the project began to grow. This caused me to abandon the build and start planning the final program. The lack of thought into the early structure was caused by me simply trying to build a prototype, which meant I didn’t plan to scale the project like I ended up doing. I learned a lot of lessons from this prototype, mainly about what not to do in the final project!

Shortly after deciding to abandon the above version, I finished the planning phase of the project and started working on the final outcome.

### Tools

The main tool of any programmer is their Integrated Development Environment (IDE). This is a selection of tools and utilities, packaged as a single editor, including but not limited to a debugger, compiler, test support, code editing window, syntax highlighting, find/replace, and countless other utilities. Very few developers code without an IDE. My chosen IDE for this project was Visual Studio 2013 Ultimate, a powerful, if a little heavy C/C++ IDE developed by Microsoft. Normally the “Ultimate” version can cost thousands of pounds for a licence, and is normally only used by large companies, however Microsoft offer free access to their development suite for students, so I was making the most of their powerful tools while I could. However, they do offer a free “Community” edition with limited features. At the time I started writing DOG, I had little interest in any other software for C/C++ development, but after using the JetBrains suite for a couple of personal projects, I would consider using their CLion IDE instead for future projects as it also runs on Linux which I use instead of Windows 10 most of the time.

An IDE is quite a heavyweight piece of software and it can be impractical to use an IDE for some smaller tasks such as quickly changing a gitignore file. For these smaller jobs, a lightweight and fast text-editor is ideal. I used two different text-editors for this project: Sublime text 3 and Visual Studio Code. These both have decent functionally (although Sublime can sometimes be lacking in features) and are ideal for making quick changes due to their fast load times and low system resource usage.

The final tool I used for this project is git. Git is the most commonly used Version Control software. Version Control is a type of program that tracks and stores all the changes and revisions you make to a codebase and allows you to revert back to any previous version or simply view how the project has developed line by line over its existence. Version Control is incredibly important, especially when partnered with a hosting solution such as GitHub, as it allows you to work from multiple locations, never lose any work, work on multiple features in different branches of the codebase, and undo / revert back to any changes made throughout the whole off the projects lifecycle. Version Control allows makes it really easy to write-up the development of a project as it shows you any changes and revisions made.

## Part (C) – Development

### Basic structure:

As mentioned in the above sections, I split DOG into three sections: The parser, the interpreter, and then the actual functionality. The most important of these was the parser. The job of the parser is to read the source file as an input, split it up into each logical line (Note multiple logical lines can be on the same physical line, like sentences in a document), tokenize each logical line, and extract data on each token.

Next, the interpreter iterates over each tokenized line parsed by the parser and uses the tokens plus additional data from the parser to decide the meaning of each line. It then calls the requisite class/function for the line and moves on to the next line.

After each line is interpreted, a function is called to actual “do” what the line is instructing. Theses functions are in a variety of classes, for example, the maths functions such as ADD() or DegToRad() are located in the Maths class. Each function performs a specific operation or procedure.

### The initial prototype:

As explained above, the initial prototype for DOG was a very simple 114-line Python script. The purpose of this script was to help me plan, design and visualise what DOG could end up looking like. The script worked by opening and reading a source file in the current directory, splitting it into lines using a semi-colon as the delimiter and storing them in an array. An array is a data-type used to store multiple values of the same data-type. An example structure of an array similar to the one used here by the PoC script would be:

[“Line one”,

“Line two”,

“Line three”]

Note that the speech marks denote Strings. A String is a collection of characters.

It then uses a simple foreach loop to iterate over all the stored lines in the array. A foreach loop designates a block of code to be ran for each item in a supplied array. For example, a foreach loop on above array would run the code once three times, first on “Line one”, then on “Line two”, and finally on “Line three”.