Developing a General Purpose Compiler

A dissertation submitted in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours) in Computing

By Daniel McCarthy

Department of Computing & Information Systems

Cardiff School of Management

Cardiff Metropolitan University

April 2017

Declaration

Hereby I declare that this dissertation under the title “Developing a General Purpose Compiler” is entirely my own work and it has never been submitted or currently being submitted for any other degree.

Supervisor: Ana Cauldron

Candidate name: Daniel McCarthy

Candidate signature: ……………………		Date: ………………………..

1 ABSTRACT

This paper explains the theory behind compiler development and linker development. The paper demonstrates and backs up the claims made with a working compiler project for the Craft programming language http://craft-language.org.

The Craft compiler is open source and can be found on GitHub: https://github.com/nibblebits/craft-compiler/

Throughout this paper we will start with a short introduction where you will be briefly introduced to the world of compilers. You will learn what a compiler is and the components that make up a compiler. After the introduction, you will then be at section 6 “A deeper understanding of compiler Internals” and after reading that you should have gained enough knowledge to start researching further into compiler development. After section 6 you will reach section 8 “Craft Compiler” where I have wrote about the difficulties and successes of this project as well as things I wish I did differently and the plans for the future of Craft compiler. You also get to see screenshots of a Snake game written in Craft language. Finally, the paper is concluded with a conclusions section.

2 ACKNOWLEDGEMENTS

I would like to thank Ana Cauldron for her kind support throughout this dissertation project.

I would also like to thank Catherine Tryfona for helping me get onto my computing course with only real world experience to show.

Finally, I would like to thank every other lecturer, friend and family member who has supported me throughout my four years of study at Cardiff Metropolitan University.

3 TABLE OF CONTENTS

4 TABLE OF FIGURES

5 INTRODUCTION

A general purpose compiler is a piece of software that converts computer code written by a computer programmer in a higher level language into a different form typically a lower form such as assembly language or machine code.

Compilers are not projects for inexperienced software developers as they are quite complicated projects and have lots of parts. A compiler may contain some or all the following parts

1. Preprocessor (typically to process macros related to the programming language in question)
2. Lexical Analysis (To turn the source code into tokens to make parsing easier)
3. Parsing (To generate an AST(Abstract Syntax Tree) of the source code to allow for easier semantic analysis and code generation)
4. Semantic Analysis (Used to ensure the tree is valid and appropriate code can be generated from it)
5. Code generation to produce assembly language (Our higher level language is converted to assembly language which represents machine code and is easier to read than machine code)
6. Assembler to produce machine code (Takes our generated assembly language and converts it to something the machine can understand or to an object file for linking in the future)
7. Optimization to remove or replace instructions generated by the assembler that are not required therefore increasing the speed of the program and reducing its size. (Optimization can be done on the AST, Assembly language and possibly but unlikely machine code)
8. Linker to create an executable file or raw binary file from all the object files required to make the executable (Takes all object files possibly generated by an assembler and merges them together as well as resolving any required addresses unknown until link time).

This paper attempts to address and explain in depth how each part of a compiler works.

Before continuing with this paper, it is recommended that you research the Intel 8086 processors instruction set if you are inexperienced with the architecture.

6 A DEEPER UNDERSTANDING OF COMPILER INTERNALS

6.1 PREPROCESSOR

A preprocessor is a program that processes its input data to produce output that can be used as input to another program (Menasce, n.d.).

The preprocessor in the C programming language implements the macro language that is used to transform C programs before they are compiled (Free Software Foundation, n.d.).

6.1.1 The C programming languages Macro Language

The C programming languages macro language is very powerful as it allows you to define object-like macros which are names with values. If those names are then used throughout your program, they are replaced with the value (Free Software Foundation, n.d.). This is all done at compile time.

Take the following code example

#define SIZE 1024

process(SIZE);

After preprocessing the reference to “SIZE” is replaced with 1024.

process(1024);

Object-like macros allow for this ease of readability so that a programmer does not need to remember immediate numbers they can just remember the names associating with them.

Object-like macros in the C programming language can also have expressions assigned to them, they can also reference each other.

Take the following code example

#define WIDTH 1024

#define HEIGHT 512

*#define SIZE WIDTH * HEIGHT*

If somebody was to use the above code and reference to the definition named “SIZE” the preprocessor would replace that value with the value 524288.

This allows for equations that will never change to be interpreted during compile time so that a program does not need to calculate it at run time or have the programmer define 524288 themselves.

The C programming languages macro language also allows the use of function-like macros (Free Software Foundation, n.d.). These function-like macros allow a programmer to define a macro that acts like a function and is interpreted at compile time.

Take the following code example of a function-like macro that multiplies two numbers

```
#define MUL(x, y) x * y
process(MUL(5, 5))
```

After preprocessing the *MUL(5, 5)* is replaced with 25. The code becomes

```
process(25);
```

There are many other ways you can use macros in the C programming language they can be a very powerful tool for software development and are usually necessary for cross compatibility between different systems as they allow you to block out parts of a program by checking if a definition is defined which you can also do with C’s macro language.

6.2 Lexical Analysis

Lexical analysis is where you take a stream of input such as text and you uniquely identify different parts of the input to form what’s known as tokens. A single token describes a part of the input. For example the input “int a = 50;” could be converted to the following tokens.

<keyword, “int>”, <identifier, “a”>, <operator, “=”>, <number, “50”>, <symbol, “;”>

The purpose of lexical analysis is to make the logic of the input a lot easier to understand. By breaking things down this way you can clearly see what is a keyword and what is an operator. This makes the job a lot easier for the parser later on.

Steven S. Muchnick describes lexical analysis in his own words: “*lexical analysis*, which analyses the character string presented to it and divides it up into tokens that are legal members of the vocabulary of the language in which the program is written (and may produce error messages if the character string is not parseable into a string of legal tokens);” (Muchnick, 1997)

6.3 PARSING

A parser takes in a token input stream created by lexical analysis and attempts to build an abstract syntax tree. This abstract syntax tree describes the flow of a program.

Parsers usually use a grammer which is a set of rules that describe how the parser should break down input. The grammer is responsible for guiding the parser so that a valid abstract syntax tree is produced.

Take the following input “a = 50 + 20;” lexical analysis may convert this to the following tokens

<identifier, “a”>, <operator, “=”>, <number, “50”>, <operator, “+”>, <number, “20”> <symbol, “;”>

The parser would then take this input and produce a tree in memory that is graphically represented in Figure 1.

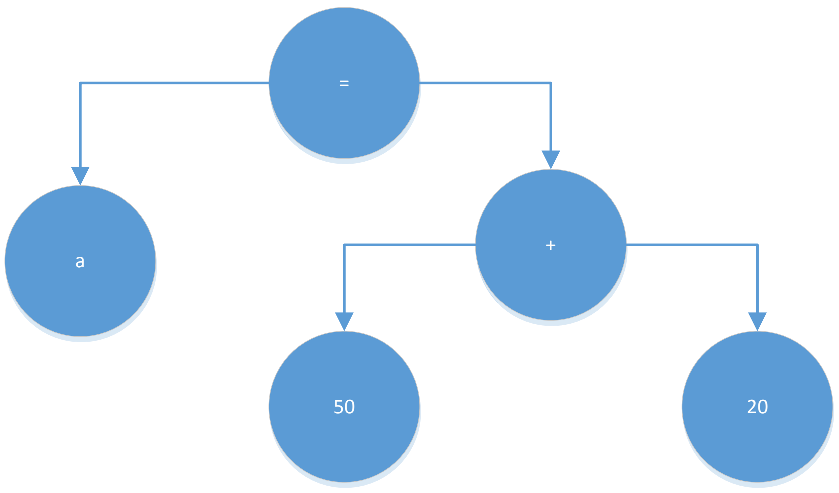


Figure 1 - Abstract syntax tree for "a = 50 + 20;"

The tree in Figure 1 may be different for the given input but it would have a similar concept.

6.3.1 Types of parsers

There are many different ways to parse input, there are LL parsers a.k.a top-down parsers, there are also LR parsers a.k.a bottom-up parsers. Parsers can use a grammer to guide the parser to produce a valid abstract syntax tree or they can be programmed to produce a valid abstract syntax tree without using a grammer.

6.3.2 LR Parsers

The concept of an LR parser was invented by Donald E. Knuth. In his paper "On the Translation of Language from Left to Right" he explains contextually and mathematically the LR parser. Donald E. Knuth stated in his paper "In this paper we single out an important class of languages which will be called *translatable from left to right;* this means if we read the characters of a string from left to right, and look for a given finite number of characters ahead, we are able to parse the given string without ever backing up to consider a previous decision." (Knuth, 1965)

Donald E. Knuth is correct on this, assume the following rules to describe a simple grammer:

1. *The furthest element to the left indicates the rule name*
2. *All elements after the rule name that are separated by a ":" symbol are the required elements or rules for this rule to be considered valid.*

Now assume the following grammer rules:

1. *E:number*
2. *E:E:operator:E*

Assume the following input

5 + 3

This becomes the following tokens after lexical analysis

<number, “5”>, <operator, “+”>, <number, “3”>

At this point the tokens created by lexical analysis will be in a stack which will be called an input stack. We shift one element off the input stack and we notice a “number” token that has a value of “5”; we then look for a rule that matches this, rule 1 seems valid so we create an “E” branch and make the number token a child of it. This is call *reducing* as we are reducing the input based on a grammer rule.

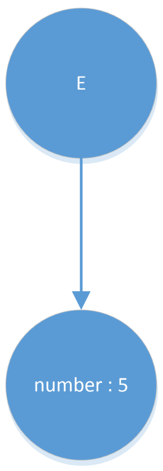


Figure 2 - number "5" reduced to branch "E"

Now that the branch has been reduced the result is pushed onto another stack which we will call the output stack. At this point this is what our stacks currently look like.

Input Stack

<operator, “+”>, <number, “3”>

Output Stack

E

So, one element has currently been reduced but the parse is not complete, we now need to continue shifting the tokens and trying to reduce the results. So we shift the next token from the input stack and we notice the token is an operator whose value is “+”. Since our output stack has an element in it we need to check if a rule exists for this element with the next input token.

So we check if there is a rule for “E:operator”. No rule exists so then we check if there is a rule for just an operator and we find that again no rule exists. The following input cannot be reduced yet so we just shift the token onto the output stack without reducing anything. This is known as a *shift* where no reduction is done. The stacks now look like this

Input Stack

<number, “3”>

Output Stack

E <operator, “+”>

We then shift the final number token off the input stack and we have a number whose value is 3. We check to see if a rule exists for “operator:number” and we notice that no rule exists so we then check if a rule exists for “E:operator:number” and once again no rule exists, so finally we check for a rule for just a number “number” and we have a match, we then reduce a number token into an E branch and push the new branch to the output stack.

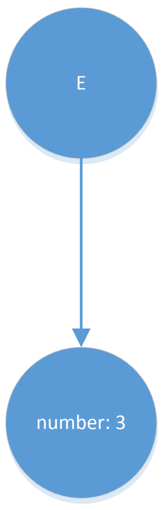


Figure 3 - number 3 reduced to branch E

The stacks now look like this

Input Stack

Empty stack

Output Stack

E <operator, “+”> E

Since our input stack is now empty there is no more input to shift onto the output stack however our output stack still has more than one element in it which means we have not finished reducing. So we check if a rule exists for “operator:E” and we find that no rule exists for this meaning that it is impossible to reduce a “<operator, “+”> E”. So we also look further down the output stack and we look for a rule that requires “E:operator:E” and we find that there is a rule for this so we reduce it and we have finished the parse and end up with the following tree shown in Figure 4.

See next page

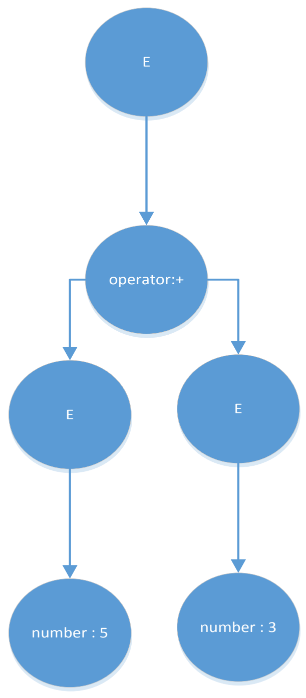
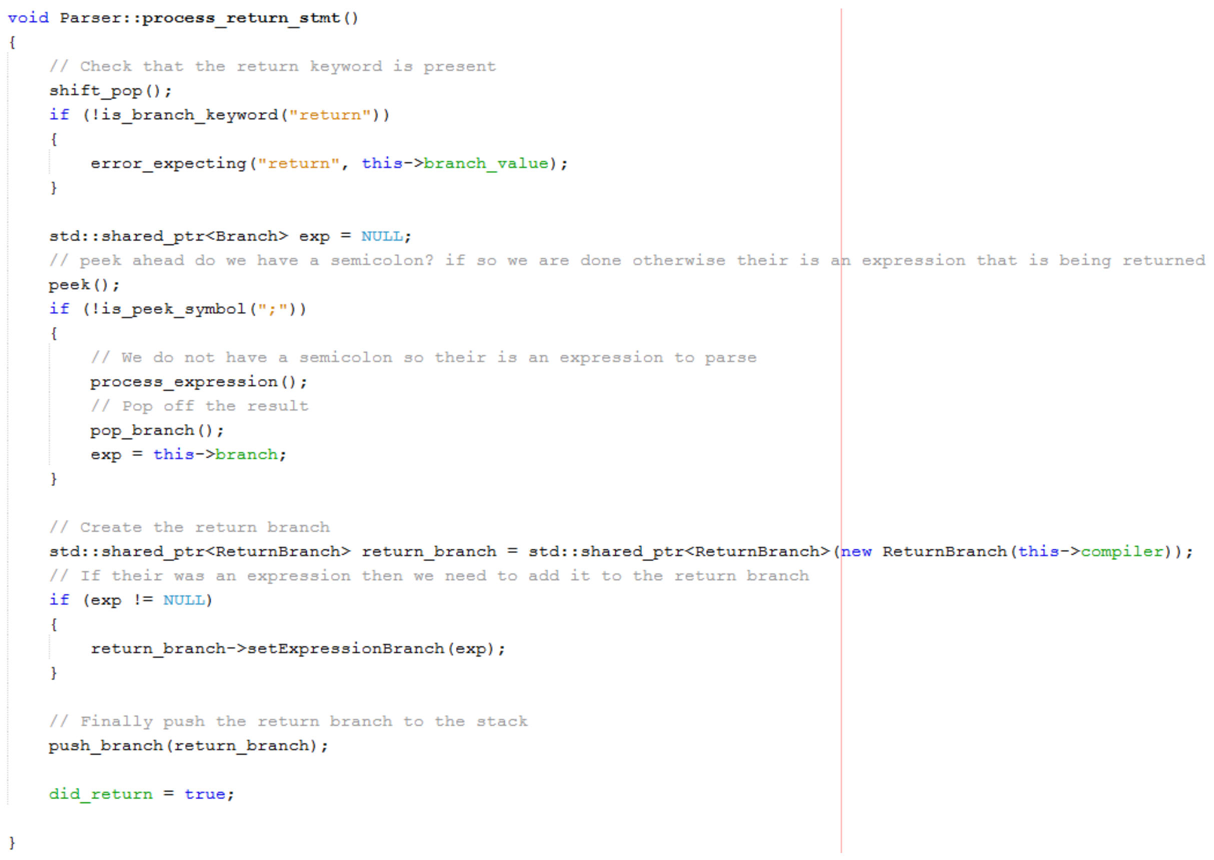


Figure 4 Tree for 5 + 3

Traditionally LR parsers work slightly different than described above but the same concept applies. The design of an LR parser contains two tables an *action table* and a *goto table*. The defined grammer is used to generate a parse table which holds the *action table* and the *goto table*. These tables guide the parser into reducing correctly. The *action table* describes how something should be shifted or reduced and the *goto table* guides where the parser should go next. It is possible for a parse to be accepted or have an error. If there happens to be an error, you call the error recovery routine. (Anon., n.d.)

6.3.3 Parsers without grammer input

You can also have parsers that take no grammer input. This type of parser is programmed to take a stream of tokens produced by lexical analysis and convert them to nodes for the abstract syntax tree and then push them to the output stack. You can see example code in Figure 5 which shows the parsing of return statements in the Craft compiler.



```
void Parser::process_return_stmt()
{
    // Check that the return keyword is present
    shift_pop();
    if (!is_branch_keyword("return"))
    {
        error_expecting("return", this->branch_value);
    }

    std::shared_ptr<Branch> exp = NULL;
    // peek ahead do we have a semicolon? if so we are done otherwise their is an expression that is being returned
    peek();
    if (!is_peek_symbol(";"))
    {
        // We do not have a semicolon so their is an expression to parse
        process_expression();
        // Pop off the result
        pop_branch();
        exp = this->branch;
    }

    // Create the return branch
    std::shared_ptr<ReturnBranch> return_branch = std::shared_ptr<ReturnBranch>(new ReturnBranch(this->compiler));
    // If their was an expression then we need to add it to the return branch
    if (exp != NULL)
    {
        return_branch->setExpressionBranch(exp);
    }

    // Finally push the return branch to the stack
    push_branch(return_branch);

    did_return = true;

}
```

Figure 5 - An image of parser code that parses return statements for the Craft programming language

From Figure 5 we can see that the parser is shifting the next token from the input stream into the output stream and then immediately popping it out of the output stream. The popped token is then checked to make sure it has a token class of “keyword” and that its value is “return”. An error is thrown if it is not a keyword whose value is “return” as this would mean it is not a return statement.

The parser then peeks ahead into the input stream to see if the next token has a class of “symbol” and its value is a semicolon “;” to signify the end of a statement. If the token is not a symbol whose value is that of a semicolon, then this must mean that the return statement has an expression so the expression is then processed and the expression is popped off the output stack and stored in a variable to be dealt with later.

Finally, at this point it is time to make the return statement node as the parser is done processing this return statement. So, the parser creates a return branch and if there was an expression that was processed it attaches it to this return branch. The return branch is then pushed to the output stack. This is a complete parse of a return statement.

For the return statement “return 50;” this would result in the tree node shown in Figure 6.

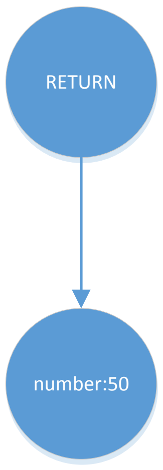
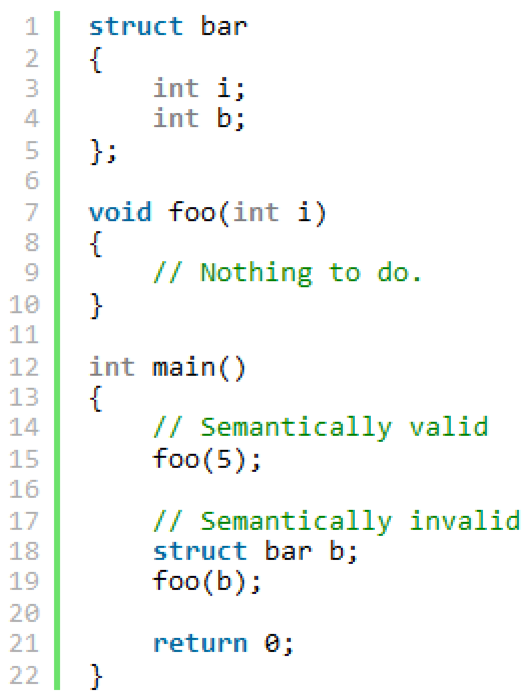


Figure 6 - RETURN node for return statement "return 50;" for the Craft Programming language

6.4 SEMANTIC ANALYSIS

Semantic analysis is where you ensure that declarations and statements of a program are semantically valid by ensuring that data types are used appropriately throughout the program. (Siegfried, 2004). This would be the case and semantically valid if you had a function that accepted one function argument whose type is an integer and someone who calls that function passes an integer. If, however they passed a structure or any other type that cannot be automatically casted into an integer then this would be illegal and the compiler would throw an error.



```
 1  struct bar
 2  {
 3      int i;
 4      int b;
 5  };
 6
 7  void foo(int i)
 8  {
 9      // Nothing to do.
10  }
11
12  int main()
13  {
14      // Semantically valid
15      foo(5);
16
17      // Semantically invalid
18      struct bar b;
19      foo(b);
20
21      return 0;
22  }
```

Figure 7 - Screenshot of semantically valid and invalid code

As you can see from Figure 7 you can see that we have a function whose name is “foo” and accepts one argument of type “int”. You can also see that on line 15 we call function “foo” and pass an immediate value of “5”. Since decimal 5 can represent an integer it is valid for arguments accepting type “int”. So, it is semantically valid.

However, on line 19 we attempt to call “foo” and pass a structure. This is semantically invalid as this structure is not an integer and does not represent an integer in anyway so a compile error would be thrown.

Return types are also semantically validated, if a function returns a pointer variable but its caller is trying to store it in a non-pointer variable this too would result in a semantic error.

6.5 CODE GENERATION

Code generation is what makes a compiler a compiler. Code generators take the AST(Abstract Syntax Tree) and create output, typically assembly language.This section will explain essential information that is required to be known to understand the generation of assembly code. After which the generation of assembly code will be discussed.

6.5.1 What is a stack?

A stack is a container that holds elements that are inserted and removed in a last in first-out(LIFO) fashion. Stacks have two operations the ability to insert an element which is known as pushing an item to the stack and the ability to remove an element which is known as popping an item from the stack. (S.Adamchik, 2009).

With a LIFO stack pushing an element to the stack adds an item to the top of the stack and popping an element from the stack removes an item from the top of the stack. (S.Adamchik, 2009).

Stacks can be very useful in software they have many purposes and without them recursion and function calls would be made difficult.

The stack of a processor is useful for a computer program as it allows a function to have its own set of local variables and access function argument variables the function caller passed to it.

The stack of a processor allows memory to be used on the local scope of a function without relying on using the heap of an operating system. This memory could hold local variables or could hold values that are temporarily stored that a code generator must generate code for as a statement cannot be completed with the registers available as they are in use.

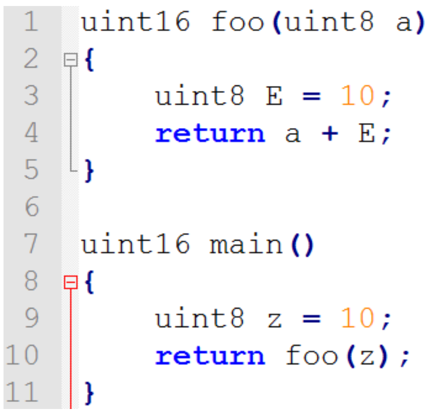
Memory on the scope can be reserved or freed at any time. Variables in other scopes within the function such as an “IF” statement’s scope will only consume that memory while the program is executing inside the “IF” statement’s scope, once the program leaves the “IF” statement’s scope the SP(stack pointer) which points to a particular part of the stack is incremented by the size of the local scope for the “IF” statement allowing the memory inside the “IF” statements scope to be usable once again. This allows the stack to be a fast way to allocate and free memory it is much faster than a heap. This reserving and freeing of memory is not limited to sub-scopes all scopes perform this way and will free their memory when program execution reaches the end of the scope.

In many processors including Intel processors calling functions or subroutines push the value of the current PC(Program Counter) + the size of the call instruction to the stack. The PC(Program Counter) holds the address of the current instruction. The value that was pushed is the address to the next instruction after the function call. The pushing of this address is essential as when returning from a function or subroutine the program needs to know where program execution must continue from.

In the Intel processor, the stack grows downwards while pushing elements.

6.5.2 What is a stack frame?

For Intel processors a stack frame holds the function callers stack frame, the return address that the function called must return to after the function has finished executing, the stored base pointer of the function caller's base pointer, and local variables of the current function. (Cataldo, 2016). A stack frame exists because the code generator generates code that will access and use the stack in such a way and it is not built into the processor its self.



```
uint16 foo(uint8 a)
{
    uint8 E = 10;
    return a + E;
}

uint16 main()
{
    uint8 z = 10;
    return foo(z);
}
```

Figure 8 - Example program written in Craft language

A graphical representation of the stack frame for function "foo" that is shown above in Figure 8 is shown in Figure 9.

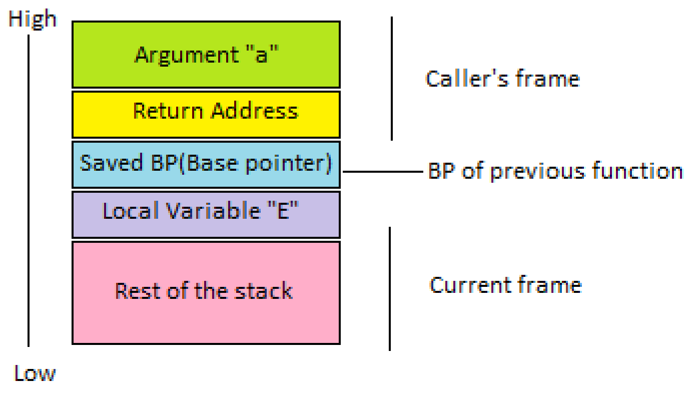


Figure 9 - Stack frame example

In this graphical representation of a stack frame the “High” represents higher up in memory and the “Low” represents lower in memory. Remember that the stack grows downwards in Intel processors.

We use the BP(base pointer) as a way to access local variables and function arguments of the function currently being executed. The BP(base pointer) becomes the value of the SP(stack pointer) at the start of the function just after storing the old BP(base pointer) on the stack. The SP(stack pointer) points to a location in memory which will be pushed or popped. Upon pushing to the stack the SP(stack pointer) is decrement a word in size. Upon popping from the stack the SP(stack pointer) is incremented a word in size.

The BP(base pointer) is useful as if we just relied on the SP(stack pointer) then while accessing variables we would need to account for the stack being used at any point during the functions execution. Such as a value being temporarily stored to the stack.

You can clearly see that function “foo” accepts an integer and this function argument variable is named “a”. When function “main” calls function “foo” it passes its local variable “z” to the function. This variable value gets pushed to the stack. The function “main” then calls function “foo” and the processor pushes the return address onto the stack automatically.

The program at this point will start executing function “foo”. Function “foo” will push the current BP(Base pointer) onto the stack. This BP(Base pointer) is the base pointer of the function caller and in this case it will be function “main”. This base pointer must be stored so that after returning from function “foo” the old base pointer can be restored and the function “main” can continue working properly. Without this functionality after returning from the function “foo” the function “main” will not be able to access its local variables correctly and nor will it be able to access the function arguments passed to its own function.

After storing the current BP(base pointer) the function “foo” then overwrites the current value of the BP(base pointer) with the value of the SP(stack pointer). Once this is done local variables of function “foo” can be accessed.

The SP(Stack pointer) then has to be subtracted by the size of all local scope variables this is to reserve the memory for use and without doing this you risk your local variables being overwritten by the stack when the stack is pushed to or popped from.

The function “foo” can access its local variables by accessing memory with the BP(base pointer) and subtracting it by x amount of bytes minus one. You must subtract it by at least one as otherwise you will be accessing or overwriting the old BP(base pointer) that was stored previously. How much you subtract it depends on a few things, the location of the variable relative to the scope, the size of the current variable and the type of variable you are accessing such as a variable with a primitive type or a variable that is a structure.

You must however subtract the sum or all previous variable sizes of the local scope that come before the variable you are accessing or writing to. This will result in the start position of the variable you are accessing on the scope.

The function “foo” can access its function arguments by accessing memory with the BP(base pointer) and adding it by x amount of bytes plus two. You must add two so you are not accessing or overwriting the return address on the stack. With function arguments data is accessed slightly differently on Intel processors. As Intel processors only allow you to push data that is a word in size you must either add a word for each variable you wish to access or ensure that you fit x number of bytes into the word before pushing it to the stack which will happen before the function call.

Many compilers also make use of registers to pass function arguments, this would overcome this problem until they run out of registers to use or a non-primitive type such as a structure needs to be accessed.

Before leaving our function, we must add to the stack pointer by the size of all local scope variables as we subtracted it earlier. Doing this will essentially allow the memory to be reused in another part of the program.

After restoring the stack pointer to its previous position we now need to pop off the old BP(base pointer). This is essential so that function “main” can continue as normal.

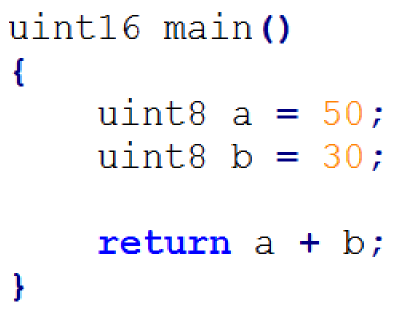
At this point we can now return from the function. The return address that was pushed to the stack by the processor upon calling the function is now restored and the processor begins executing from there.

6.5.3 Generation of assembly code

A compiler’s job is to take input and produce output. This output is typically assembly language but a code generator can output anything and is only valid if it maps directly with the input source code. Even if the code generator takes input and converts it to another programming language this is still code generation.

The output produced by a code generator must map directly to the source language the compiler is compiling. In simple terms the compiler that is producing your program must output code that does what your input source file was written to do.

Take the following input shown in Figure 10 below.



```
uint16 main()
{
    uint8 a = 50;
    uint8 b = 30;

    return a + b;
}
```

Figure 10 - Simple program that returns a + b written in Craft language

This is a simple program written in Craft language that returns “a + b”. After code generation, the following assembly language that will soon be assembled for the Intel processor is produced shown in Figure 11. This output maps directly with the input source code.

```
 1 global _main
 2 _main:
 3 push bp
 4 mov bp, sp
 5 sub sp, 2
 6 mov ax, 50
 7 mov [bp-1], al
 8 mov ax, 30
 9 mov [bp-2], al
10 xor ah, ah
11 xor ax, ax
12 mov al, [bp-1]
13 xor ch, ch
14 xor cx, cx
15 mov cl, [bp-2]
16 add ax, cx
17 add sp, 2
18 pop bp
19 ret
```

Figure 11 - Assembly output that adds two variables and returns the result

Looking at this assembly language may appear hard to understand for somebody who has not written in assembly language for the Intel processor before.

On line 1 we specify that the label "_main" is global and should be accessible by other parts of the program. This is not an instruction of the intel processor more of an instruction to the assembler. Globals will be handled at link time and the linker will resolve addresses of the global labels for an executable program.

On line 2 is the label "_main" this label represents the entry point of function "main". The reason this label is generated with an under scroll at the start of the label is to prevent functions names from conflicting with existing instructions of the processor. For example, without the under scroll someone could name the function "push" and it would represent a "push" instruction instead of a label which was originally intended. In assembly labels map to a part of an executable it's a way of saying we do not know the address of this part of the program yet but let's just call it "_main" and resolve it later.

On line 3 we are pushing our current BP(base pointer), the current BP(base pointer) is the BP(base pointer) of the previous function that called the function "main".

On line 4 we then overwrite the current BP(base pointer) with the stack pointer.

On line 5 we are subtracting the SP(stack pointer) by two this is required so that we have room in our stack frame for accessing local variables. We have two variables variable "a" and variable "b" they are both one byte long which is why we are subtracting two.

On line 6 we move decimal “50” into the AX(Accumulator) register.

On line 7 we are moving the lower 8 bits of the AX(Accumulator register) into memory address of the BP(base pointer) subtracted by one. This is variable “a”.

On line 15 you can see the same thing is happening but it is instead we are storing variable “b” in the CX(Counter) register.

On line 16 you can see that the AX and the CX registers are added together the result will be stored in the “AX” register as it is the first operand of the “add” instruction.

If the caller of function “main” cared about the return value, the “AX” register contains the result so the function caller would access that register. Obviously, the programmer is oblivious to all of this and appropriate code is generated at compile time by the code generator. In short when returning from functions the “AX” register is used to store the return result which is then accessed by the function caller.

This is a simple example of what a code generator does and for more complicated scenarios more complicated assembly code is generated.

6.6 ASSEMBLING

Assembling is a process where an assembler takes what is known as assembly language and produces machine code that is equivalent to the assembly language provided.

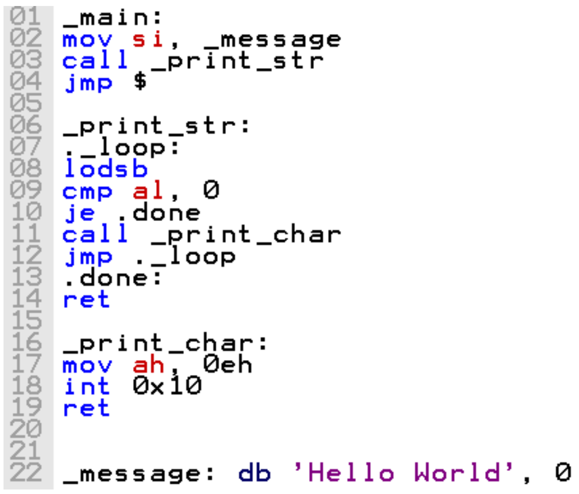
Assembly language maps directly with instructions of a processor. Assembly language exists because working with machine code is harder.

With assembly language, a single line of code, can represent a single instruction of the processor and is easier than writing the program in binary. Assembly language also allows the use of labels which represent places in memory relative to the program being assembled. Without this ability or without assembly language a programmer may need to manually re-offset all the instructions if they should change some of the program. Assembly language solves all these problems.

Assemblers assemble assembly language and each instruction in the assembly language is translated to exactly one target instruction. (Salomon, 1992).

Assembly language does not describe a single entity it is more of a category much like a cake could be a chocolate cake or a raspberry cake assembly language can describe instructions for an 8086 processor, a PIC16F628A processor or something else.

Assembly language maps directly to machine code which means writing in assembly language for the 8086 processor is very different from writing assembly language for the PIC16F628A processor. The assembly code written uses different instructions for every target processor.



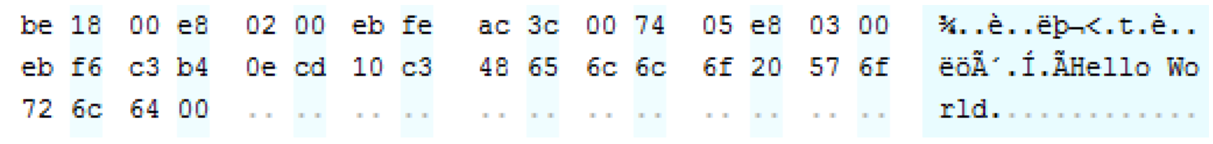
```
_main:
mov si, _message
call _print_str
jmp $

_print_str:
._loop:
lodsb
cmp al, 0
je .done
call _print_char
jmp ._loop
.done:
ret

_print_char:
mov ah, 0eh
int 0x10
ret

_message: db 'Hello World', 0
```

Figure 12 - Assembly language for the 8086 processor that prints "Hello World" to the screen



```
be 18  00 e8  02 00  eb fe  ac 3c  00 74  05 e8  03 00  ¾..è..ëþ¬<.t.è..
eb f6  c3 b4  0e cd  10 c3  48 65  6c 6c  6f 20  57 6f  ëöÃ´.Í.ÃHello Wo
72 6c  64 00  .. ..  .. ..  .. ..  .. ..  .. ..  .. ..  rld.............
```

Figure 13 - Machine code equivalent after assembling

Assembly code that targets the 8086 processor is shown in Figure 12 when ran through an 8086 assembler the machine code shown in Figure 13 is generated. The machine code has been represented as hexadecimal in this paper for easier reading.

Each instruction written in assembly language represents only one instruction. Some assemblers also include macros which enhance the experience of the assembly programmer.

6.7 OPTIMIZATION

Optimization is a concept about improving the quality, speed and reducing the size of a computer program while ensuring that the program performs as it was programmed to. Programs are often optimized during the compile-time stage. There are many types of optimization such as peephole optimizations, local optimizations, loop optimizations and more.

6.7.1 Peephole Optimizations

Peephole optimizations is a code optimization technique where local inspection of the object code is taken place to find and modify inefficient sequences of instructions. Peephole optimization takes place late in compilation and is done by sliding a *peephole* over the object program and replacing instruction sequences with shorter and faster instruction sequences. The peephole typically consists of two to three instructions in sequence with each other. (Chakraborty, 2014).

Peephole optimizations eliminate redundant instructions such as redundant load and store instructions, unreachable instructions, useless test and compare instructions, inconsequential instruction sequences. Peephole optimizations also eliminate coalescing of jump instructions. Strength reduction can also take place where slower operations are replaced with faster equivalents. (Chakraborty, 2014).

6.8 LINKING

Linking is the process of collecting and combing code and data into a single file that can be loaded into memory and executed. Linking can be performed at compile-time or at load time. (Bryant O'Hallaron, 2010). In other words, the executable loader of an operating system can perform linking of a program or the compiler can perform linking of object files. Typically, both are done.

Linking is done with a piece of software known as a linker.

Linkers have many benefits for one they make it possible for programs to be written in multiple programming languages which may not be possible without linkers. This is possible because compilers and assemblers can produce object files. These object files describe what has been compiled. An object file contains executable machine code along with data that was defined in the source file that was compiled.

In an object file the executable code is not complete. Offsets to pacific parts of the program may have not been calculated yet. This is either because the target data is on another segment or that the target data is external meaning it is in another object file. Many implementations of programs that create object files will not require an offset to be calculated at link time if the data is defined in the same source file and is in the same segment. The reason for this is because many processors including Intel allow you to do operations relative to the instruction in memory being executed, this means absolute addresses do not have to be known.

For static linking, any unknown offsets should be known by link time or this will result in a linking error.

Another reason linking is useful is that you do not have to re-compile the entire program each time you want a new executable. This is because object files hold a part of the program and not the full program. You may need a full re-build however if you break the API that the source file for an object file was using. In that case a rebuild is recommended or risk your program not running correctly.

So, linking simply is loading one or more object files and merging their contents together while fixing and relocating offsets that were not known until link-time with the hopes of generating an executable file.

6.8.1 Object Files

Object files hold different parts of a computer program that will help make up the executable file during link-time, load-time or run-time.

Object files come in three forms, relocatable object files, executable object files and shared object files. (Bryant O'Hallaron, 2010).

Relocatable object files contain binary code and data that can be combined with other relocatable object files at compile time to create an executable object file (Bryant O'Hallaron, 2010).

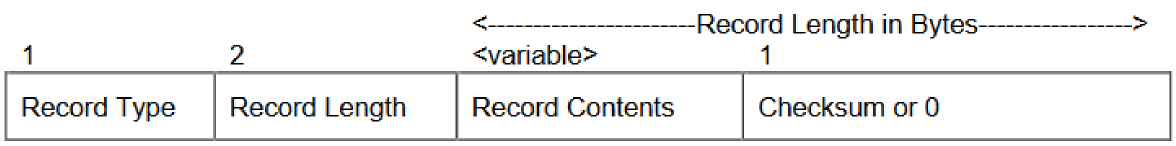
Executable object files contain binary code and data that can be copied directly to memory and then executed. (Bryant O'Hallaron, 2010). Windows executable files are an example of executable object files.

Shared object files are relocatable object files that can be loaded into memory and linked dynamically during when the program loads or during the programs execution. (Bryant O'Hallaron, 2010). An example of a shared object file is Linux's SO (Shared Object) file and for Windows the DLL (Dynamic Link Library) file.

6.8.1.1 OMF(Relocatable Object Module Format)

The OMF(Relocatable Object Module Format) is an old object format that was typically used in the DOS era.

The OMF design composes of records. These records describe a pacific part of the object file such as a segment or code or data that should be accessible outside of the object file. Some records also contain names which can then be referenced to by different records.



		<-----------------------Record Length in Bytes	----------------->
1	2	<variable>	1
Record Type	Record Length	Record Contents	Checksum or 0

Figure 14 - Graphical representation of the record structure for the OMF(Relocatable Object Module Format) (TIS Committee, 1995)

The format of an OMF record is shown above in Figure 14. The Record Type field is 1 byte in size and contains the type of record this is. If the least significant bit of the Record Type field is odd, then this indicates that certain numeric fields in the record contain 32-bit values. However, if the least significant bit is even then this indicates that the fields in the record contain 16-bit values. The fields that are affected by this least significant bit depend on the record type. (TIS Committee, 1995).

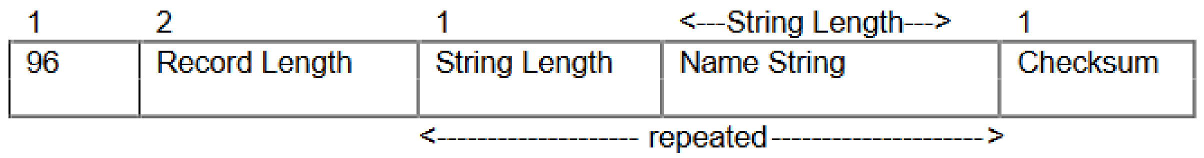
The Record Length field is 2 bytes long and holds the remaining number of bytes proceeding the Record Length field. Essentially this field will hold the size of the record contents plus one for the checksum. (TIS Committee, 1995).

The Record Contents field is the record payload it holds information for a record and its data depends on the record in question as each record is treated and formatted differently. The Record Contents field varies in size depending on the type of record. (TIS Committee, 1995).

The Checksum field is a 1 byte field and contains the negative sum (modulo 256) of all other bytes in the record. Many compilers ignore this checksum field and just write a zero byte rather than computing a checksum. (TIS Committee, 1995).

After each record comes another record or the end of the file.

The LNAME's record



1	2	1	<---String Length--->	1
96	Record Length	String Length	Name String	Checksum
		<-------------------- repeated	--------------------->	

Figure 15 - Graphical representation of the LNAMES Record in the OMF(Relocatable Object Module Format) (TIS Committee, 1995)

The LNAME's record structure is shown above in Figure 15. The LNAME record can hold multiple strings a NULL name is also valid. This record holds names that are related to the target program these names include segment names, and external reference names. (TIS Committee, 1995).

The OMF(Relocatable Object Module Format) has many complicated records such as the SEGDEF record which describes a segment such as the "data" or "code" segment. Another complicated record is the FIXUPP record which describes offsets that need to be fixed during link time as they are currently unknown.

7 LITERATURE REVIEW

7.1 INTRODUCTION

Please read section 6 *“A deeper understanding of compiler Internals”* before reading this literature review.

Compilers and the electronic computer have been in existence for a short period compared to other technologies. Compilers have influenced society dramatically. It is hard to imagine a world without computers and in this stage in time if all computers stopped existing we would be back in the dark ages due to how much is dependent upon them. Think of the traffic light that regulates traffic to avoid an accident, think of the satellites that monitor our weather and can predict disasters before they have even happened, think of the interconnectivity and how you can talk to any place in the world in a fraction of a second. Computers make this possible and the compilers job is to make the programmer’s job easier and cleaner. Without compilers and assembler’s programmers would be writing in machine code and it is unlikely we would ever be at the stage we are without them. Writing in machine code is a tedious process and something as simple as adding a new instruction may lead to many instruction offsets requiring to be changed manually. This literature review aims to show the history of compilers.

7.2 COMPILER HISTORY

The first programming language was first proposed in 1949 by John Mauchly the language was known as Short Code. William Schmitt implemented Short Code in the same year for the BINAC computer and then in 1950 rewrote Short Code for the UNIVAC computer with the assistance of Albert B.Tonik and J. Robert Logan. Details of the UNIVAC Short Code were never published (Metropolis, 1980). Short code was interpreted and not compiled.

In October 1951 Grace Hopper began to write the world’s first compiler and in 1952 it was complete. The compiler compiled a language called the A0-System (Beyer, 2009).

A compiler-compiler is a piece of software which you pass a grammar for a target language and you pass an instruction set for your target system and the compiler will create a compiler for you. This concept was the idea of Tony Brooker and was developed at the University of Manchester between 1960 and 1964. (Lavington, 2016)

The LR Parser was invented by Donald Knuth in 1965 in a paper named “On the Translation of Languages from Left to Right” LR parsers read input from left to right with a Rightmost derivation. (Knuth, 1965).

The concept of LL(1) grammars was first introduced by Lewis and Stearns in 1968. (Esik, 1993).

Korenjak in 1969 was the first to show that parsers for programming languages could be produced using the parsing techniques that Donald Knuth introduced (Johnson, 2012).

In 1954 IBM had a team of mathematician’s and programmers to develop Fortran, one of the first programming languages and compilers. The team was led by John Backus and he did a good job even though the hardware was limited. (Nilges, 2004).

Many problems had to be solved during the development of the 704 FORTRAN 1 compiler including many optimization problems. Nine people who were principal planners and programmers of this compiler would come together to solve these problems and prove for the first time that efficient object programs could be compiled for a machine with built-in floating point and indexing. (Metropolis, 1980). The development of this compiler was from 1954 to 1957. (Anon., n.d.).

Robert A. Nelson and Irving Ziller constructed methods for analysing and optimizing loops and references to arrays. Their methods could move computations from the program to the compiler and from inner to outer loops when a situation was identified where this was possible. The optimizer could also identify a circumstance where a single instruction in the exit path of a loop could be removed. (Metropolis, 1980).

Sheldon Best invented solutions for optimizing the use of index registers. The optimization was based on the expected frequency of execution of various parts of the program. (Metropolis, 1980).

Peephole optimization is a type of optimization that is performed over a small set of instructions with the goal of replacing sets of instructions with shorter and faster sets of instructions. Peephole optimization was invented by William McKeeman in 1965. (Bhatt & Harshad, 2013).

Lex is a lexical analyser generator written by Mike Lesk and intern Eric Schmidt in 1975. (Levine, 2009). Lex is written along with C or Ratfor and when put through the Lex program generator source code is produced for the given grammer. The format of Lex source starts with definitions, then rules then finally user subroutines. The definitions section may contain the selection of a host language, a character set table, a list of start conditions or adjustments to the default size of arrays. The rules section allows you to provide a requirement syntax and an action which will happen if the requirement is found in the given input. The user subroutines are just defined functions that the programmer may wish to call at a given point in time. (Lesk & Schmidt, 1975).

Yacc is a tool for imposing structure on the input to a computer program. A programmer using Yacc defines rules for describing input structure, these rules also specify code to invoke when the rules are recognised. A low-level routine is used to do the basic input. (Johnson, 1975). Once the source code is run through Yacc the Yacc interpreter generates a function that's called a *parser*. This function calls the user-supplied low-level input routine called the lexical analyser to pick up items called tokens from the input stream. The tokens are organised according to the input rules that are also called *grammer* *rules* and when one of the rules is recognised the supplied action for the particular rule is invoked. (Johnson, 1975).

The C programming language was planned in the early 1970s as a system implementation language for the nascent Unix operating system and came into being in the years of 1969-1973. By early 1973 the essentials of modern C were complete. The C language was strong enough for the Unix kernel to be rewritten for the PDP-11 in the summer of that year by Dennis Ritchie and Ken Thompson. During 1973-1980 the C programming language's type structure obtained unsigned, long, union and enumeration types. (Ritchie, n.d.).

The first high level language to have a self-hosting compiler was called NELIAC. This compiler was written for Lisp by Hart and Levin at MIT in 1962. They wrote a Lisp compiler in Lisp, and tested the software in an existing Lisp interpreter. Once the compiler had been improved the compiler could then compile its own source code, making it the world's first self-hosting compiler for a high-level language. (Wikipedians, n.d.)

PL/C is a programming language that was developed at Cornell University it is based on IBM's PL/I language. PL/C was designed in the early 1970's to help teach people how to program a computer. Cornell also developed a compiler for PL/C and it was based on its earlier CUPL compiler and was used in college-level programming courses. (Conway & Gries, 1975).

WATBOL is the University of Waterloo COBOL compiler. The compiler was designed to compile COBOL programs quickly and provide error diagnostics after the statement that had an error. The compiler was also designed to help students to debug their own programs without receiving help from the tutor. (Hurdal, et al., 1972) In 1969-1970 the WATBOL compiler was completed. (University Of Waterloo, n.d.).

The Pascal programming language was first designed in 1971 by Niklaus Wirth who was a professor at the Polytechnic of Zurich in Switzerland. Pascal was designed to be a simplified version of the Alogol language and was designed for educational purposes. The Algol language dates from 1960. (Cantù, 2008).

In 1979 work on C with Classes began, by 1983 the first C++ implementation was in use, by 1984 C with classes was named C++, by 1985 the first commercial C++ compiler was released and was named Cfront., by 1987 GNU C++ was released. (Stroustrup, 2007).

C with Classes was designed and implemented by Bjarne Stroustrup as a research project in the Computing Science Research Center of Bell Labs. When Bjarne Strousturp joined in 1979 he was told to “do something interesting”. He was given suitable computer resources and encouraged to talk to people. He was given one year before having to present his work for evaluation. (Stroustrup, 2007).

8 CRAFT COMPILER

8.1 INTRODUCTION

Craft Compiler is an open source project that was developed to gain the experience needed to write this paper and so the project could be used as a proof of concept for what has been wrote about.

Craft Compiler is a general purpose compiler with over 20000 lines of code. Craft Compiler is also a compiler framework. The framework allows someone to write code generators for Craft language that target pacific processor architectures.

The framework also allows for object format readers and writers to be written and allows for linkers to be written. Code generators, object format reader and writers, and linkers are all compiled to DLL(Dynamic Link Libraries) so that they can be imported by the compiler. Once that happens communication between the compiler and the libraries can begin.

Since the compiler must be compatible with multiple object format readers and writers as well as multiple code generators and linkers many of these systems are abstract so that the compiler can talk with an interface between the compiler and the element in question. Code generators, linkers and object formats all extend their own abstract classes which are used as an interface by the compiler.

Currently Craft Compiler can only target the 8086 processor. Bootable code can be written in Craft language but must be aided by assembly language. MS-DOS COM applications can also be written.

You can find the source code for Craft Compiler on GitHub here: https://github.com/nibblebits/craft-compiler

Craft Compiler also depends on another project that was written for this compiler. This project is called MagicOMF and allows the compiler to read and write files that represent the OMF(Relocatable Object Module Format). MagicOMF is also open source and can be found here: https://github.com/nibblebits/MagicOMF.

Craft Compilers official website can be found here: http://craft-language.org.

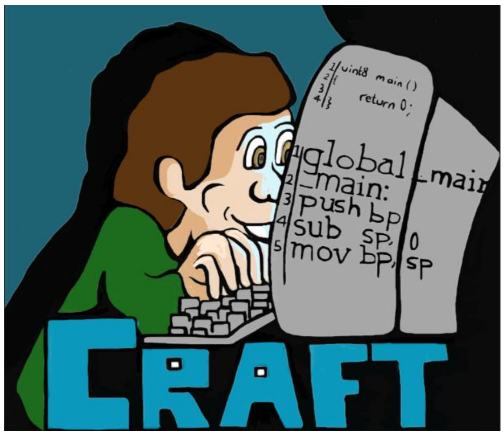


Figure 16 - Logo for Craft Language; Illustrated by Ellen Rees

8.2 History

The idea for this dissertation paper stems back to year one of my computing degree in 2014-5 where I thought about creating a scripting language or programming language which was originally going to be called “Feather” but then renamed to “Arrow”. In multimedia class in year one of my Computing degree we had to make a website CV as an assignment and in the website CV that I made I wrote about these ideas and stated “The point of Arrow is that it will be able to be linked with other executables to allow the language to be interpreted in other applications. This is an important part of the language as it will mean it will mean you could make plugin systems within your application and have the plugins be written in Arrow. I would provide functions built into the interpreter library that will allow the programmer of the other application to add their own custom functions directly into the interpreter or even over ride other functions that already exist within the interpreter. “ (McCarthy, 2014-5).

As I started to develop the project I decided to instead call the scripting language Elf but then I thought to myself it sounds too much like the ELF(Executable and Linkable Format) which is an object format. I spoke to Ana Cauldron who was not my super-visor at the time to see what she thought about the issue and she agreed I should change the name.

At this point I decided I should just start again as the current implementation was not great and I also wanted to create a compiler instead of an interpreter. So, I did just that I started again and this time I was creating a compiler that targeted Goblin language.

After many months of work, I decided the name Goblin did not sound right and it took a long time to find a better name, I had many suggestions from friends and from strangers and eventually came up with the name Craft. I renamed the whole project to Craft and began further development of the Craft compiler and Craft language.

The project was started on the 27th of May 2016 and its first release was on the 3rd of April 2017. The development of Craft compiler took ten months or 311 days.

Almost every day of this duration time was spent on the project.

8.3 Lexical Analysis and Parsing

Creating a Lexer for Craft compiler was straight forward as I had already had previous experience from my failed attempt of writing an interpreter when creating ELF.

The parsing on the other hand was tedious and difficult. The design I chose was a shift reduce parser that accepted a grammer and attempted to create an AST(Abstract Syntax Tree). This worked well in many cases but there were times rules would conflict as my implementation of the shift reduce parser was poor. You can read more about some of the parser problems I was facing in the compiler diary: http://www.craft-language.org/Diary.pdf on page 6 and 7 and throughout other pages in the diary.

Eventually I chose to start writing the parser again but I decided to program the entire parser rather than relying on a grammer being inputted and creating an AST(Abstract Syntax Tree) based on the grammer.

This worked very well and it was the C compiler named “8cc” that gave me the idea of this as they too parsed all input through programming rather than accepting a grammer.

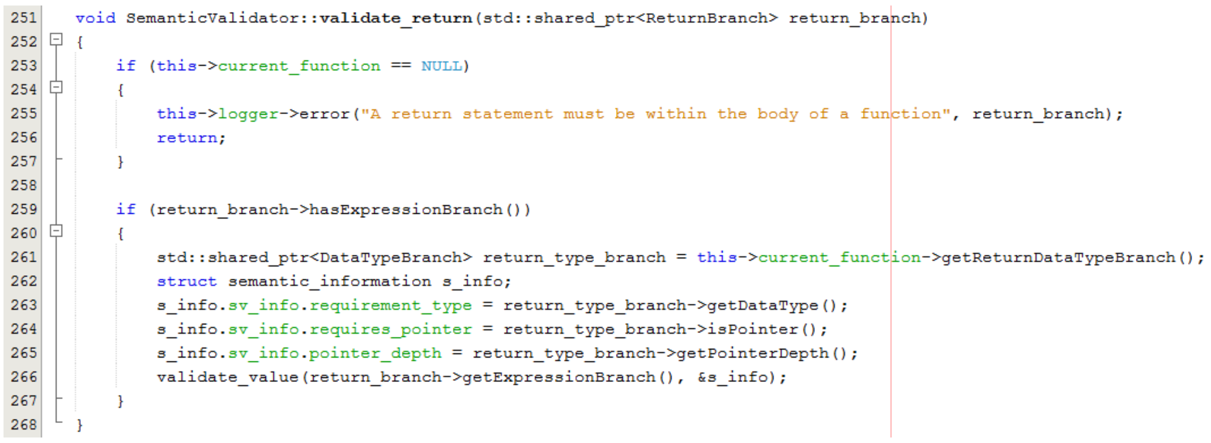
All parsing problems went away at this point as I had full control of what would happen next. An example of the algorithm for parsing a return statement is shown below in English.

Peek at next token

```
Is the next token a keyword and has a value of “return”
    Create a “return” branch
    Pop the return keyword from the input stack
    Is the next token anything other than a semicolon
        Process the expression
        Pop the branch and store it in variable A
        Set the return branch expression to variable A
    Process the semicolon
    Push the return branch to the output stack
```

8.4 SEMANTIC VALIDATOR

The semantic validator was easy to do. All that had to be done was loop through the entire abstract syntax tree and ensure that it matches given rules for an event. For example, take the Craft language code: *uint8 a = 333;* The compiler’s semantic validator will see that an assignment is being made, it will then create some temporary rules such as the type that a given value must fit into. In this case, it is *uint8.* Now the semantic validator will begin validating the value and it will see the number *333.* The data type *uint8* can only hold 8 bits in the Craft programming language this means only a maximum number of 255 can be stored. Based on the rules created earlier the semantic validator knows that it must fit into a *uint8* and realises that decimal *333* is above decimal 255 and this causes the semantic validator to produce a compiler warning.



```
251  void SemanticValidator::validate_return(std::shared_ptr<ReturnBranch> return_branch)
252  {
253      if (this->current_function == NULL)
254      {
255          this->logger->error("A return statement must be within the body of a function", return_branch);
256          return;
257      }
258
259      if (return_branch->hasExpressionBranch())
260      {
261          std::shared_ptr<DataTypeBranch> return_type_branch = this->current_function->getReturnDataTypeBranch();
262          struct semantic_information s_info;
263          s_info.sv_info.requirement_type = return_type_branch->getDataType();
264          s_info.sv_info.requires_pointer = return_type_branch->isPointer();
265          s_info.sv_info.pointer_depth = return_type_branch->getPointerDepth();
266          validate_value(return_branch->getExpressionBranch(), &s_info);
267      }
268  }
```

Figure 17 - Validation of return statements in Craft compiler

In Figure 17 the method for validating return statements in Craft compiler is shown. On line 253 the semantic validator ensures it is currently inside a function as if it is not then this is an error. The parser should make it impossible for this to ever be the case so I am considering removing that code. On line 259 we are checking that the return statement has an expression and if it does we validate its value based on the current function’s return type. The *validate_value* method will then ensure that the expression matches the given rules.

8.5 Code Generator

When developing this project, I needed a target platform.

Originally the code generator I was writing was not for an existing processor but instead to a virtual instruction set I designed that would be run on a virtual machine I would have written. The lack of design I had was that I was generating directly to byte-code rather than to a form of assembly language first. This made things a lot harder as it meant the code generator had more to do such as remembering offsets throughout the binary file. With assembly language, this would be a lot easier as a label could have been used.

I eventually chose to write a code generator for the 8086 processor instead primarily because I was familiar with the architecture and because raw binaries can be run on MS-DOS which also runs on the 8086 processor. If I chose a 32 bit platform and wanted to target Windows I would have been forced to implement the PE(Portable Executable) format for Windows. This would have been difficult for the given time frame.

Upon developing the 8086 code generator things started to get a lot harder when I had to work with scope variables. Scope variables are much harder to access than global variables. Accessing scope variables requires understanding of the 8086 stack and principles on how to store and access data on the stack properly and efficiently. For example when accessing a local scope variable Craft compiler subtracts from the BP(base pointer) the size of the variable along with the size of the scope up to the variable in question and all parent scopes above it. It gets more complicated when accessing arrays and structures as you then have to do the same as previously mentioned but also then add to the BP(Base Pointer) by a certain amount depending on the array index and structure variable you are accessing. The reason you must add to the BP(base pointer) while accessing structures and arrays on the scope is to maintain compatibility between global variables and scope variables. The stack grows downwards and you access local scope variables by subtracting the BP(base pointer) but when accessing global variables you proceed upwards in memory not downwards. This is the reason when accessing array and structure variables in a scope you must subtract the position of the variable on the scope and subtract the size and then finally add however many bytes to access what you are looking for.

I designed the framework to calculate an offset relative to zero or relative to a scope this solved a lot of my positioning problems and it required many rewrites and modifications before I got it right.

There are many times where the position cannot be known at compile time I also designed the framework to handle this it calls functions that the code generator will pass to it that are invoked whenever the position cannot be fully known at compile time. The code generator can then generate assembly code that will fill in the blanks.

The hardest part about the code generator was developing an efficient positioning system.

But there are many other challenges to face with a code generator such as when returning from a function while inside nested scopes the compiler needs to subtract from the SP(stack pointer) by the size of the scope that the return statement is present in and all scopes above it including the function scope to avoid corrupting the stack.

Global variables, function arguments, and scope variables are all accessed differently. Global variables proceed upwards in memory, scope variables down, and function arguments also proceed upwards in memory but each element is a word long. Even if the variable the function accepts is a *uint8* it will still access it as a word in size. This is because the stack only allows the pushing of words and function arguments are pushed to the stack before calling a function.

8.6 VIRTUAL OBJECT FORMAT AND THE OMF(RELOCATABLE OBJECT MODULE FORMAT)

As Craft compiler is also a framework and it does not have one pacific target, it can have any object format, any code generator and any linker.

I had to develop a virtual object format layer that would abstractly treat all object formats the same. I came up with an idea of having a virtual object system. In this system all object formats follow a similar structure, they all can register global and external entities and they can all have segments.

This type of setup allowed for objects to all be treated the same by a linker, without the linker having pacific instructions for a pacific object format. It just communicates with the virtual object format layer. The object format module for the object type handles its own internals such as writing and reading to a file and sticks to this virtual abstraction interface.

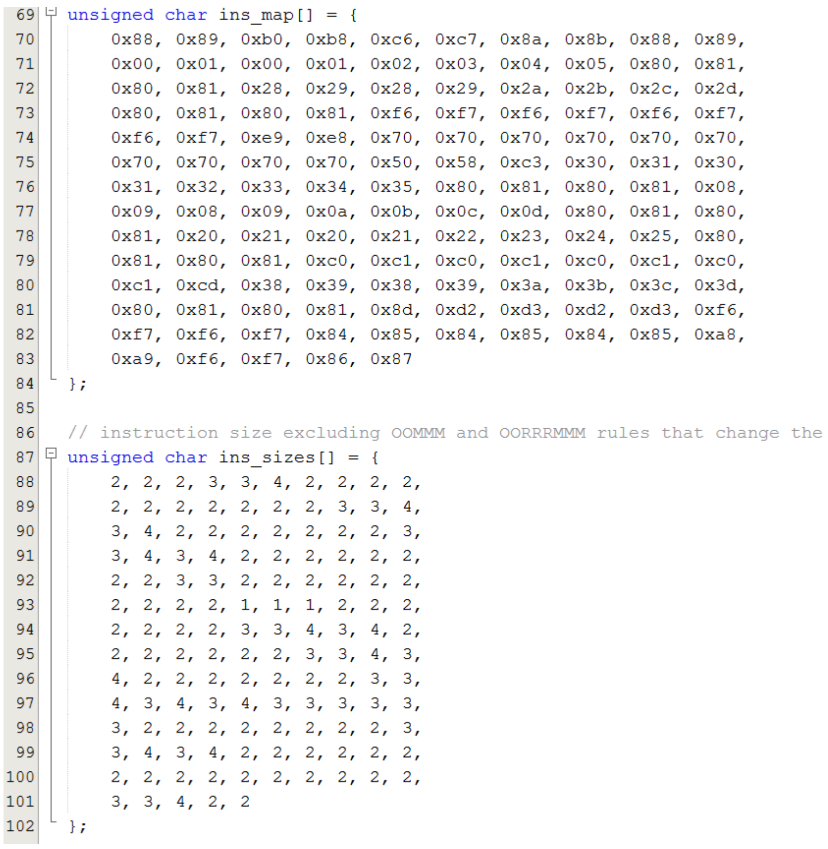
The object format I chose to implement for this project was OMF(Relocatable Object Module Format). OMF is explained in detail in section 6.8.1.1 of this paper.

8.7 ASSEMBLER

During late development of the 8086 code generator, I then had to start writing an assembler to assemble the assembly code generated by the code generator. At first I decided to design it so every assembly instruction would be handled by its own method. This proved to be time consuming and I decided to be clever with how I designed the assembler. I started noticing patterns in the instruction set for the 8086 code generator. Here is the instruction set reference I used:
https://courses.engr.illinois.edu/ece390/resources/opcodes.html

If you read through that web page you will notice in the “Instructions and opcodes” section, there are five categories. If you now look through the instructions you will probably start to notice a pattern as well, these categories are mixed inside instructions and are common. Essentially there is only so many instruction patterns that can exist which was good for me as this meant I could create a design that would allow me to easily add instructions.

I decided I would use arrays, structures, rules and logic to guide the assembler and with this design whenever I wanted to add a new assembler instruction I would just append the arrays.

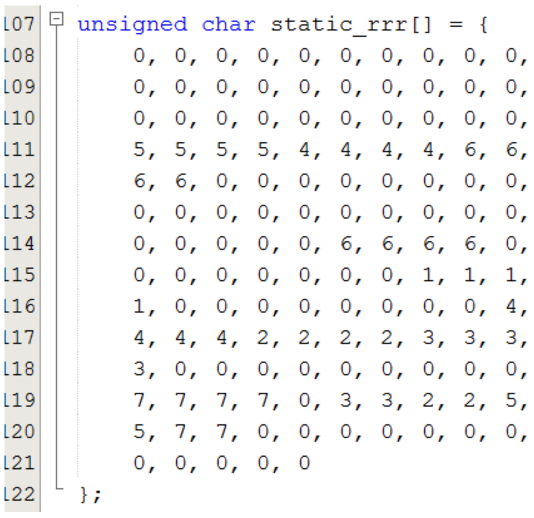


```c
unsigned char ins_map[] = {
    0x88, 0x89, 0xb0, 0xb8, 0xc6, 0xc7, 0x8a, 0x8b, 0x88, 0x89,
    0x00, 0x01, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x80, 0x81,
    0x80, 0x81, 0x28, 0x29, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d,
    0x80, 0x81, 0x80, 0x81, 0xf6, 0xf7, 0xf6, 0xf7, 0xf6, 0xf7,
    0xf6, 0xf7, 0xe9, 0xe8, 0x70, 0x70, 0x70, 0x70, 0x70, 0x70,
    0x70, 0x70, 0x70, 0x70, 0x50, 0x58, 0xc3, 0x30, 0x31, 0x30,
    0x31, 0x32, 0x33, 0x34, 0x35, 0x80, 0x81, 0x80, 0x81, 0x08,
    0x09, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x80, 0x81, 0x80,
    0x81, 0x20, 0x21, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x80,
    0x81, 0x80, 0x81, 0xc0, 0xc1, 0xc0, 0xc1, 0xc0, 0xc1, 0xc0,
    0xc1, 0xcd, 0x38, 0x39, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d,
    0x80, 0x81, 0x80, 0x81, 0x8d, 0xd2, 0xd3, 0xd2, 0xd3, 0xf6,
    0xf7, 0xf6, 0xf7, 0x84, 0x85, 0x84, 0x85, 0x84, 0x85, 0xa8,
    0xa9, 0xf6, 0xf7, 0x86, 0x87
};

// instruction size excluding OOMMM and OORRRMMM rules that change the
unsigned char ins_sizes[] = {
    2, 2, 2, 3, 3, 4, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 3, 3, 4,
    3, 4, 2, 2, 2, 2, 2, 2, 2, 3,
    3, 4, 3, 4, 2, 2, 2, 2, 2, 2,
    2, 2, 3, 3, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 1, 1, 1, 2, 2, 2,
    2, 2, 2, 2, 3, 3, 4, 3, 4, 2,
    2, 2, 2, 2, 2, 2, 3, 3, 4, 3,
    4, 2, 2, 2, 2, 2, 2, 2, 3, 3,
    4, 3, 4, 3, 4, 3, 3, 3, 3, 3,
    3, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    3, 4, 3, 4, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 4, 2, 2
};
```

Figure 18 - 8086 assembler instruction map and instruction sizes arrays

In Figure 18 we have two arrays defined, *ins_map* and *ins_sizes.* The *ins_map* array maps indexes of the array to opcodes of the 8086 instruction set. The *ins_sizes* array maps indexes to instruction sizes.

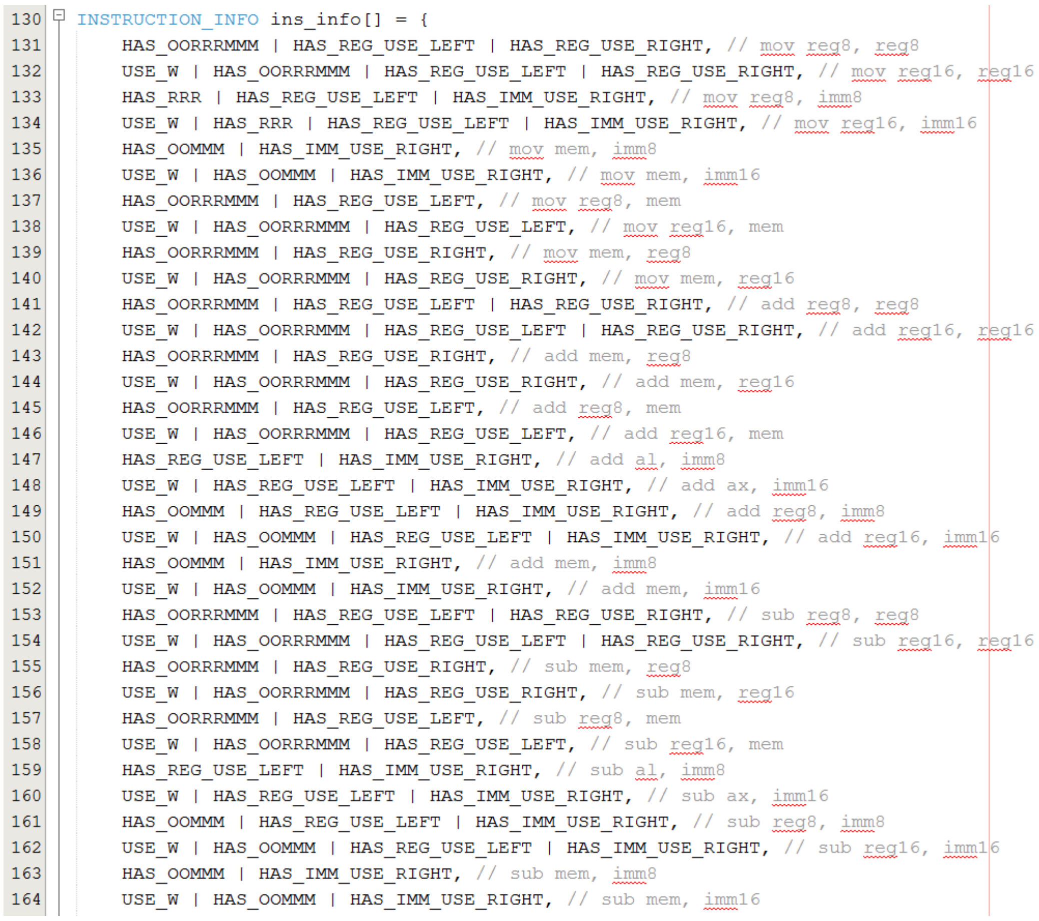


```
unsigned char static_rrr[] = {
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        5, 5, 5, 5, 4, 4, 4, 4, 6, 6,
        6, 6, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 6, 6, 6, 6, 0,
        0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        1, 0, 0, 0, 0, 0, 0, 0, 0, 4,
        4, 4, 4, 2, 2, 2, 2, 3, 3, 3,
        3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        7, 7, 7, 7, 0, 3, 3, 2, 2, 5,
        5, 7, 7, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0
};
```

Figure 19 - 8086 assembler rrr array

The “static_rrr” array shown in Figure 19 holds the *rrr* value for an instruction. *rrr* is a definition inside the instruction set reference that specifies a part of a byte in an instruction.

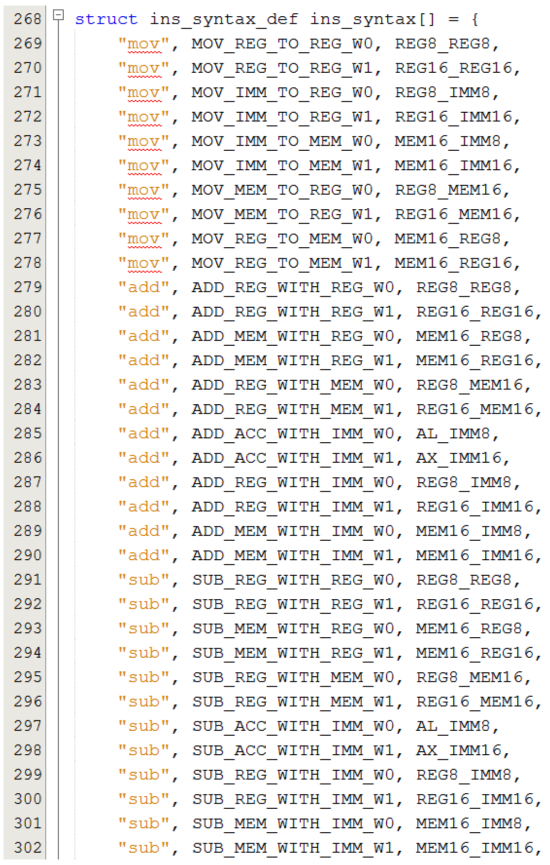
If *rrr* is not valid for an instruction then a zero is filled, however a zero can still represent a valid *rrr* value. Whether or not the “static_rrr” array is used depends entirely on the assembly instruction.



```
INSTRUCTION_INFO ins_info[] = {
    HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // mov reg8, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // mov reg16, reg16
    HAS_RRR | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // mov reg8, imm8
    USE_W | HAS_RRR | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // mov reg16, imm16
    HAS_OOMMM | HAS_IMM_USE_RIGHT, // mov mem, imm8
    USE_W | HAS_OOMMM | HAS_IMM_USE_RIGHT, // mov mem, imm16
    HAS_OORRRMMM | HAS_REG_USE_LEFT, // mov reg8, mem
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT, // mov reg16, mem
    HAS_OORRRMMM | HAS_REG_USE_RIGHT, // mov mem, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_RIGHT, // mov mem, reg16
    HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // add reg8, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // add reg16, reg16
    HAS_OORRRMMM | HAS_REG_USE_RIGHT, // add mem, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_RIGHT, // add mem, reg16
    HAS_OORRRMMM | HAS_REG_USE_LEFT, // add reg8, mem
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT, // add reg16, mem
    HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // add al, imm8
    USE_W | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // add ax, imm16
    HAS_OOMMM | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // add reg8, imm8
    USE_W | HAS_OOMMM | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // add reg16, imm16
    HAS_OOMMM | HAS_IMM_USE_RIGHT, // add mem, imm8
    USE_W | HAS_OOMMM | HAS_IMM_USE_RIGHT, // add mem, imm16
    HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // sub reg8, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT | HAS_REG_USE_RIGHT, // sub reg16, reg16
    HAS_OORRRMMM | HAS_REG_USE_RIGHT, // sub mem, reg8
    USE_W | HAS_OORRRMMM | HAS_REG_USE_RIGHT, // sub mem, reg16
    HAS_OORRRMMM | HAS_REG_USE_LEFT, // sub reg8, mem
    USE_W | HAS_OORRRMMM | HAS_REG_USE_LEFT, // sub reg16, mem
    HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // sub al, imm8
    USE_W | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // sub ax, imm16
    HAS_OOMMM | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // sub reg8, imm8
    USE_W | HAS_OOMMM | HAS_REG_USE_LEFT | HAS_IMM_USE_RIGHT, // sub reg16, imm16
    HAS_OOMMM | HAS_IMM_USE_RIGHT, // sub mem, imm8
    USE_W | HAS_OOMMM | HAS_IMM_USE_RIGHT, // sub mem, imm16
```

Figure 20 - 8086 assembler ins_info array

The “ins_info” array shown in Figure 20 guides the assembler with generating the appropriate machine code for a given instruction, it holds essential information for an instruction such as weather it has a register on the right, weather it is expecting a 16-bit register or weather it needs to use the “static_rrr” array and more.



```
268 struct ins_syntax_def ins_syntax[] = {
269     "mov", MOV_REG_TO_REG_W0, REG8_REG8,
270     "mov", MOV_REG_TO_REG_W1, REG16_REG16,
271     "mov", MOV_IMM_TO_REG_W0, REG8_IMM8,
272     "mov", MOV_IMM_TO_REG_W1, REG16_IMM16,
273     "mov", MOV_IMM_TO_MEM_W0, MEM16_IMM8,
274     "mov", MOV_IMM_TO_MEM_W1, MEM16_IMM16,
275     "mov", MOV_MEM_TO_REG_W0, REG8_MEM16,
276     "mov", MOV_MEM_TO_REG_W1, REG16_MEM16,
277     "mov", MOV_REG_TO_MEM_W0, MEM16_REG8,
278     "mov", MOV_REG_TO_MEM_W1, MEM16_REG16,
279     "add", ADD_REG_WITH_REG_W0, REG8_REG8,
280     "add", ADD_REG_WITH_REG_W1, REG16_REG16,
281     "add", ADD_MEM_WITH_REG_W0, MEM16_REG8,
282     "add", ADD_MEM_WITH_REG_W1, MEM16_REG16,
283     "add", ADD_REG_WITH_MEM_W0, REG8_MEM16,
284     "add", ADD_REG_WITH_MEM_W1, REG16_MEM16,
285     "add", ADD_ACC_WITH_IMM_W0, AL_IMM8,
286     "add", ADD_ACC_WITH_IMM_W1, AX_IMM16,
287     "add", ADD_REG_WITH_IMM_W0, REG8_IMM8,
288     "add", ADD_REG_WITH_IMM_W1, REG16_IMM16,
289     "add", ADD_MEM_WITH_IMM_W0, MEM16_IMM8,
290     "add", ADD_MEM_WITH_IMM_W1, MEM16_IMM16,
291     "sub", SUB_REG_WITH_REG_W0, REG8_REG8,
292     "sub", SUB_REG_WITH_REG_W1, REG16_REG16,
293     "sub", SUB_MEM_WITH_REG_W0, MEM16_REG8,
294     "sub", SUB_MEM_WITH_REG_W1, MEM16_REG16,
295     "sub", SUB_REG_WITH_MEM_W0, REG8_MEM16,
296     "sub", SUB_REG_WITH_MEM_W1, REG16_MEM16,
297     "sub", SUB_ACC_WITH_IMM_W0, AL_IMM8,
298     "sub", SUB_ACC_WITH_IMM_W1, AX_IMM16,
299     "sub", SUB_REG_WITH_IMM_W0, REG8_IMM8,
300     "sub", SUB_REG_WITH_IMM_W1, REG16_IMM16,
301     "sub", SUB_MEM_WITH_IMM_W0, MEM16_IMM8,
302     "sub", SUB_MEM_WITH_IMM_W1, MEM16_IMM16,
```

Figure 21 - 8086 assembler ins_syntax array

The “ins_syntax” array shown in Figure 21 guides the assembler with choosing the correct instruction to generate, it holds the instruction name along with the instruction type it matches to and finally the operand rules it must meet for this to be the valid instruction.

Take the first element on line 269, this states that if you have a *mov* instruction where both operands are of *reg8* then choose the *MOV_REG_TO_REG_W0* instruction. The following assembly code *mov cl, cl* would meet that requirement.

The assembler does not generate the instruction as a flat binary, instead it generates the instruction into a virtual segment which is a part of the virtual object format layer discussed previously in the paper. After assembling is complete the object format writer for the target object format can then take this information generated by the assembler and produce an appropriate object file out of it.

8.8 LINKER

The Craft compiler also comes with a linker built in. This linker like most things in Craft compiler acts as a virtual layer so that multiple linkers can be written without changing the design of the compiler. In Craft compiler once a linker is loaded object files are added to it and then it is told to link. Depending on the type of linker that is used will determine the executable file that is created.

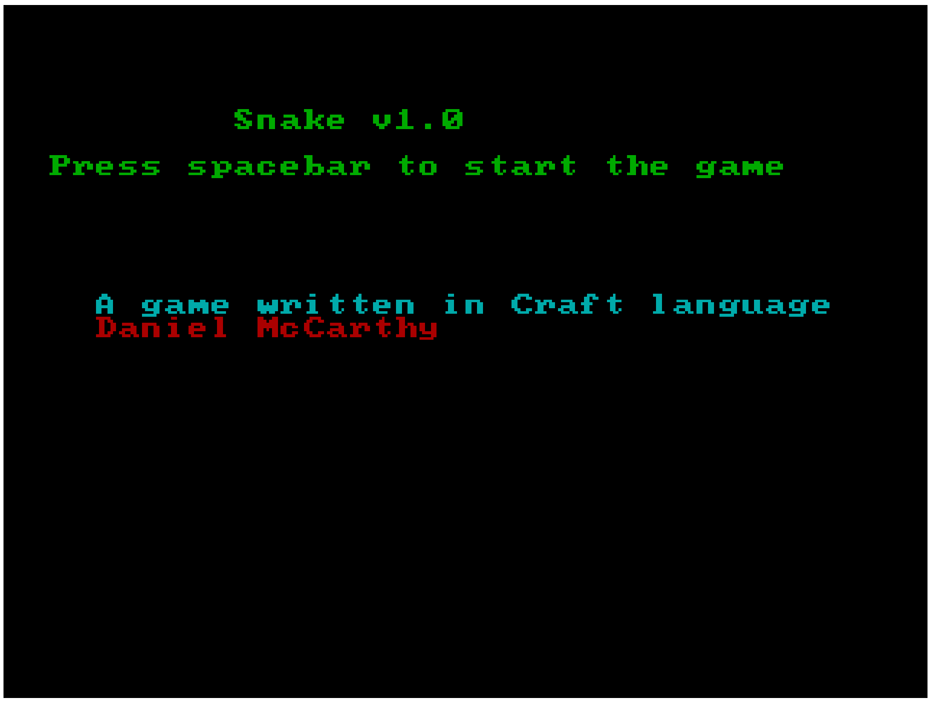
8.8.1 The Bin Linker

The bin linker is a linker module for Craft compiler that can link objects into a single raw binary file. This raw binary file can then be run directly on the processor that the object files are targeting. This is the case when linking object files created for the *8086 processor*. The executable generated could then be run on MS-DOS as a COM file which is just a flat binary executable that MS-DOS can run.

8.9 SNAKE GAME

To demonstrate Craft compiler's power a snake game was written in Craft language and assembly language you can find the source code for the main file which was written in Craft language here: https://github.com/nibblebits/craft-compiler/blob/master/bin/code_examples/8086/Snake/main.craft

Below in Figure 22, Figure 23 and Figure 24 are screenshots of the game.



```
Snake v1.0
Press spacebar to start the game


A game written in Craft language
Daniel McCarthy
```

Figure 22 - Snake v1.0 main menu

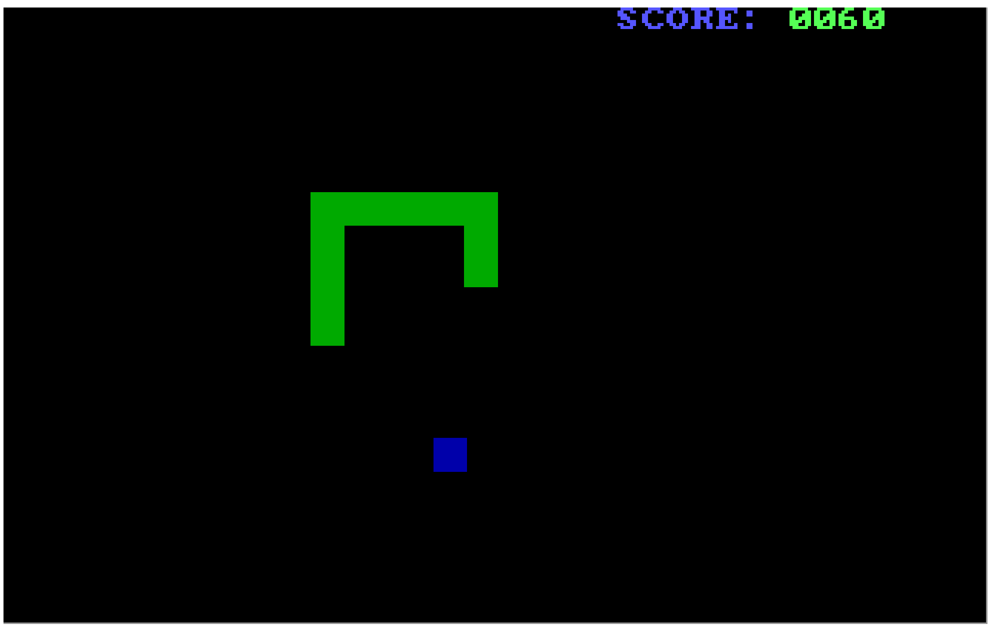
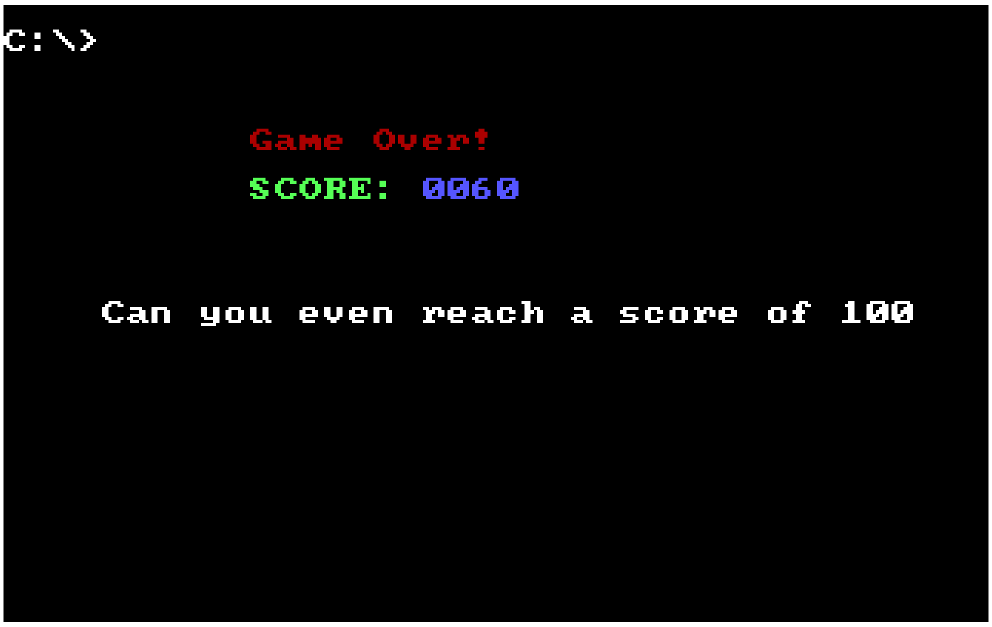


Figure 23 - Snake v1.0 game play



```
C:\>

          Game Over!
          SCORE: 0060


    Can you even reach a score of 100
```

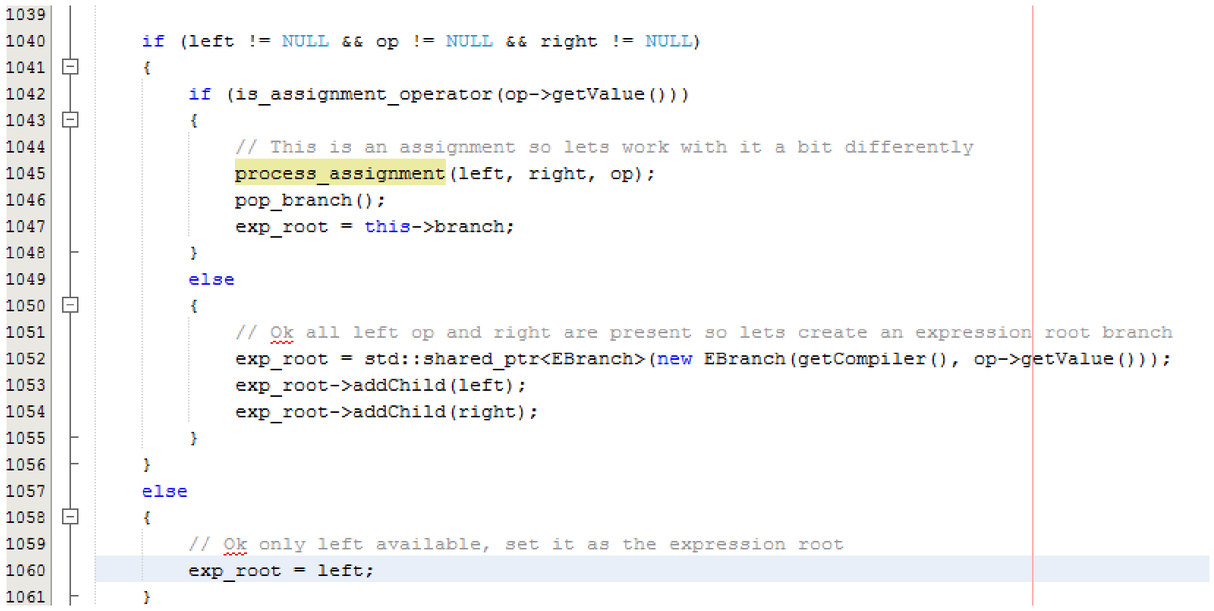
Figure 24 - Snake v1.0 game over screen

The snake game mocks the player if they do bad to try to make the player want to get a better score. If the player has a good score more positive messages are shown.

8.10 What should have been done differently

One of the biggest flaws in Craft compiler is also an advantage. Craft compiler's branches in the tree are too pacific and this leads to many problems.

Take the "ASSIGN" branch for example; this branch should have been an "E" branch for an expression but it was not made this way so that assignments could have special methods related to them to do assignment pacific things. However, the disadvantage of this is that assignments are expressions by nature. By treating assignments differently from expressions, it leads to extra code while working with expressions to deal with assignments.



```
1039
1040        if (left != NULL && op != NULL && right != NULL)
1041        {
1042            if (is_assignment_operator(op->getValue()))
1043            {
1044                // This is an assignment so lets work with it a bit differently
1045                process_assignment(left, right, op);
1046                pop_branch();
1047                exp_root = this->branch;
1048            }
1049            else
1050            {
1051                // Ok all left op and right are present so lets create an expression root branch
1052                exp_root = std::shared_ptr<EBranch>(new EBranch(getCompiler(), op->getValue()));
1053                exp_root->addChild(left);
1054                exp_root->addChild(right);
1055            }
1056        }
1057        else
1058        {
1059            // Ok only left available, set it as the expression root
1060            exp_root = left;
1061        }
```

Figure 25 – Craft compilers parser; expression processing

If you look above at Figure 25 you will see part of one of the expression processing methods inside the Craft compiler parser. On line 1045 the processing of assignments is done if the operator of the expression is an assignment operator. This would have never had to be done if the assignment was treated as an expression but at the same time without doing this methods relating to assignments directly that are inside the *class* that represents an assignment would not exist. This would make code a little harder to write but would result in a more efficient system.

Another problem that is being faced due to this type of design is getting the address of functions. Currently Craft language can support returning the address of variables but not of functions and if this was to be implemented in the future it would be difficult because variable access is handled by its own branch. The parser will see an *identifier* in a return statement as a *variable identifier branch* and this means it will be treated like variable. Now the problem is how does the parser know if this *identifier* is representing a function or a variable. To answer that question, it does not and it would have to do non-parser related things such as keeping track of functions and variables and calculating which branch should be used.

8.11 Future Plans

Some of the future plans for Craft language and Craft compiler is to support the use of switch statements, provide optimization such as *peephole optimization,* create a register system that keeps track of CPU registers so that better code can be generated, and provide a standard library for the language. The development of the standard library has been started and is currently in development.

Another feature that will be added and is also currently in development is Craft compilers own static library format. It is essential that a static library format is supported as currently libraries are represented as normal object files and this is a problem when linking as it means the entire library is included in the executable file when linking statically. With a specially crafted library format parts of the library that are not used can be ignored, this leads to smaller executable files. Another way to get around this is to split a library into different objects and have the user of the library only link with the objects they are using, but then what if the libraries are using any of their own objects then the programmer would have to make sure to link with multiple object files just so the library could support its self.

Another feature that may be added is the ability to have optional arguments for function calls. Finally, the last feature that comes to mind is the ability to have unlimited function arguments. How that could work is when you call a function that supports this feature and you reach the undefined function argument the code generator could just create a *push* instruction and push the value even though it does not know if it's even legal. The receiving function could then use a compiler macro to access these invisible arguments. There is more that would need to happen for this to work correctly such as keeping track of how many undefined function arguments are pushed so that the SP(Stack Pointer) can be restored successfully.

I plan to support this project for as long as I can and I am sure many more features other than the ones described will be developed.

9 CONCLUSIONS

The main objective of this paper was to show the reader what is required for them to be able to write their own compiler, and to help guide them in the right direction. After reading this paper the reader should now have an awareness of lexical analysis where input is converted to tokens for the parser. An understanding of parsing where tokens are used to create an abstract syntax tree which directly represents the input. The reader should also be aware of semantic analysis where validation of the abstract syntax tree takes place. This paper further establishes how compilers typically store information on local scopes which is usually done using the stack frame concept. The reader also should now be aware of code generators where the abstract syntax tree is used to generate code which is commonly assembly language. Assemblers should also now be understood where an assembler converts assembly language into machine code. The reader should also now be aware of optimization, linkers and object formats.

Hopefully the reader has learnt something from this paper and has been inspired and wishes to write a compiler of their own. Compilers are powerful pieces of software and once you learn how to write one the experience gained will be invaluable.

10 Appendix

10.1 Ethics Approval Form

When undertaking a research or enterprise project, Cardiff Met staff and students are obliged to complete this form in order that the ethics implications of that project may be considered.

If the project requires ethics approval from an external agency (e,g., NHS), you will not need to seek additional ethics approval from Cardiff Met. You should however complete Part One of this form and attach a copy of your ethics letter(s) of approval in order that your School has a record of the project.

The document ***Ethics application guidance notes*** will help you complete this form. It is available from the Cardiff Met website. The School or Unit in which you are based may also have produced some guidance documents, please consult your supervisor or School Ethics Coordinator.

Once you have completed the form, sign the declaration and forward to the appropriate person(s) in your School or Unit.

PLEASE NOTE:

Participant recruitment or data collection MUST NOT commence until ethics approval has been obtained.

PART ONE

Name of applicant:	Daniel McCarthy
Supervisor (if student project):	Dr Ana Calderon
School / Unit:	Cardiff Met School of management
Student number (if applicable):	20053963
Programme enrolled on (if applicable):	BSc (Hons) Computing
Project Title:	Developing a general purpose compiler
Expected start date of data collection:	25/10/2016
Approximate duration of data collection:	n/a
Funding Body (if applicable):	n/a

Other researcher(s) working on the	n/a
Will the study involve NHS patients or	No
Will the study involve human samples and/or human cell lines?	No
Does your project fall entirely within one of the following categories:	
Paper based, involving only documents in the public domain	Yes
Laboratory based, not involving human participants or human samples	No

Practice based not involving human participants (eg curatorial, practice audit)	Yes
Compulsory projects in professional practice (eg Initial Teacher Education)	No
A project for which external approval has been obtained (e.g., NHS)	No
If you have answered YES to any of these questions, expand on your answer in the non-technical summary. No further information regarding your project is required. If you have answered NO to all of these questions, you must complete Part 2 of this form	
In no more than 150 words, give a non-technical summary of the project	
This project is a general purpose language compiler for the “Craft” programming language. To compile for one or many different platforms with the possibility for any new platform to be developed with a minimum of 8 bit registers. The project consists of a Lexer to preform lexical analysis, A parser to turn the lexical analysis tokens into an AST(Abstract Syntax Tree), A type checker to check that the AST is valid, A code generator to convert this tree to assembly language, An assembler to assemble this assembly language into machine code. Finally, a linker to convert the objects into an executable.	
DECLARATION: **I confirm that this project conforms with the Cardiff Met Research Governance Framework** **I confirm that I will abide by the Cardiff Met requirements regarding confidentiality and anonymity when conducting this project.** **STUDENTS: I confirm that I will not disclose any information about this project without the prior approval of my supervisor.**	
Signature of the applicant: Daniel McCarthy	Date: 20/10/2016
FOR STUDENT PROJECTS ONLY	
Name of supervisor: Ana Calderon	Date:23/11/2016
Signature of supervisor: Ana Calderon	
Research Ethics Committee use only	
Decision reached:	Project approved [✓ x] Project approved in principle ☐ Decision deferred ☐ Project not approved ☐ Project rejected ☐
Project reference number: **2016D0070**	
Name: Dr Hilary Berger	Date: 23/11/2016
Signature: Dr Hilary Berger	
Details of any conditions upon which approval is dependant: None	

11 REFERENCES

Anon., n.d. *A Brief History of FORTRAN/Fortran.* [Online]
Available at: https://www.ibiblio.org/pub/languages/fortran/ch1-1.html
[Accessed 24 02 2017].

Anon., n.d. *LR Parsing.* [Online]
Available at: http://digital.cs.usu.edu/~allan/Compilers/Notes/LRParsing.pdf
[Accessed 15 03 2017].

Beyer, K. W., 2009. *Grace Hopper and the Invention of the Information Age.* s.l.:Smithsonian Institution.

Bhatt, C. H. & Harshad, B. B., 2013. *Peephole Optimization Technique for analysis and review of.* [Online]
Available at:
https://pdfs.semanticscholar.org/6ce2/e99865863ad262c610b9acb626f4ad24650e.pdf
[Accessed 24 02 2017].

Bryant O'Hallaron, 2010. Computer Systems: A Programmer's Perspective. In: *Computer Systems: A Programmer's Perspective.* s.l.:Pearson; 2 edition, pp. 623-624.

Cantù, M., 2008. Essential Pascal. In: *Essential Pascal.* Piacenza, Italy: s.n., p. 12.

Cataldo, G. D., 2016. *Stack Frames - A Look From Inside.* s.l.:Apress.

Chakraborty, P., 2014. *Fifty years of peephole optimization.* [Online]
Available at: http://www.currentscience.ac.in/Volumes/108/12/2186.pdf
[Accessed 13 04 2017].

Conway, R. & Gries, D., 1975. *An introduction to programming: A structured approach using PL/1 and PL/C-7.* Cambridge, Mass: Winthrop Publishers.

Esik, Z., 1993. *Fundamentals of Computation Theory: 9th International.* s.l.:s.n.

Free Software Foundation, n.d. *The C Preprocessor.* [Online]
Available at: https://gcc.gnu.org/onlinedocs/cpp/index.html#Top
[Accessed 27 01 2017].

Hurdal, R. J., Milne, W. & Zarnke, C., 1972. *WATBOL.* [Online]
Available at: http://csg.uwaterloo.ca/sdtp/watbol.html
[Accessed 13 04 2017].

Johnson, M., 2012. *Miscellaneous Parsing.* [Online]
Available at:
https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/150%20Miscellaneous%20Parsing.pdf
[Accessed 10 02 2017].

Johnson, S. C., 1975. *Yacc: Yet another Compiler-Compiler.* [Online]
Available at: http://www.hpdc.syr.edu/~chapin/cis657/yacc.pdf
[Accessed 11 04 2017].

Knuth, D. E., 1965. *On The Translation Of Languages from Left to Right.* [Online]
Available at: http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf
[Accessed 04 02 2017].

Lavington, S., 2016. *Tony Brooker and the Atlas Compiler Compiler.* [Online]
Available at:
https://elearn.cs.man.ac.uk/~atlas/docs/Tony%20Brooker%20and%20the%20Atlas%20Compiler%20Compiler.pdf
[Accessed 14 02 2017].

Lesk, M. E. & Schmidt, E., 1975. *Lex - A Lexical Analyzer Generator.* [Online]
Available at: http://epaperpress.com/lexandyacc/download/lex.pdf
[Accessed 01 03 2017].

Levine, J., 2009. flex & bison: Text Processing Tools. In: *flex & bison: Text Processing Tools.* s.l.:O'Reilly Media, Inc., p. 9.

McCarthy, D., 2014-5. *Multimedia - Dan's Professional CV - Arrow - Scripting Language,* s.l.: s.n.

Menasce, D., n.d. *The pre-processor.* [Online]
Available at: https://it.wikitolearn.org/C%2B%2B/The_pre-processor
[Accessed 27 01 2017].

Metropolis, N., 1980. *A History of Computing in the Twentieth Century.* s.l.:s.n.

Muchnick, S. S., 1997. Advanced Compiler Design Implementation. In: *Advanced Compiler Design Implementation.* s.l.:s.n., p. 2.

Nilges, E. G., 2004. Build your own .NET language and Compiler. In: M. Smith & K. Winquist, eds. *Build your own .NET language and Compiler.* s.l.:Apress, p. 2.

Ritchie, D. M., n.d. *The Development of the C Language.* [Online]
Available at: http://heim.ifi.uio.no/inf2270/programmer/historien-om-C.pdf
[Accessed 11 04 2017].

S.Adamchik, V., 2009. *Stacks and Queues.* [Online]
Available at: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html
[Accessed 06 04 2017].

Salomon, D., 1992. *Assemblers and Loaders.* [Online]
Available at: https://www.davidsalomon.name/assem.advertis/asl.pdf
[Accessed 08 04 2017].

Siegfried, R. M., 2004. *Compiler Construction - Lecture 8 Semantic Analysis.* [Online]
Available at: http://home.adelphi.edu/~siegfried/cs372/372l8.pdf
[Accessed 18 02 2017].

Stroustrup, B., 2007. *Evolving a language in and for the real world: C++ 1991-2006.* [Online]
Available at: http://www.stroustrup.com/hopl-almost-final.pdf
[Accessed 14 04 2017].

TIS Committee, 1995. *Relocatable Object Module Format.* [Online]
Available at: http://pierrelib.pagesperso-orange.fr/exec_formats/OMF_v1.1.pdf
[Accessed 09 04 2017].

University Of Waterloo, n.d. *Chronology - 1970s: The Evolution of The University of Waterloo Continues.* [Online]
Available at: https://cs.uwaterloo.ca/40th/Chronology/1972.shtml
[Accessed 13 04 2017].

Wikipedians ed., n.d. Compiler Construction. In: *Compiler Construction.* s.l.:s.n.