CMP2089M – Deliverable 1

Compiler Planning

Make a simple compiler for a new language

University of Lincoln

Group 25

AUTHORS	CONTACT DETAILS
RYAN EASTER	16635772@students.lincoln.ac.uk
TAYLOR THREADER	17645110@students.lincoln.ac.uk,
ARRAN BANKS	17639031@students.lincoln.ac.uk
HARRY LANGHAM	15624847@students.lincoln.ac.uk
ALICE JOHNSTON	16635053@students.lincoln.ac.uk

Aim and Objectives:

The aim of this project is to create a basic complier for a new language that will demonstrate the fundamental principles of programming such as: arithmetic operations, loops and assignment operations. Our aim can be broken down into several objectives, as follows.

To achieve our aim the main objective to first understand is the functionality and fundamentals of a proficient compiler. We can define a complier as: '*A program that can read a program in one language, the source language and translate it into an equivalent program in another language, the target language.' (Babu et al, 2015, p.01)* The range of current compilers available varies substantially in complexity, this largely depends on the scope of features available to that language and the target architecture.

The next objective is to define what principles our complier will demonstrate. Given our aim, it will comprise of basic features such as: arithmetic operators, loops, variable declaration and data types. Given that we finish the preceding early we can implement further features.

Our third objective is to define the syntax. In relation to our aim, we must create a very simple language. By conforming to programming standards an int will always represent an integer and so forth. We will also have reservations over keywords to prevent duplication and misuse. Additionally, ceasing the user's ability to create duplicate variable names. We will follow worldwide conventions for arithmetic operators such that addition represents '+', subtraction represents '–', etc.

The next objective is to determine an implementation language for our compiler. We have decided the language to be C based as this is the most suitable route given our groups current skill set.

The final objective is to determine what architecture to build our complier around as each architecture varies in its instruction set. After some research we found that the Intel-X86 architecture is one of the most dominant to date and therefore we decided to use this as our target platform. However, according to Ghuloum *(2015, p.28) '*the compiler we develop is small enough to be easily portable to other architectures, and most of the compiler passes are platform independent.' and due to the nature of our complier this may also apply to us and therefore will allow for interoperability.

We have several development objectives which will begin once planning is complete. These follows:

- Lexical Analysis,
- Syntax Analysis,
- Semantic Analysis,
- Optimisation,
- Code Generation,
- Testing and final optimisation.

Academic Literature:

Compiler Construction - *Waite, W et al. (1996)*
Like 'Compilers: Principles, Techniques & Tools', this literature details the incremental processes to creating a compiler. However, the differences in this source vary given its extensive theoretical information and coding examples. We will use this literature as further reading when directly creating the complier in deliverable two. This will allow us to gain a further understanding and create a more functional model.

An incremental approach to compiler construction - *Ghuloum, A. (2006)*
*Th*e following literature gives a brief overview to the possible techniques and different schemes one can utilize when creating a complier. Many online tutorials have referenced this paper detailing it to be a valid and reputable source for developers with limited knowledge on compiler creation. During the project, this academic literature will be used as a basis for our initial research into creating a complier.

Compilers: Principles, Techniques & Tools. 2nd Edition - *Aho, Alfred et al. (2007)*
Advised to us by our supervisor, 'Compilers: Principles, Techniques & Tools' is well known as an influential book amongst the compiler community with its high validity and reputability. He stated that the given source will provide our group with extensive knowledge on the core components of complier design, development and implementation. From this we have established the foundation of our project which will revolve around the content residing within this source. We will use each chapter as a guide for how we can approach each method to create a complier.

Compiler Construction - *Singh, A et al. (2013)*
The following paper '*Compiler Construction*' provides an adequate insight into the utility behind the software 'Lex and Yacc'. This software provides a service for lexical analysis and high-level parsing. The referenced paper supplies a descriptive overview of their functionality and related processes, as well as providing advantages which will allow us to ultimately conclude on which software, we will utilize to optimize our design implementation. This will allow us to gain an understanding of the software 'Lex' which will be used in the Lexical analysis phase for deliverable two.

Parsing and Compiler design Techniques for Compiler Applications - *Babu, R et al. (2015)*
The following paper demonstrates a generalized overview on the structure and methodology used to construct a compiler. From the preceding we can acquire a brief understanding of Parsing, Semantic Analysis, Code Optimization and so on. These can then be applied to the development of our planning procedures. Given the previous statements we will use this as a precursor to other academic literature listed in this document to attain a generalized overview before beginning to gather a more extensive collection of knowledge.

Project Plan:

A Gantt chart has been created to display each objective mentioned in the ‘Aims and Objectives’ heading. The chart displays our estimated timeframes for the project planning and development process. The chart will be consistently updated throughout the project duration.

In order to precisely allocate estimations for timeframes, we as a group concluded the most efficient route to traverse for the attainment of data was to allocate work packages for each individual group member to:

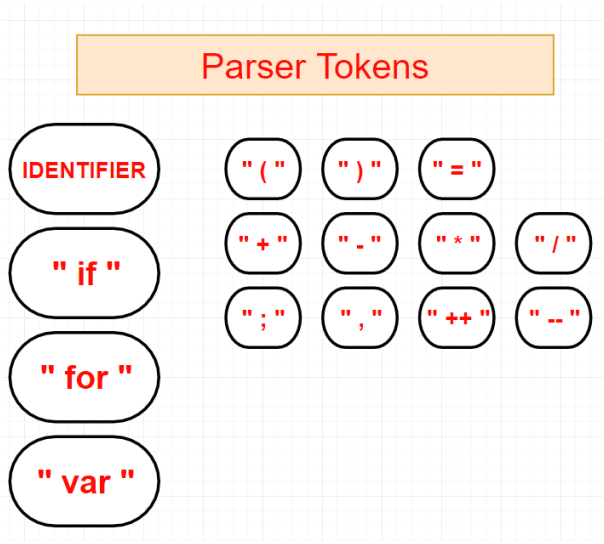
- Create a “Hacky” version of the compiler.
- Attain comprehensive knowledge for the methodology used to create a compiler via reading.
- Extensively research on assembly language.

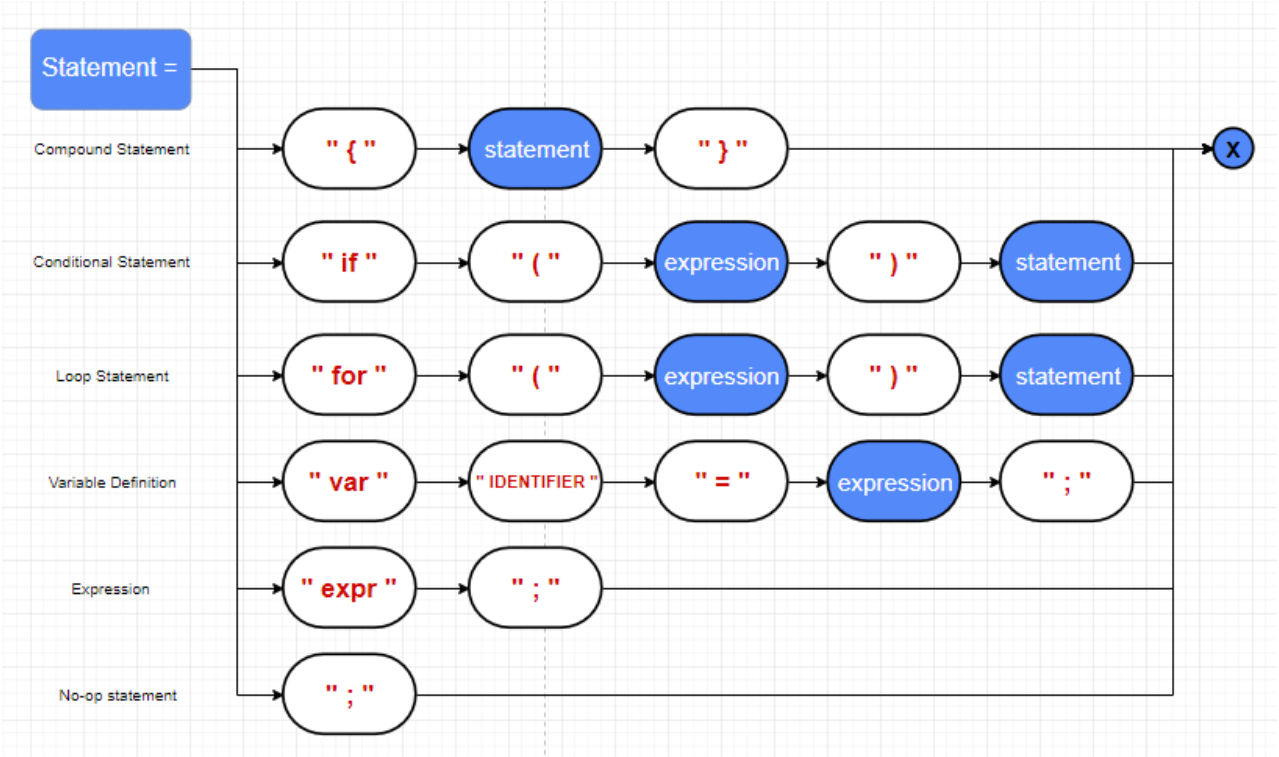
Our reasoning behind this was to ensure we were able to effectively cover all areas within the development process to deduce an approximation for timeframes and a synopsis of our initial state for deliverable two. As a risk management strategy, we decided to implement a ‘hacky’ complier, a functional but less proficient method. This is an alternative work stream to our more elegant and proficient complier that will run in concurrency. Given our time frame this will ensure we have a deliverable to demonstrate. Thus follows,

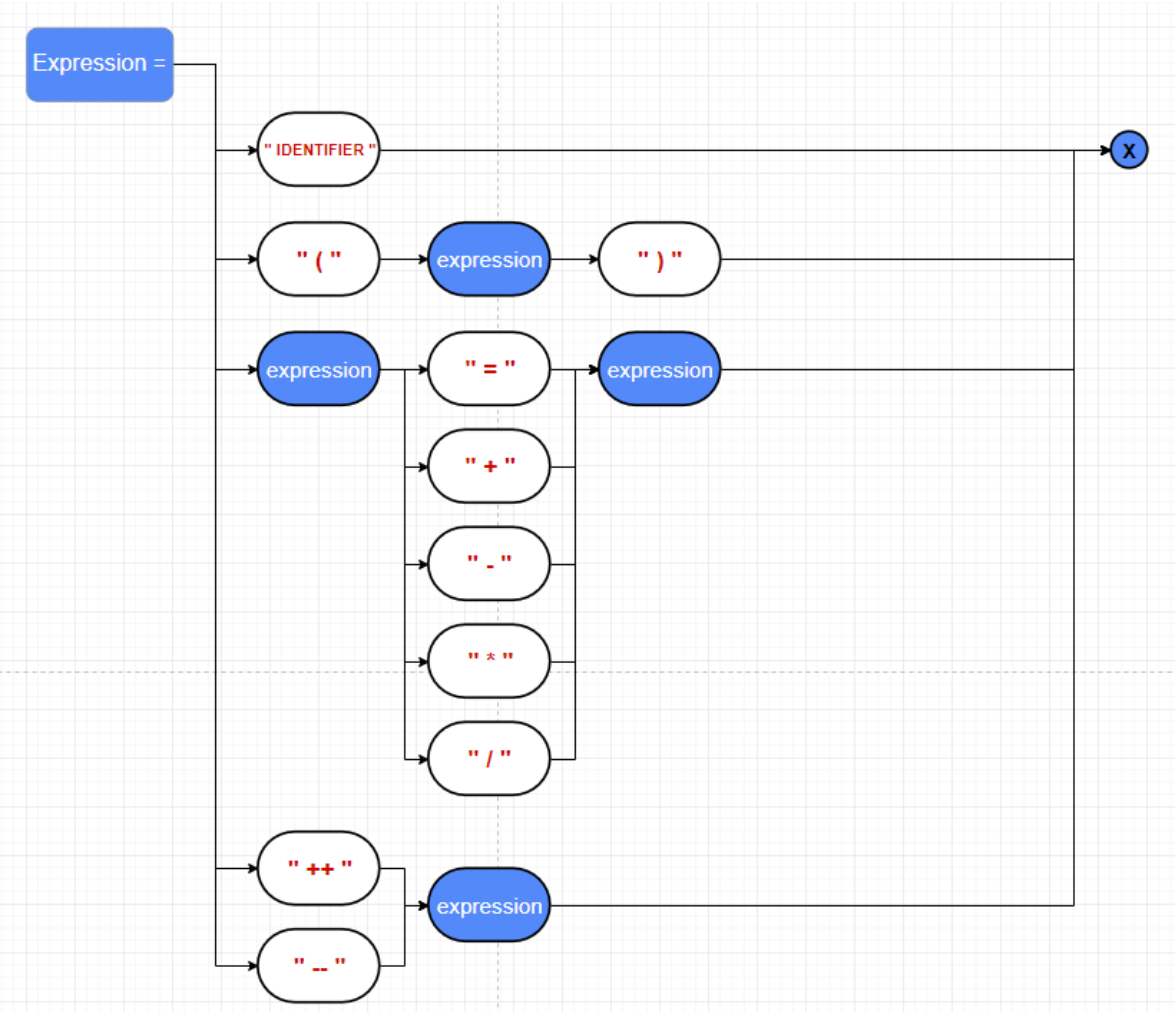
```cpp
//http://cpprocks.com/files/c++11-regex-cheatsheet.pdf
#include <iostream>
#include <string>
#include <regex>
#include <fstream>
using namespace std;
int *func_variable(string code, int *variables)
//takes string, translates letter and number and changes the value at the correct spot in an array
{
    int num; int space;
    space = code[4]-97;
    num = code[8]-48;

    *(variables + space) = num;
    return variables;
}
int *func_arithmetic(string code, int *variables)
{
    int destination_letter = code[0]-97;
    int add_letter = code[4]-97;
    int num = code[8] - 48;

    if (code[6] == '+')
    {
        *(variables + destination_letter) = *(variables + add_letter) + num;
    }
    else if (code[6] == '-')
    {
        *(variables + destination_letter) = *(variables + add_letter) - num;
    }
    else if (code[6] == '*')
    {
        *(variables + destination_letter) = *(variables + add_letter) * num;
    }
    return variables;
}
int *func_for(string code, int * variables)
{
    unsigned first = code.find("for");
    unsigned last = code.find("end_for");
    string for_code = code.substr(first, last - first);
    //for (int i = 0, i = 3, i = i + 1) x = x + 5,

    int destination_letter = for_code[34] - 97;
    int add_letter = for_code[38] - 97;
    int num = for_code[42] - 48;
    int for_num = for_code[20]-48;

    for (int i=0; i<for_num; i++)
    {
        if (for_code[40] == '+')
        {
            *(variables + destination_letter) = *(variables + add_letter) + num;
        }
        else if (for_code[40] == '-')
        {
            *(variables + destination_letter) = *(variables + add_letter) - num;
        }
    }
    return variables;
}
```

```
void regex_search()
{
    int variables[26] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    string code = "int x = 3 for (int i = 0, i = 3, i = i + 1) x = x + 5, end_for";
    regex reg_variable("(int [a-z] = [0-9])"); string variable;
    regex reg_add_subtract("([a-z] = [a-z] [+-] [0-9])"); string add_subtract;
    regex reg_for("for"); string for_loop;
    smatch match;

    if (regex_search(code, match, reg_variable))
    {
        variable = match.str(1);
        func_variable(variable, variables);
    }
    if (regex_search(code, match, reg_add_subtract))
    {
        add_subtract = match.str(1);
        func_arithmetic(add_subtract, variables);
    }
    if (regex_search(code, match, reg_for))
    {
        for_loop = match.str(1);
        func_for(code, variables);
    }


    for (int i = 0; i < 26;i++)
    {
        cout << *(variables + i) << " ";
    }


}

int main()
{
    ofstream myfile ("assembly_code.txt");
    /* ... */
    regex_search();

}
```

Possible language syntax:







We have created logs to ensure our supervisor is up to date with our current progression, attendance and decision taking. We have a document to make note of any questions to which we are continually appending, to obtain feedback and useful information to utilise to complete our project at an optimum and efficient rate.

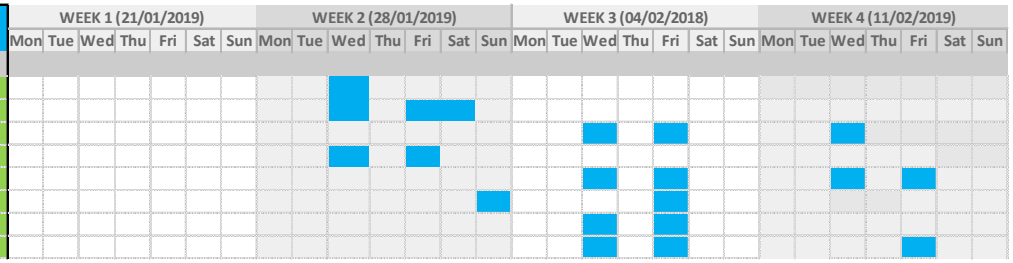
For our version control software, we deduced Git was the best option given its entire history of code changes, ability to switch to older versions of code pushes, local branching, multiple workflows and convenient staging areas.

The planning section for the second deliverable was completed concurrently with the first in preparation for deliverable two. We concluded that our precursor language for this project would be C++ given all our group members familiarity with it, as well as the architecture being x86 given its popularity and mainstream use. Finally, the name of our language we declared is ‘TwoBasic’.

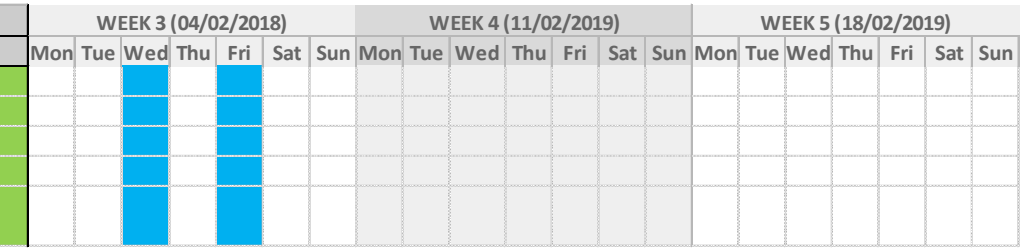
In this section we explicitly clarify on the functionality for each method within deliverable two’s development stage. Lexical Analysis is the initial phase which modifies source code by breaking down syntaxes into a series of tokens by removing whitespace or comments. Syntax Analysis, known as parsing, is the next stage which takes the output from a lexical analyser and applies production rules to detect any errors within the code. The following output is an abstract syntax tree or parse tree. Thirdly is Semantic Analysis which validates the meaning of our previously outputted AST. It helps to interpret symbols, types, and their relations with each other. It usually includes type

checking, scope resolution and array-bound checking. Next is code generation, this process our compiler converts the source code to object code. Lastly is optimisation. This phase will be done after the target code has been generated. It will involve optimally utilising memory hierarchy to increase the speed of the program.

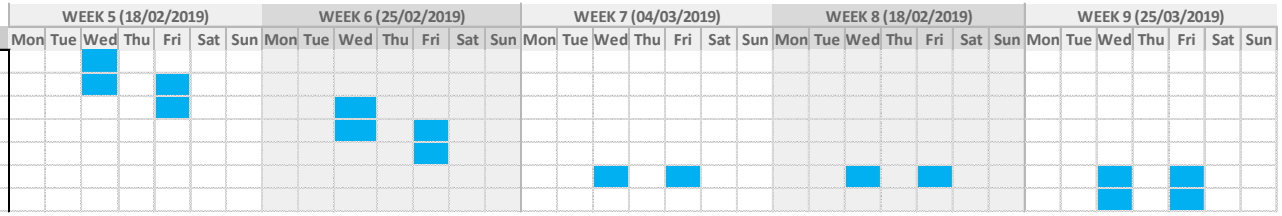
NAME	CODE	TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)	TEAM MEMBER	PERCENT COMPLETE
		Assignment 1					
Arran Banks	AB	Create Attendance & Progress Log (7)	30/1	30/1	1	TT	100%
Alice Johnston	AJ	Establish Aims & Objectives (1)	30/1	2/2	3	AB/TT	100%
Harry Langham	HL	Source Academic Literature (2)	30/1	1/2	3	RE/AB/TT	100%
Ryan Easter	RE	Create Gantt Chart (3)	30/1	1/2	2	TT	100%
Taylor Threader	TT	In Depth Explanation For Project Plan (3)	6/2	15/2	4	TT/AB/HL	100%
All Members	ALL	Create Risk Matrix (4)	1/2	8/2	1	RE	100%
		Fill Contribution Section (5)	6/2	8/2	2	TT	100%
		Create List of References (6)	6/2	15/2	3	RE	100%



Assignment 2					
Planning					
Determine & Setup Version Control Software	6/2	8/2	2	ALL	100%
Define Syntax	6/2	8/2	2	ALL	100%
Determine Principles Our Compiler Will Demonstrate	6/2	8/2	2	ALL	100%
Define Implementation Language	6/2	8/2	2	ALL	100%
Determine An Architecture The Compiler Will Execute On.	6/2	8/2	2	ALL	100%



Development		
Lexical Analysis	2/20	0%
Syntax Analysis	2/20	0%
Semantic Analysis	2/27	0%
Intermediate Code Generation	28/2	0%
Optimisation	29/2	0%
Code Generation	29/02	0%
Final Testing & Optimisation	28/2	0%



Hazard Probability

Probability Score	Details
1	Will almost never occur
2	Occurs rarely
3	An uncommon occurrence, but possible
4	Will most likely occur
5	Is guaranteed to occur or occur frequently

Hazard Impact

Consequence of risk (Impact)	Details
1	The consequence of this risk will be low to no effect. Typical risks involve minor cuts and bruises. Minor damage to property.
2	This is a minor risk, basic first aid may be needed. The risk can be managed with correct controls in place.
3	The consequence of this risk is of medium effect and may occur over time, but the risk is manageable. Examples include trips and falls and high stress levels.
4	The risk is serious, may result in hospitalisation or incapacitation. May also involve data breach or loss.
5	The risk is fatal and may result in death, examples may include electrocution or fire.

Risk Range: (Severity)	
1 – 8: Low Risk	
9-12: Medium Risk	
13-25: High risk	

Risk	Probability	Impact	Severity	Details	Mitigation/Management
Time constraint	3	3	9	The project is incredibly complex and requires a lot of work. With there being so many moving pieces we run the risk of not having the compiler finished before the deadline especially as the deadline isn't that long.	To ensure that we finish the project on time we will need to ensure that we are managing our time effectively and that we are continually talking to each other. We will also be spilt up into two different teams to make two versions of the compiler. One of them will be a quick barebones version that has been “hacked” together. The other version will be a properly built and detailed version.
Incorrect Version Control	2	4	8	With such a large project and so many components needing to be continually updated, version control is crucial. If we do not implement proper version control, the possibility lies that we could lose all our work and be set back by weeks.	Ensuring that we set up a proper version control such as git and making sure that all work is correctly communicated efficiently throughout the group. This will ensure that no work is unnecessarily over written.
Buggy Software	3	3	6	Building a compiler is very complex with many components so it is going to be very susceptible to bugs. If we have too many bugs the compiler won't be very efficient and will regularly break.	To stop the compiler being riddled with bugs we will need to conduct regular and intensive testing to identify bugs and then proceed to fix them.
Getting stuck on a piece of work	4	4	16	This project is very large and very complex. With everyone doing different pieces of the work we run the risk of getting delayed if someone is unable to complete their section on time especially if another member of the group needs said piece of work to continue their own work.	To try and mitigate this issue we are going to develop two versions of the compiler, one being a quick unstructured version, and another professionally built version. The quick unstructured compiler will serve as a filler in case we get so stuck we cannot continue. Another way we will try and mitigate the issue is to temporarily halt development and focus everyone's attention on the

					issue and see if anyone else is able to resolve the issue. Last resort would be going to our supervisor to see if they are able to assist us.
Ensuring front end and back end communicate	3	4	12	The front end of the compiler is based on analysis of the source code and identifying all the components and grammar. This then leads onto the backend which is based on the synthesis stage. In this stage the intermediate code is then converted into the assembly language. One issue we are going to have is linking up the frontend and backend together due to different programming styles and programming methods.	To ensure that we can get the frontend and backend fixed together correctly we will need to ensure that there is regular communication between all people within the group. This is to make sure that everyone understands what is happening and know how it all works. We will also need to make sure all the code is properly commented and made as simple as possible to counter any possible confusion.

Individual's Contribution

All members contributed equally to the preliminary plan by both writing and proving new insights into the sections. Due to this we have assigned a 20% contribution mark to each group member. Please see the table below which details each contribution from the group members.

Name	Student ID	Contribution	Percentage	Signature
Taylor Threader	17645110	Aims & Objectives Academic Literature Project Plan Appendix (a, b, c)	20%	T.Threader
Arran Banks	17639031	Aims & Objectives Academic Literature Reference section Risk Matrix	20%	A. Banks
Alice Johnston	16635053	Preliminary code research Creation of the Lexer Starting the creation of "hacky" compiler	20%	A. Johnston
Ryan Easter	16635772	Preliminary research of assembly language and code generation Risk Assessment References	20%	R. Easter
Harry Langham	15624847	Preliminary research for Aims & Objectives Academic Literature Project plan	20%	H. Langham

References

Aho, Alfred et al. (2007) Compilers: Principles, Techniques & Tools. 2nd ed. Addison-Weasley [PDF] Available at: http://www.informatik.unibremen.de/agbkb/lehre/ccfl/Material/ALSUdragonbook.pdf[Accessed 08 Feb 2019]

Babu, R et al. (2015) Parsing and Compiler design Techniques for Compiler Applications. [PDF] Available at: http://www.ijritcc.org/download/1427436031.pdf [Accessed 08 Feb. 2019].

Ghuloum, A. (2006) An incremental approach to compiler construction. [PDF] Available at: http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf [Accessed 08 Feb. 2019].

Singh, A et al. (2013) Compiler Construction. [PDF] Available at: http://www.ijsrp.org/research-paper-0413/ijsrp-p16108.pdf [Accessed 06 Feb 2019]

Waite, W et al. (1996) Compiler Construction. [PDF] Available at:https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf [Accessed 06 Feb. 2019].

Appendix

Colour Code	Value
	Present
	Absent
	Not scheduled for the session

Date	Time	Taylor Threader	Arran banks	Alice Johnston	Ryan Easter	Harry Langham	Supervisor: Dr Charles
30/01/19	12:00 - 13:00						
01/02/19	11:00 - 13:00						
02/02/19	12:00 – 16:00						
06/02/19	10:00 – 11:00						
06/02/19	12:00 – 13:00						
08/02/19	11:00 - 13:30						
13/02/19	10:00 – 11:00						
14/02/19	12:30 – 15:30						
14/02/19	16:20 – 17.20						
15/02/19	11:00 - 11:30						
15/02/19	11:30 – 13:00						
16/02/19	8:00 – 11:00						
16/02/19	19:00 – 19:30						
20/02/19	10:00 – 11:00						
20/02/19	12:00 – 2:00						

Date	Time	Progress Log
30/01/19	12:00 – 13:20	• Creating attendance and progress log • Started work on Gantt Chart • Discussing what language we want to use • Researching a starting point • Defined base tokens • Created a centralized research document • Broke down Assignment 1 brief in a document • Researching lambda calculus
01/02/19	11:00 – 13:25	• Found out the steps we need to take to build a compiler - Lexical Analysis - Syntax Analysis - Semantic Analysis - Intermediate Code Generation - Optimization - Code Generation • Updated Gantt Chart for Assignment 2 • Researched the functionality of lexical and syntax analysis in relation to our aim. • Preliminary research on how to build a lexical analyzer starting to build a prototype • Found: software used for Lexical Analysis: (Flex) • Found: software used for Syntax Analysis: (GNU Bison) • Found: two academic papers on complier design and programming properties. • Started to update the document plan – discussed as a group the aim of our project and the objectives relating to it. • Started the risk assessment.
02/02/19	12:00 – 16:00 17:00 - 18:00	• Finished Aims and Objectives part for assignment 1 • Update Gantt Chart • Started work on code for Lexer/ Tokenizer
03/02/19	13.00 - 15.00 18:00 - 20:00	• Created risk matrices and detailed levels of risk. • Identified possible risk, with details and mitigations. • Worked more on code for Lexer & completed code
06/02/19	10:00 – 11:00 12:00 - 13:00	• Discussing risks related to our compiler (Risk matrix) • Discussing what to demonstrate to our supervisor for the meeting (8th Feb, 11am) • Finished Gantt Chart
08/02/19	11:00 – 13:30	• First meeting with Charles to discuss our current progress

		• Discussed splitting into 2 teams, 1 to create a 'hacky' a compiler and another team to build a full compiler • Designated specific jobs to each person to help speed up work • Collected 2 books from the library for reference while building • Setup GitHub to host version control and added all members to the project. • Found book: Programming from the Ground up. • Updated the risk assessment to include new risks • Added references for the books we have used
08/02/19	17:00 – 22:00	• Read Syntax Definition section of 'Compilers Principles, Techniques and Tools' and made notes on the section • Implemented what was learnt in reading part previously, these implementations follow: • Created Context free Grammar (Syntax) - Defined Tokens - Defined Grammar Specification - Defined nonterminal symbols - Defined Production rules
10/02/19	16:00 – 18:00	• Looked up the process of intermediate code generation • Defined the assembly language needed and conversion methods+ • Defined process required and instruction sets needed • Updated risk assessment - finished
11/02/19	15:00 -18:00	• Started creating a "hacky" compiler
13/02/19	10:00 – 11:00	• Updated Gantt chart • Created Decisions taken logs (Work and tasks allocation)
14/02/19	12:30 – 15:30 16:20 – 17:20	• Finished Academic Literature (2) • Updated Project plan
15/02/19	11:00 – 13:00	• Met with Supervisor • Continued work on the "hacky" compiler • Determined the architecture we would use • Updated Appendix • Updated Individual contribution form.
16/02/19	8:00 – 11:00 19:00 – 19:30	• Finished writing project plan • Formatting whole document and • Validating word count • Updated Gantt chart • Finishing touches to deliverable one
20/02/19	10:00 – 11:00 12:00 – 14:00	• Discussing our current state with deliverable one • Updated deliverable one summary document

NAME	CODE
Arran Banks	AB
Alice Johnston	AJ
Harry Langham	HL
Ryan Easter	RE
Taylor Threader	TT
All Members	ALL

Date	Time	Decisions taken (Work and task allocations)
30/01/19	12:00 – 13:00	• Gantt Chart: **TT** • Logs Doc: **TT** • Language discussion: **ALL (**Choice: (C++)) • Researching what to do: **ALL** • Breakdown of Assignment 1: **AB**
01/02/19	11:00 – 13:00	• Methodology needed to create compile: **ALL (**research) • Find academic sources for reference: **ALL** • Discussion of the aims and objectives: **ALL** • Start Risk Assessment: **RE**
02/02/19	12:00 – 16:00	• Aims and Objectives (1) Finished: **TT & AB** • Gantt Chart Update: **TT** • Creation on Lexer/Tokenizer (Coding): **AJ**
06/02/19	10:00 – 11:00 12:00 – 1:00	• Discussion around risks: **RE & AB** • Gantt chart finished: **TT** (The content)
08/02/19	11:00 – 1:30 17:00 – 22:00	Designating paired working assignment research roles and partners: **ALL** • Roles: - **TT & HL**: Book Research (Add more references + gather info) - **AB & RE**: Learning assembly language - **AJ**: Creation 'hacky' compiler • Discussion of version control software: **ALL** (Git/GitHub) • Setting up version control software for group members: **R** • Design and development of Context free grammar (Syntax): **TT**
10/02/19	16:00 – 18:00	• Research Intermediate code generation: **RE**

		• Defining the assembly language arithmetic needed and conversion methods: **RE**
13/02/19	10:00 – 11:00	• Gantt Chart Update: **AB** (Minor fix – calendar dates added)
14/02/19	12:30 – 15:00 16:20 - 17:20	• NULL • NULL
15/02/19	12:00 – 14:00	• **TT** – will update the deliverable one document
16/02/19	8:00 – 11:00	• NULL
20/02/19	10:00 – 11:00	• We decide the Gantt chart is too large for deliverable one's summary document: **ALL**
20/02/19	10:00 – 11:00 12:00 – 14:00	• NULL • Decision to correct any mistakes in deliverable one's summary document: **ALL**