

School of Electronics and Computer Science
Faculty of Physical Sciences and Engineering
UNIVERSITY OF SOUTHAMPTON

Author: Elliot Alexander
April 30, 2019

Project supervisor: Professor Andrew Brown
Second examiner: Dr Su White

EDSAC Recreation Project: FORTRAN Compiler

A final report submitted for the award of
Master of Computer Science

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

A final report submitted for the award of Master of Computer Science

by Elliot Alexander

The EDSAC recreation project is attempting to reconstruct a working replica of the EDSAC (Electronic delay storage automatic calculator) at the National Museum of Computing located at Bletchley Park[1]; replicating the machines functionality in largely the same manner as the original machine. As a part of this reconstruction, a simpler process is required for writing programs. Original programs for EDSAC were typically hand written, taking simple mnemonic representations of instructions and requiring an intimate knowledge of the machine, its components and instruction set to assemble, a process which quickly became cumbersome[2]. In 1957 the FORTRAN II Programming Language became available, albeit for an entirely different machine. The objective of this project is to construct a cross compiler, capable of taking FORTRAN II code and compiling executable instructions for EDSAC. The result is a large subset of FORTRAN II implemented for EDSAC, encompassing the majority of control flow and arithmetic operations, although lacking some more advanced input / output and data management functionality. The compiler significantly reduces the investment required for writing simple programs, however has a functionality ceiling.

Contents

1	Statement of Originality	1
2	Project Description	3
2.1	Project Description	4
2.2	Project Goals	5
2.3	Project Scope	5
3	Literature Review	7
3.1	EDSAC	7
3.1.1	Initial Orders	7
3.1.2	Main Memory	8
3.1.3	Aside - EDSAC Simulator	9
3.1.4	Instruction Set	10
3.1.4.1	Program Flow	11
3.1.4.2	Architecture of EDSAC	11
3.1.4.3	Comparison Operators	12
3.1.5	I/O on EDSAC	12
3.1.5.1	Output	12
3.1.5.2	Input	12
3.1.6	Data instructions (Pseudo-orders)	13
3.1.7	Bounds on EDSAC data	14
3.1.7.1	EDSAC's data ceiling	14
3.1.7.2	EDSAC's data floor	15
3.1.8	Subroutines on EDSAC	15
3.2	FORTRAN	16
3.2.1	FORTRAN II	16
3.2.2	FORTRAN on EDSAC	16
3.2.3	Specifying FORTRAN	17
4	Proposed Design	19
4.0.1	Planning	19
4.1	Software Plan Outline	19
4.2	User Manual	20
4.3	Compiling FORTRAN II	20
4.4	Performance of FORTRAN II on EDSAC	21
4.5	Examination of specific elements	21
4.5.1	Outputted Program Structure	21

4.5.1.1	Control Combinations	22
4.5.2	Arithmetic Values	22
4.5.3	Program libraries	23
4.5.4	Symbol Table	25
4.5.4.1	The relation of Three Op Code	25
4.5.4.2	Function Symbol Tables	26
4.5.5	DO Loops	26
5	Justification for approach	31
5.1	Design considerations	31
5.2	Software considerations	32
5.2.1	Platform agnosticism	32
5.2.2	Included tools and libraries	32
5.2.2.1	Boost	32
5.2.2.2	Flex and Bison	32
5.2.2.3	cxxopts.h	33
6	Testing	35
6.1	Unit testing	35
6.1.1	Tokenization Testing	35
6.1.2	Statement Testing	35
6.2	Integration Testing	36
7	Future Work	37
7.1	Common Blocks	37
7.2	Floating Points	37
7.3	I/O	37
7.4	Division	37
8	Project Outcome	39
8.1	Project Documentation	39
8.2	Aside - EDSAC disassembler	39
8.3	The EDSAC Compiler	40
8.3.1	Usefulness of the EDSAC compiler	40
8.3.1.1	Example Programs	40
8.3.2	Benefits of the Compiler	40
8.3.3	Incompatibilities and performance	41
9	Critical Evaluation	43
9.1	Completion of Goals	43
9.2	Scope of implementation	43
9.3	Quality of implementation	44
9.4	Quality of documentation	44
9.5	Project planning	44
9.6	Conclusion	45
	Bibliography	47

Appendices	51
Appendix A. Word Count Considerations	53
Appendix B. Project Planning documentation	53
Appendix C. EDSAC Disassembler Output	58
Appendix D. Software Versions	59
Appendix E. License Statement	59
Appendix F. EDSAC Character Output Codes	60
Appendix G. DO loop example	61
Appendix H. Subroutine jump example	63
Appendix I. Arithmetic Parser Example	64
Appendix J. Testing examples	65
I.1 Tokenization testing	65
Appendix K. Example programs	68
K.1 Cubes	68
K.2 IF Statements	69
K.3 Descending Integers	71
K.4 Sum of cubes	72
K.5 Sum of natural numbers up to N	74
K.6 Subroutine sum of Naturals up to N	75
K.7 Calculation of Factors	77
K.8 Prime Numbers	79
Appendix L. Design Archive Index	83
Appendix M. Original Project Brief	83

List of Figures

3.1	Sir Maurice Wilkes' Initial 1950 Specification of the EDSAC order codes. Reproduced from [3]	9
3.2	The main screen of Martin Campbell-Kelly's EDSAC Simulator. Reproduced from [4]	10
3.3	The makeup of an EDSAC instruction. Reproduced from [4]	11
3.4	The makeup of an EDSAC Data value, both short and long. Adapted from [5]	13
4.1	The parse tree for the expression $\text{PRINT } X + 1 - 2 * Y$	23
4.2	The structure of a program outputted by the compiler.	24
4.3	Self-modifying function example, which injects a return value at the end of the function.	25
4.4	A simple illustration of the concept of mappings	27
4.5	The data structures of the Symbol Table and the interfacing controller class	28
4.6	An example of Overlapping DO loops and their implementation with a First In First Out queue.	29
B.1	Gantt Chart Key	53
B.2	Project time plan	55
B.3	Project time plan (Continued)	56
B.4	Development detailed time plan	57

1. Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

I have acknowledged all sources, and identified any content taken from elsewhere. I have used resources produced by someone else. The source file `cxxopts.h` is included in the design archive, licensed under an open MIT license. The relevant license statement is included in the appendix of this document, and the author is credited both within this document and within the file. I did all the work myself, or with my allocated group, and have not helped anyone else. The material in the report is genuine, and I have included all my data/code/designs. I have not submitted any part of this work for another assessment. My work did not involve human participants, their cells or data, or animals.

2. Project Description

Based at the National Museum of Computing at Bletchley Park, the Electronic Delay Storage Automatic Calculator (EDSAC) replica project is attempting to reconstruct a functional replica of the original machine constructed at the Cambridge Computer Laboratory[1] [3]. EDSAC is widely considered one of the world's first programmable computers, and the project is aspiring to reconstruct the machine back to its original state of completion in 1949 [6]. The machine was one of the most advanced in the world at the time, and took almost three years to construct [4]. However, running programs on EDSAC is a cumbersome process, involving an intimate knowledge of the machine and the instruction set, with programs often being found to be buggy or broken at runtime and with little or no way of effectively validating programs before execution [7]. While the main memory of EDSAC (known as the 'main store') was examinable, arithmetic or semantic errors were notoriously difficult to diagnose [4]. In addition to this, several complex and long since archaic techniques were used to construct programs for the EDSAC, the largest of which often being a major undertaking to write and test. A notable example of this is OXO, considered by some to be the first video game, an implementation of noughts and crosses completed by Alexander Sandy Douglas as part of his PhD thesis [8]. The project earned Douglas the worlds first PhD in Computer Science. EDSAC was one of the first machines to allow simple mnemonics in the place of instructions, although this did little to alleviate the problem of writing programs. Additionally, EDSAC had support for 'open' and 'closed' subroutines, with open subroutines functioning as an inserted section of tape to be integrated into a main program, and closed subroutines able to be accessed through a change in control flow before returning control to the main program [9]. While this was extraordinary at the time, the complex semantics of the instruction set present a serious challenge to anyone attempting to write modern EDSAC programs. With regard to the reconstruction project, this difficulty has remained, despite the existence of various alleviating tools such as Martin Campbell-Kelly's x86 EDSAC simulator [10]. It is clear a new solution for effectively writing EDSAC programs is needed.

2.1 Project Description

The immediate solution for this problem, as is common on embedded systems today, is a cross compiler capable of compiling a higher level language into a series of executable instructions for the EDSAC machine. This would allow a programmer to write code within the comforts of a modern system, while appreciating the benefits of pre-runtime syntactic and semantic analysis a compiler brings. A compiler can significantly reduce the likelihood of small errors within programs that while easily fixed becoming costly and time consuming to deal with, while dramatically lowering the barrier of entry to EDSAC.

A compiler enables other unique benefits. During the original lifespan, EDSAC was reliant on a variety of complex techniques to keep programs maintainable. Examples include the propensity of contemporary programmers to use self modifying code, a concept long since deprecated in the mainstream of modern computer science due to the undue complexity it introduces[11]. However, this technique was considered 'an essential part of the architecture' in terms of implementing closed subroutines, which relied heavily on modifying return instructions prior to executing a subroutine [9] [11]. Another example of this is the problem of direct addressing, with the machine's instruction set relying on programmers using simple offsets, or directly addressing all instructions and data values. This introduced a serious maintainability problem, as a programmer realising an earlier mistake in their program and remedially inserting an inserting or removing an instruction would then be required to modify every proceeding / preceding address in the program to ensure correct addressing of data values and jump points [4]. A compiler abstracts this process from the user, allowing the user to write and modify complex programs without the need for a consideration of the addressing of individual memory locations [4].

This project entails a cross compiler, written in C++, capable of interpreting FORTRAN 2 code and assembling a series of EDSAC instructions. The final cross compiler should run from an executable and be capable of taking several files as inputs, outputting the appropriately linked code to be loaded onto the EDSAC machine. The compiler should have configuration options for both EDSAC and debugging output. An essential part of all compilers is providing meaningful and useful error messages and warnings to the end user, allowing the user granular control to achieve their goals without the compiler acting as a hindrance; simultaneously offering a barrier between the user and obviously destructive actions. This is another key goal of the project, helping to ensure that the code produced from the compiler is useful and not prone to fault. Further to this, documentation of the project as a whole is essential, as is maintainable and potentially adaptable code.

2.2 Project Goals

The project goals can be summarised as follows:

- The compiler should be capable of interpreting FORTRAN II code and output a series of instructions capable of running on the EDSAC machine.
- The compiler should be platform agnostic, running on a modern machines as an executable, supporting recent versions of all major operating systems
- The compiler should maintain a high standard of documentation, and ensure the code base is maintainable.
- The project should remain accurate and durable, with a focus on informative error and warning messages for the user.
- The project should enable meaningful debugging of programs, offering (where appropriate) semantic warnings and a high level of output to the user.

2.3 Project Scope

The scope of the project is reasonably well constrained, with accurate, informative compiler errors and documentation being the core goals outside of a functional compiler. The focus of the project will be on quality, with the real world value of additional features being carefully considered before any attempt is made to implement them, particularly when additional features may arrive at the expense of quality assurance elsewhere. The ultimate goal of the project will be the implementation of quality core functionality (i.e. a working compiler). Should the project progress ahead of schedule, a focus will remain on improving and expanding stability, optimization, semantic analysis, user warnings and error messages.

3. Literature Review

3.1 EDSAC

EDSAC was initially conceived at the Cambridge Computing Laboratory (known at the time as the Mathematical Laboratory) in 1947, operating for eleven years before being succeeded by EDSAC 2 in 1958 [2][3][6]. The machine was the brainchild of Sir Maurice Wilkes, an instrumental figure in the development of Computer Science as well as EDSAC’s development, having been associated with the University of Cambridge Computing Laboratory since conception in 1937[12][13].

EDSAC represents the world’s first stored program computer operating in regular service, providing a significant scientific computing capability to the University of Cambridge; with a thousand-fold increase in computing power compared to the mechanical calculators which preceded it [6] [2]. The original machine contained only 512 individual bits of usable storage, later expanded to 1024, a portion of which was consumed by the ‘Initial Orders’ [14] [6]. EDSAC was conceived as an extension on the mechanical calculators of the day, which while able to perform a series of operations, those operations would need to be specified in real time by a user. The machine was intended to “[enable] the process of mechanization to be carried a stage further, in that it can perform automatically a series of arithmetic operations according to a prearranged plan or ‘programme’” [15].

3.1.1 Initial Orders

Initial Orders is an EDSAC program functioning as a simple bootloader, and represents one of the earliest notions of programmatic abstraction by allowing a machine to intake a program prior to execution, rather than just directly consuming machine code pre-loaded into memory [14]. Initial Orders was quickly replaced with Initial Orders 2, an expansion on the simple assembler. The EDSAC replica project will only aim to support Initial Orders 2, with the original Initial Orders being substantially less powerful and relatively short lived. The machine’s conventional orders were inputted on five hole telegraphic punched tape, a format which caused great difficulty, as the accuracy of the punched

holes at the time was questionable [6] [3]. It was noted by John Lindley the “incredible difficulty [they] ever had to produce a single correct piece of paper tape with the crude and unreliable home-made punching, printing and verifying gear available in the 1950’s.” [7]. The replica project will aim to continue using punched tape where possible, albeit with some modern comforts of accuracy and automated generation. Initial Orders 2 was constructed by David Wheeler, a PhD student studying under Maurice Wilkes [16]. The original initial orders occupied the first 32 memory locations on the machine, with the later Initial Orders 2 occupying 56 (an additional eight addresses were reserved for global variables). The Initial Orders were placed in memory as the machine started, prior to commencing execution from location zero[6]. The problem of loading these orders into memory was solved through the use of a series of uni-selector switches[17] [16]. This allowed the bit pattern for each instruction in the Initial Orders to be loaded directly into memory prior to the start of execution, where the machine would commence from zero. This technique is long irrelevant for modern systems, but was a practical solution considering the minimal size of even the larger Initial Orders 2. Initial Orders 2 was essential for interpreting ‘pseudo-orders’ into usable instruction values or ‘machine code’ for the EDSAC, consuming each loaded tape instruction one value at a time and converting it into a properly assembled bit pattern before placing it consecutively in the machines main memory, normally beginning at address 56 or 64 [16][17] [4]. This facility also allowed the inclusion of constant offsets, both standard and user defined. For example, the inclusion of θ (written as @). This character code could be appended to the end of the address of an instruction ingested by Initial Orders 2, offsetting the address by the value of the current instruction [17] [4]. The value of this offset was defined by the control code TnK , where n is the value of the programs starting location. Typically, this value was set to 64, as some subroutines may use the addresses immediately following the Initial Orders [17]. This was useful for referencing a specific line instruction in a program, even when the location of the program might have changed in memory.

3.1.2 Main Memory

The ‘main store’, or the machines main memory, was constructed of 32 tanks, each containing a further 32 memory locations consisting of a 17 bit memory space. This memory space included a reserved sign bit for data values, and an unused bit for instructions [6] [1]. This unused bit was later used to add a rudimentary index register, something EDSAC initially lacked [17]. The length of these words could be expanded to 35 bits, with a central padding bit and a short / long bit set at the end of each instruction. [6]. A long value effectively shifted the decimal point of the value to the middle of the straddled address, with the lower value representing the integer and the higher address representing the fractional value of the float [6]. EDSAC, as was common in the era, was centered around the concept of an accumulator, a single larger register associated with an adder[9] [17]. The accumulator was capable of holding 71 bits, an important

EDSAC Order Code

Order	Explanation
<i>A n</i>	Add the number in storage location <i>n</i> into the accumulator.
<i>S n</i>	Subtract the number in storage location <i>n</i> from the accumulator.
<i>H n</i>	Transfer the number in storage location <i>n</i> into the multiplier register.
<i>V n</i>	Multiply the number in storage location <i>n</i> by the number in the multiplier register and add into the accumulator.
<i>N n</i>	Multiply the number in storage location <i>n</i> by the number in the multiplier register and subtract from the contents of the accumulator.
<i>T n</i>	Transfer the contents of the accumulator to storage location <i>n</i> , and clear the accumulator.
<i>U n</i>	Transfer the contents of the accumulator to storage location <i>n</i> , and do not clear the accumulator.
<i>C n</i>	Collate the number in storage location <i>n</i> with the number in the multiplier register, i.e., add a 1 into the accumulator in digital positions where both numbers have a 1 and a 0 in other digital positions.
<i>R 2ⁿ⁻²</i>	Shift the number in the accumulator <i>n</i> places to the right, i.e., multiply it by 2^{-n} .
<i>L 2ⁿ⁻²</i>	Shift the number in the accumulator <i>n</i> places to the left, i.e., multiply it by 2^n .
<i>E n</i>	If the number in the accumulator is greater than or equal to zero, execute next the order which stands in storage location <i>n</i> ; otherwise, proceed serially.
<i>G n</i>	If the number in the accumulator is less than zero, execute next the order which stands in storage location <i>n</i> ; otherwise, proceed serially.
<i>I n</i>	Read the next row of holes on the tape, and place the resulting 5 digits in the least significant places of storage location <i>n</i> .
<i>O n</i>	Print the character now set up on the teleprinter, and set up on the teleprinter the character represented by the five most significant digits in storage location <i>n</i> .
<i>F n</i>	Place the five digits which represent the character next to be printed by the teleprinter in the five most significant places of storage location <i>n</i> .
<i>Y</i>	Round off the number in the accumulator to 34 binary digits.
<i>Z</i>	Stop the machine, and ring the warning bell.

Figure 3.1: Sir Maurice Wilkes' Initial 1950 Specification of the EDSAC order codes. Reproduced from [3]

note as it allowed two full length numbers to be multiplied together and added into the accumulator without loss of digits [18]. From here, values could be outputted across four memory addresses, or shifted by an appropriate number of bits to save the output in either a short or long address value. The multiplier was 35 bits long. The accumulator could be shifted in place, however care was needed to ensure that no overflow occurred [18]. EDSAC featured no warning for the programmer when an overflow occurred, an oversight which remained a problem until the addition of an overflow alarm later in the machine's lifespan [3].

3.1.3 Aside - EDSAC Simulator

An extremely useful tool for developing software directly for EDSAC is Martin Campbell-Kelly's EDSAC simulator, which allows programs to be run under both Initial Orders 1 and 2, in addition to supplying original and reconstructed versions of a variety of

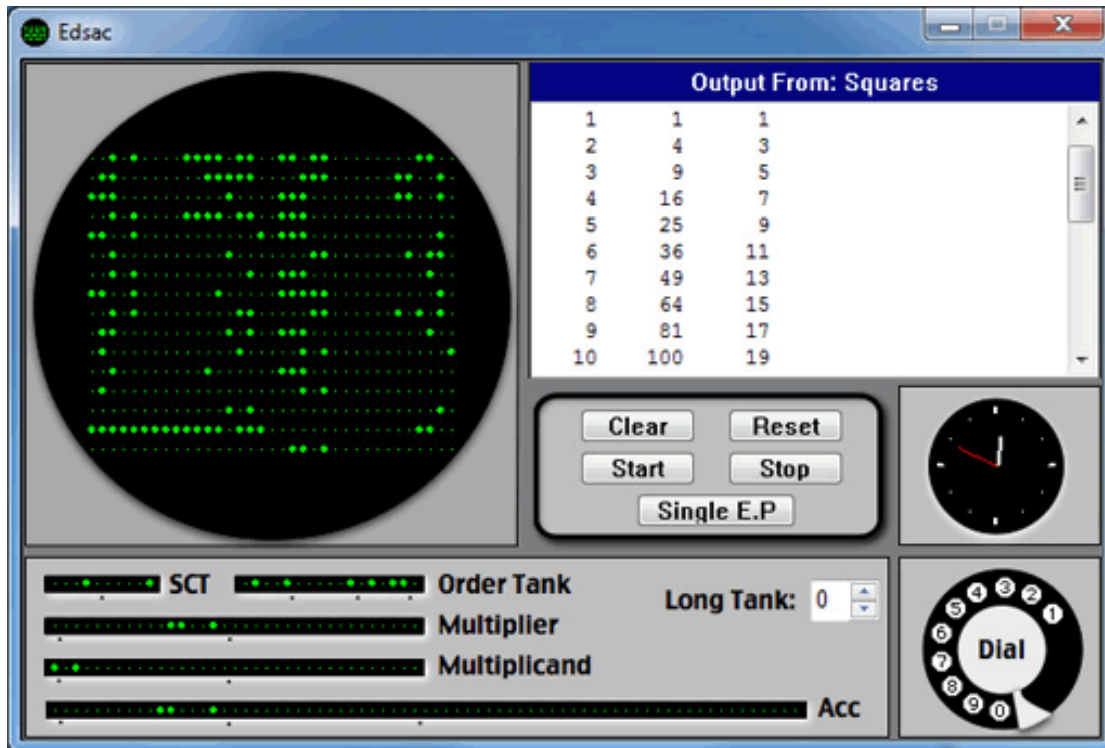


Figure 3.2: The main screen of Martin Campbell-Kelly’s EDSAC Simulator. Reproduced from [4]

subroutines for EDSAC. The simulator also allows step-through execution, allowing easier debugging of typically difficult semantic or numerical errors, in addition to a live view of the states of the multiplier, accumulator and the current operation [4].

3.1.4 Instruction Set

Shortly after EDSAC’s conception, the development of simple mnemonics allowed for instructions to be referenced by a single representative character. These mnemonics were represented as a five bit ‘character code’, corresponding to a specific character, in the topmost five bits of a word in memory. Each instruction was also associated with an address and indicator as to the operand size [6] [3]. Figure 3.2 includes a list of EDSAC instructions, as specified in Wilkes’ 1950 paper entitled ‘The EDSAC’, and the makeup of an instruction is illustrated in Figure 3.3 [3].

The following section will briefly highlight some of the difficulties in programming with EDSAC, as well as warning the reader to some of the potential pitfalls and specifics which remain both unintuitive and poorly documented.

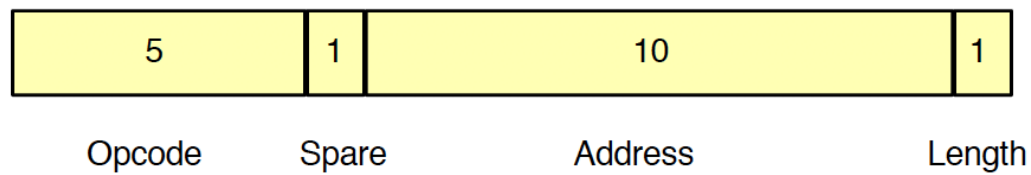


Figure 3.3: The makeup of an EDSAC instruction. Reproduced from [4]

3.1.4.1 Program Flow

The instruction set contained only 17 operations (see Figure 3.1), and was extremely limited. There was no unconditional GOTO, so any attempt to modify program flow programatically required either a) a clearing and resetting value of the accumulator or b) prior knowledge of the contents of the accumulator. Control over program flow was provided through two operators, ‘E’ and ‘G’, representing ‘Jump to address N if the accumulator is positive’ and ‘Jump to address N if the accumulator is negative’ respectively. These instructions work purely on the basis of the sign of the accumulator, hence the former (‘E’), will trigger if the value of the accumulator is zero [18] [9]. This is achieved most easily by clearing the accumulator into some unused memory address, before jumping with the E instruction. An empty accumulator has zeroed sign bit, hence the sign of the accumulator is positive and the jump completes. The downside of this technique is the value in the accumulator is not preserved.

[n]	TnF
[n+1]	ExF

Listing 3.1: An example of a simple jump instruction on EDSAC, where n is a discarded memory address and x is the jump location.

3.1.4.2 Architecture of EDSAC

The use of an instruction as it’s own data value serves as a good example of a concept which is essential to grasp when reasoning about and constructing values for EDSAC, the modern concept of a modern Von Neumann architecture. EDSAC was originally conceived as a spiritual successor to ENIAC, and was constructed at the forefront of ideas about a combined store for programs and data [19] [20]. EDSAC hence remains a notable example of a computer which not only shares a bus for both program and data values, but makes no attempt to distinguish between the two. Instructions and data are ultimately the same values within EDSAC, merely interpreted in a different manner[21].

3.1.4.3 Comparison Operators

Another notable exclusion was the lack of a direct comparison operation. This made comparisons of two values expensive, particularly if they were long values. Division was also not provided, and was accomplished through the use of a subroutine (D6), and had no native support in the instruction set. Similar subroutines were provided (either restored through the efforts of Martin Campbell-Kelly’s simulator, or as original routines) for exponentiation, resolving the root of a square, as well as a variety of input and output routines.

Determining if a value is zero is also problematic, and hence these kinds of operations were generally avoided.

3.1.5 I/O on EDSAC

3.1.5.1 Output

EDSAC featured only one output instruction, **O**, which printed the most significant five bits of the provided address to the teleprinter. The teleprinter has two modes, ‘Figure Shift’ and ‘Letter Shift’. These can be cycled by printing the letter and figure shift characters in Martin Campbell-Kelly’s simulator, although this traditionally would’ve been a hardware switch on the teleprinter [17]. These letter and figure shift characters are Erase, represented as ‘*’ and π , represented as ‘#’. Hence, an example of toggling the teleprinter into ‘Figure Shift’ mode is as follows:

```
[0] 01F
[1] *F

Note that:
[1] *F => 01111 0 00000 00000 0
Which outputs the character code for 01111, the most significant five bits.
```

Listing 3.2: An example of the EDSAC instructions required to toggle the EDSAC simulator into ‘Figure Shift’ mode.

When in Letter Shift mode, the character value of a bit pattern would be printed to the teleprinter. In Figure Shift mode, the numerical value would be printed. A full list of these characters is included in Appendix F.

3.1.5.2 Input

Input is achieved through the **I** instruction, which takes as an input the next row of holes on the provided input tape and places them into the least five significant locations

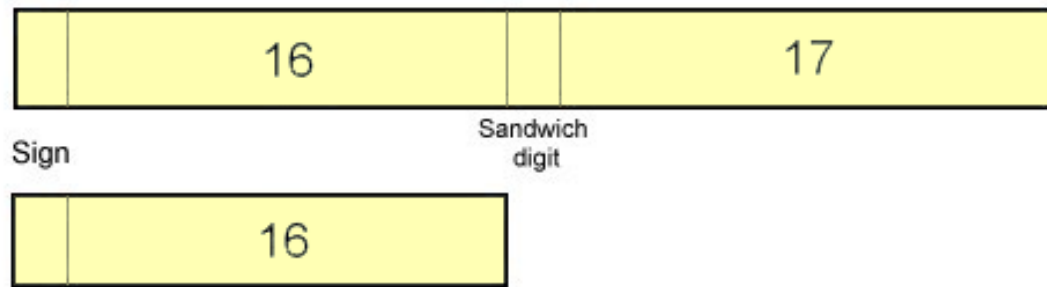


Figure 3.4: The makeup of an EDSAC Data value, both short and long. Adapted from [5]

of address N. Practical input was relatively difficult to achieve, however some useful subroutines exist to aid the process, namely M20, which takes an integer input places it as a data value in a specified address location. The use of such routines in conjunction with more complex I/O is most visible (and useful) in some of the postmortem routines, for example PM5, which dumps the entire program memory from a specified address.

3.1.6 Data instructions (Pseudo-orders)

Data values were loaded directly from memory as a seventeen bit pattern, a nuance which makes the process of loading long data values into memory difficult. The convention for loading memory addresses involved the concept of pseudo-orders', orders which would then be interpreted into a bit pattern by the Initial Orders. While EDSAC allowed an instruction address of 10 bits - enough to reference every location in the store, the range of data values representable varied from -65536 to $+65535$, or 2^{17} values. This was achieved in practice through the use of character codes to assemble the correct bit value for the entire address. The general algorithm for short values is as follows.

```

if ( n < (211 * 2) )
    if ( n mod 2 == 0 ):
        data_val = P(n / 2)F
    else:
        data_val = P(n/2)D
else:
    // See extended data values

```

Listing 3.3: The algorithm for generating data values.

For short integers (i.e. *when* $n < 2^{11}$) P may be specified as an instruction operator, P representing a empty 00000 in the topmost five bits. Therefore, for an data value of 10, the mnemonic instruction would be (note that F represents a short bit value):

```
P5F -> 00000 0 000000101 0 = 10
```

Note that the short / long bit is used as a part of the data value, hence the largest bit value which can be assembled without the user of character codes is 2^{11} .

If a larger data value is required, the top bits of the field can be modified with the preceding character code. The sign of the value is determined by the first bit, and instructions can be either 17 or 35 bits. A common pitfall here is the inclusion of a sandwich bit between the first and second data values for long data values. This sandwich bit is unused for instructions, and was used later in EDSAC's lifetime as an index register, however is used for defining data values [9] [2]. This sandwich bit is best indicated in Figure 3.4

Therefore, in order to insert a number larger than the 2048 limit of an 11 bit address, an different character code is used. A complete list of character codes (including the essential figure and letter shift codes) is included in Appendix F. This is best demonstrated by representing both the highest and lowest possible numbers. It is important to note that while there is an unused bit present between the opcode and address for EDASC instructions, this bit becomes a valid part of the data value and is **not** ignored. In the following explanatory example, both the highest and lowest possible values are represented as single addresses.

3.1.7 Bounds on EDSAC data

3.1.7.1 EDSAC's data ceiling

In order to represent the highest positive value, the sign bit needs to be zero. Hence, the maximum possible representation of a positive number ingested in 17 bits, with one sign bit, is 01111 1 11111 11111 1. Therefore, the highest decimal value possible inside the top five bits represented by the character code is 01111, the character code for which is * (representing a character erase inside EDSAC). When representing data

values, the unused middle in position six bit from operations is now considered, hence the highest possible value represented within 11 bits is $2^{11} - 1 = 2047$. Finally, we need to set the final short / long bit to 1. This gives a value of:

```
*2047D => 01111 1 11111 11111 1 => 65535
```

Broken down, this is:

```
*      => 01111
2047   => 1 11111 11111
D      => 1
```

The final F / D flag represents a short / long address with regard to instructions, however is merely a placeholder for the value of the final bit in the context of data instructions. A complete list of character codes and their bit values is included in Appendix F.

3.1.7.2 EDSAC's data floor

In order to represent the lowest possible value in EDSAC, an inverted value of the highest possible value is required. This includes the sign bit. Therefore, the character code for the top five bits of the data entry must represent 10000, with the sign bit being the leftmost place. This character code is a blank space within EDSAC, however is represented as '.'. The remainder of the address must be zeroed out, and this can be represented by an address value of 0, and a short/long bit of F (0). Hence, with '.' => 10000, and the remainder of the address zeroed out:

```
.0F => 10000 0 00000 00000 0 => -65536
```

It should be noted that this is only for whole numbers. Using a double address, a range of floating points can also be expressed. A full list of character codes is included in Appendix F.

3.1.8 Subroutines on EDSAC

Subroutines, or 'library routines' as they were known, were referred to as either open or closed (in a convention not dissimilar to the concept of Open and Closed subroutines in FORTRAN) [22] [9]. Open subroutines were designed to be inserted directly into the program text, whereas closed subroutines commonly used a pre-set parameter (sometimes the value in the accumulator). This pre-set parameter represents a memory address of the location which the subroutine will jump back to once execution has completed. Subroutines often intake a variety of argument values, usually set within the address space of the initial orders [10] [23]. A complete list of subroutines, both original and restored, can be found in the accompanying documentation for Martin Campbell-Kelly's EDSAC simulator [23]. Subroutines were generally referred to by a type and

an integer, for example **P6** represents print routine six, and **PM5** represents postmortem routine 5.

3.2 FORTRAN

The FORTRAN language was introduced by IBM in 1954 in accompaniment to the IBM 704 Data Processing machine [24] [25]. Originally developed by a team led by John Backus, FORTRAN I emerged from the idea that an extensive series of operations resulting in a single, simple arithmetic operation should be expressed as a single command which when passed to the machine, would be interpreted as a set of predefined instructions; with the product emerging three years later as the worlds first high-level language [26] [25] [27]. Originally an acronym for FORMula TRANslation, FORTRAN allowed programmers to express complex mathematical notations without the fear of complex and often difficult to diagnose numerical or semantic errors in their programs, which made the process of building executable programs costly and meticulous. As FORTRAN developed further, different groups made a variety of modifications to the original specifications, and the following versions of FORTRAN quickly became a complicated web of small modifications to the language for machine or task specific purposes. This led to the first official specification of FORTRAN by the American Standards Organisation in 1966, a version of the language now colloquially known as FORTRAN '66 [27]. The introduction of FORTRAN was an important precursor for modern compiler theory, and offered contemporary programmers a significant reduction in the time investment required to write programs for machines of the day [28].

3.2.1 FORTRAN II

FORTRAN was quickly followed by FORTRAN II in 1958, a revised version which offered an expanded feature set including the facility for functions and subroutines, as well as the COMMON block [28]. FORTRAN II introduced six new statements, and was considered 'compatible' with the original FORTRAN, as all the original statements remained [28]. In addition, FORTRAN II introduced language level support for subroutines and functions.

3.2.2 FORTRAN on EDSAC

FORTRAN has been expanded massively over the last 60 years, and remains in use in an almost unrecognisable form, although modern compilers do still support compiling FORTRAN I and II code [29]. Despite FORTRAN's history with machines of this particular epoch, FORTRAN code never ran on EDSAC. The construction of programs for

EDSAC relied on the talents of programmers writing by hand what would be considered in a modern context as assembly language. Early versions of FORTRAN represented one of the first ventures into programming language design, and as such some of the design decisions are considered entirely redundant by modern standards, for example the exclusion of whitespace as a significant character [28] [29].

3.2.3 Specifying FORTRAN

FORTRAN on EDSAC presents a significant challenge, with the language initially having been conceived for an IBM machine which appeared in the wake of EDSAC, there is no consideration for the feasibility of various instructions and operations on EDSAC. This has left some features troublesome to implement, and others significantly more expensive than previously conceived. Additionally, the lack of a true language specification and range of machine specific functionalities in early versions of FORTRAN makes identifying the syntax and semantics of FORTRAN II a task in itself.

A notable example is division and alternative GOTO statements. Heavy use GOTO statements in a FORTRAN program is difficult to port across to EDSAC, mainly due to the difficulties with the lack of an unconditional GOTO instruction and blighted comparison operators [29] [9]. In addition, the original specifications for the language included support for certain hardware features which are simply not present on EDSAC, for example the ability to determine if an accumulator or quotient overflow has occurred, neither of which have hardware support in EDSAC [29] [28].

This lack of a clear specification also extends to an inability to find example programs, particularly those written in versions of FORTRAN prior to FORTRAN '66 [24] [27].

4. Proposed Design

4.0.1 Planning

As with almost all large software projects, the most important elements come far before any code is written, with documentation and planning of the finished product being far more important in ensuring the functionality, features and polish of the finished product are as desired. Because of this, a large portion of the early stages of the project were devoted to planning and consideration of the challenges, such that a detailed software plan can be produced prior to any code being written. This plan should be sufficient to enable someone new to the project to reasonably recreate the code base's functionality. The aim of the software plan is as follows:

1. The software plan should reflect the finished software as closely as possible- i.e. consideration for the challenges and their solutions should be completed long before any code is written.
2. Be as widely encompassing as possible - another adequately informed person should be able to reconstruct the software from scratch using the provided documentation. i.e. The software plan should encompass the full breadth of the software.
3. Be as detailed as reasonably possible - another adequately informed person should be able to reconstruct the software's functionality in detail.
4. Ensure the software is well defined and designed, containing the scope and organization of the software to ensure readability, elegance and structure. This will later enable extensibility, maintenance and potential modification. It is unlikely that the compiler will encompass all possible functionality, and may be updated and maintained at a later date. Hence, this is an important consideration.

4.1 Software Plan Outline

The software plan consists of several key parts, namely:

- An annotated flow chart, showing each process of the compilation in detail, in addition to offering supporting resources and notes on the higher level functionality of the compiler.
- A large data structure diagram, showing each data structure used within the compiler, the accompanying elements, and the position in the larger structure.
- A volume of supporting resources, including state transition diagrams and flow charts of particular algorithms, in addition to documentation relating to time and feature prioritisation, as well as contingency planning for unimplemented features.

As mentioned previously, the inclusion of the software plan would be impractical, however it is included as a collection of documents in the Design Archive. This also includes time planning, as well as feature prioritisation and contingency planning.

4.2 User Manual

In reality, any practical attempt to write code using the compiler will be reliant on the Users Manual. This is included as an independent document, containing a language specification, semantic information and a how to guide for general use of the compiler.

4.3 Compiling FORTRAN II

Whitespace is not considered relevant, in an early quirk of FORTRAN II. Hence the statements `ASSIGN X TO 10` and `ASSIGNXT010` are functionally identical. This makes FORTRAN extremely difficult to parse using conventional parsing techniques. Consider the following example of a FORTRAN DO Loop.

```
DO100F = 1,10
```

A modern approach to Left-Right tokenisation would struggle to differentiate the statement from a variable declaration, the statement is not uniquely identifiable from a variable declaration until the comma indicated is seen by the tokeniser.

```
DO100F = 1,10
    ^ At this point, the statement becomes uniquely identifiable
```

A modern tokenizer would likely interpret the above as:

```
<Variable Name> <Equals> <Integer> <Comma> <Integer>
```

This example captures a microcosm of a problem which exists throughout the language [30].

4.4 Performance of FORTRAN II on EDSAC

While it is certainly possible to compile FORTRAN II code for EDSAC, the semantics of particular FORTRAN statements are not suited to the capabilities of EDSAC. For example, statements requiring repeated comparisons in particular are rather greedy when run on EDSAC. An example of this is a COMPUTED GOTO:

```
GOTO X, (5, 10, 15)
```

Listing 4.1: The semantics of a COMPUTED GOTO statement in FORTRAN II

The semantics of this statement are:

```
if ( x < 0 )  
    -> Goto address 5  
if ( x == 0 )  
    -> Goto address 10  
if ( x > 0 )  
    -> Goto address 15
```

This particular instruction is expensive, primarily due to the lack of a comparison operator, in addition to the ability to compare instructions to zero. EDSAC features a positive and negative GOTO instruction, **E** and **G** respectively, however this only checks the sign bit of the accumulator, making differentiating between zero and any other positive integer inherently expensive.

4.5 Examination of specific elements

Some elements of the software plan are briefly discussed in more detail in the following sections, including the overall structure of the outputted code, and a brief examination of the structure of the symbol table and line mapping data structures. Additionally, a brief description of the problems of and solutions for the implementation of DO loops.

4.5.1 Outputted Program Structure

The outputted structure of compiled code is not immediately obvious, and is worth briefly explaining. In this regard, an understanding of the outputted code, as well as how and where it is loaded into on the machine, may be useful for anyone attempting to diagnose semantic bugs with compiled code generated by the compiler. The first 56 addresses of all programs are occupied by Initial Orders 2, with the compiled EDSAC code then filling the rest of the address space. Output starts with bundled subroutines and libraries, then user defined subroutines and libraries, followed by User defined functions and subroutines. The main program is then included, with the symbol table filling the end of the outputted code.

4.5.1.1 Control Combinations

The compiler only utilises basic control codes, although more are available [4]. The most important control combination is **TnK**, where n is the program load point. Therefore, this address is always the first address of the generated code from the EDSAC compiler.

Following this, the control code **GKTZ** is used to set the @ parameter to the current instruction and then restore it. This character code is a variable set within Initial Orders, which when included in an instruction address appends the current value to the address, functioning as a rudimentary offset for memory addressing. This also allows the final control combination to return program flow to the start once execution has completed.

The control combination **EZPF** is used to return the program to the @ parameter, and indicate a stop. This control combination is appended to the end of the program body of every program generated by the EDSAC compiler. Note that this instruction is appended prior to the symbol table output. Allowing control flow to run into data values is potentially dangerous, as long values may be treated as instructions.

4.5.2 Arithmetic Values

The compiler handles Arithmetic expressions. Arithmetic expressions any valid language element in which arithmetic operations can be performed (i.e. **PRINT X + 5**) as separate to the main body of program. These values are not manipulated directly, and instead dropped into a subsection of the compiler which remains entirely responsible for calculating these values. This subsection, known as the Arithmetic Parser, takes advantage of more traditional lexing and parsing libraries (See ‘Design Considerations’). This sub-parser takes in an arithmetic string, builds a parse tree of nodes of the arithmetic statement, before returning two values. The first is a list of three op code instructions required to perform the computation necessary to calculate the value, and the second is a Symbol Table reference containing the location of the final value. This three op code can then be injected into the original statement, and the value of the output used interchangeably with any other data value in the compiler. The benefit of this is that complex arithmetic statements are processed entirely anonymously. This functionality is encapsulated in a single function call which invokes the sub-parser, taking only a string as input, and returns a computed arithmetic string. This sub-parser uses the same symbol table and line mapping functionality, while functioning effectively in parallel to the main compiler. An annotated example of a simple program utilising the arithmetic parser is included in I.

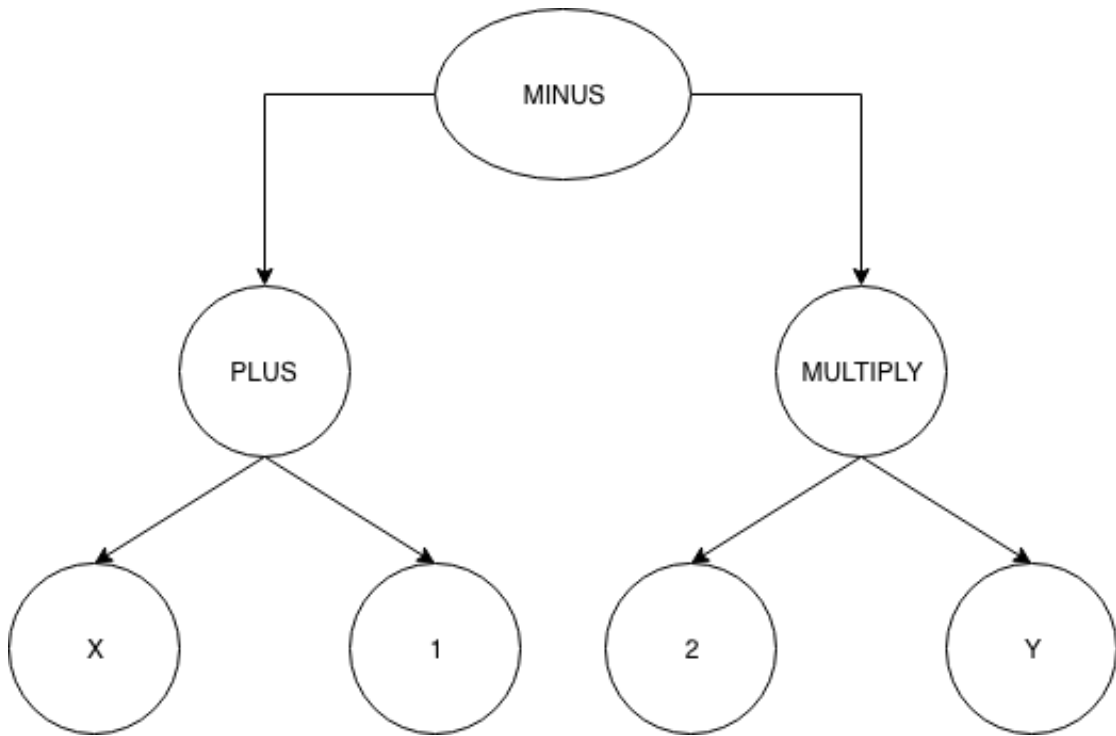


Figure 4.1: The parse tree for the expression **PRINT** $X + 1 - 2 * Y$

4.5.3 Program libraries

Program libraries are included from the compiler, as it is essential to offset memory references by the exact size of libraries provided. They are included as a string, and either enabled via a command line argument or an instruction which requires them.

The implementation of functions within the compiler is identical to a closed library routine on the original EDSAC. The advantages of the majority of control combinations are useless when using a compiler (for example, inbuilt program address offsets in instructions - unused in the compiler, which has a constant picture of all addresses at all times), however closed subroutines are an example of where control combinations are essential. Subroutines on EDSAC are implemented by loading the value of the return address into the accumulator, then jumping to the instruction start using the value in the accumulator. The first instruction of the subroutine is always **GK**, which loads the value of the instruction into address 42, followed by **A3F** and **TnK**, where n is the penultimate address of the subroutine (the final address should remain a STOP code). This combination constructs a return instruction in the accumulator, combining the initial instruction from the calling instructions with a constant value to construct a jump back to the calling point. This is then outputted into the instruction penultimate to the end of the subroutine (which must leave the accumulator empty), injecting a return to calling point at the end of the subroutine [9].

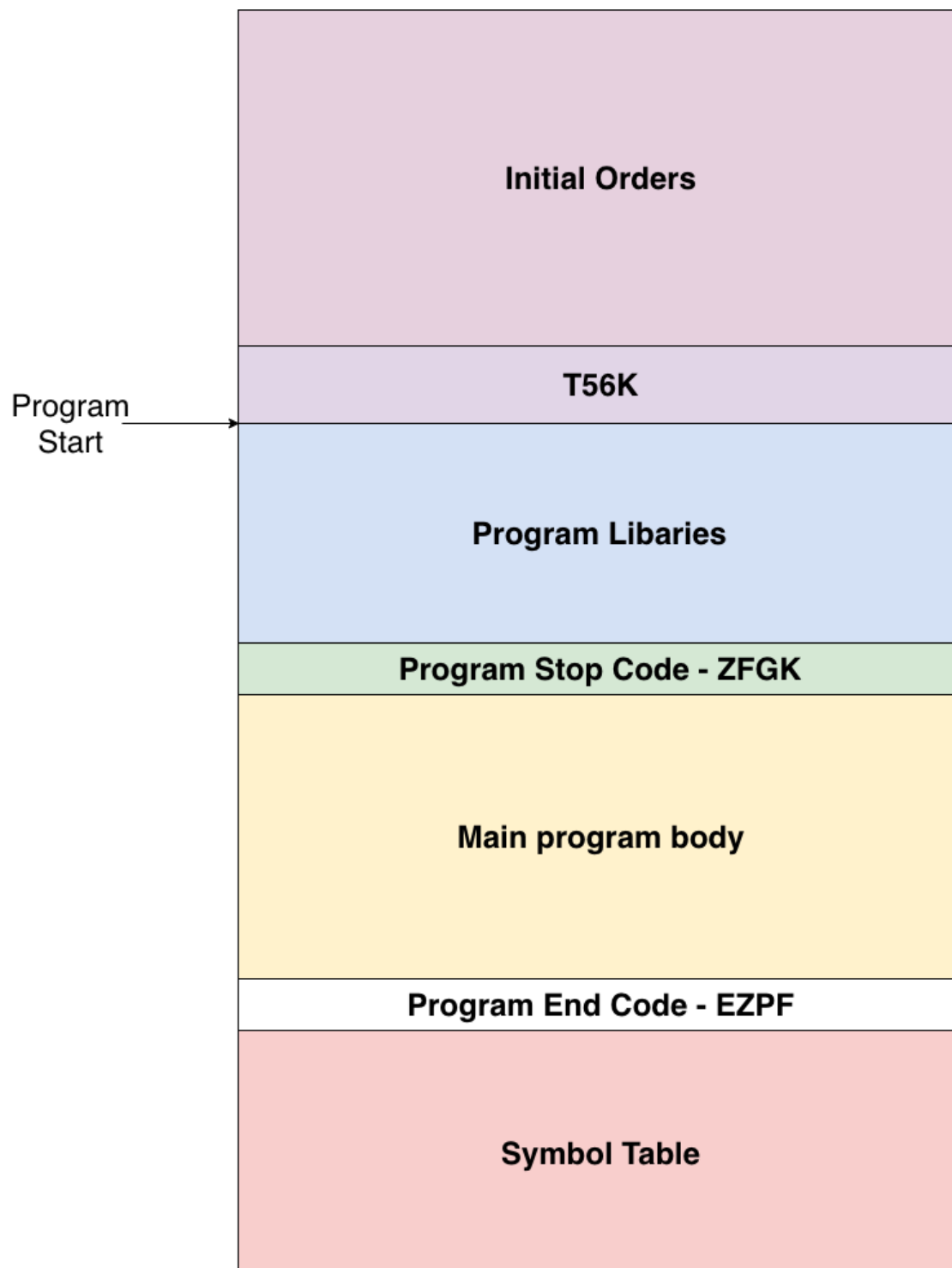


Figure 4.2: The structure of a program outputted by the compiler.

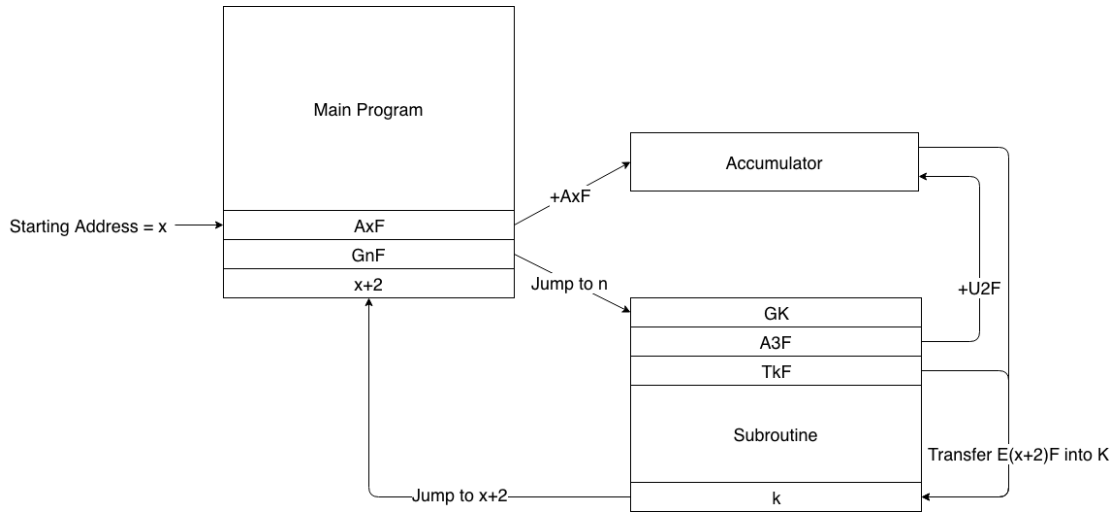


Figure 4.3: **Self-modifying function example, which injects a return value at the end of the function.**

An illustration of this process is included in Figure 4.3, where x is the starting address, n is the first address of the subroutine and k is the last address of the subroutine. Additionally an example is included in Appendix H.

4.5.4 Symbol Table

The singular Symbol Table is a misnomer, there are in reality, a minimum of four Symbol Tables. One for Declared, Undeclared, Temporary and Common variables. Each of these is used interchangeably depending on the context that the variable is declared within. These four symbol tables are contained and accessed through a controller, known as STController. This controller provides a single accessor method for the whole compiler. STController also provides methods for adding to each specific table, which are identical with the exception of temporary variables. Temporary variables are added, and then only accessed by a reference which must be maintained by the calling function, as when a reference is destroyed it cannot be retrieved. In this way, STController acts as a single controller and access point for all data values inside the compiler. Note that the data structures of both STController and each Symbol Table are included in figure 4.5.

4.5.4.1 The relation of Three Op Code

The proposed design for the actual Symbol Table instance ties in with the proposed design for the internal representation of instructions, or three op code. Each instruction contains an address, which is stored inside the internal representation as a pointer to an address in the symbol table. As each program statement is processed, variables and values in the symbol table are added and retrieved. The initial symbol table starts from address zero and builds up references to values stored within it as integers, and when

retrieved a pointer to this integer is returned. In this way, at the end of the generation of three op code, each instruction holds a pointer to a value within the symbol table. These values can then be offset, modifying the values held in all instructions at the same time.

A similar technique is used for line mapping, where an instruction might hold in the address field a pointer to an entry in the overall line-map for the entire program. A rolling track of the size of the program is maintained, and each line reference is offset by the size of the final program generated before it. Figure 4.4 shows this, albeit in a simplified form.

4.5.4.2 Function Symbol Tables

When in a function scope, STController generates four new Symbol Tables, and directs all operations on the Symbol Table to these instead of the main program Symbol Tables. In this way, each statement processed is unaware of whether the program is currently inside a function or subroutine, with STController providing a consistent unified interface for all Symbol Table operations. This approach also allows a high level of debugging information, with STController able to filter and log Symbol Table operations as they are processed.

4.5.5 DO Loops

The implementation of DO loops comes in two parts, the handling of the statement DO, and a line by line check for the ending of that DO loop. DO loops in FORTRAN follow the syntax `DO<Integer><Variable>=<Integer><Integer>`. Alternatively, the later two integers can be replaced by variable references, although for simplicity we will consider them as integers here. The semantics of this statement are simple, loop from the current line up to the line labeled DO, iterating the first integer each loop. Once the iterating integer equals the second, exit the loop, but involves a two phase process:

1. Upon seeing a DO statement, enter the loop. Overlapping DO loops are not allowed, however nested DO loop's are. A 'FIFO' queue is employed here. If the end of a loop isn't the most recent loop added, then the loops are overlapping and an error is thrown.
2. The second stage involves a separate area of code which checks each line label as a statement is processed. If the line label is equal to the end of a DO loop, the back of the FIFO queue is popped and compared the to loop. If the wrong loop is at the back of the queue, the loops are overlapping. See Figure 4.6.

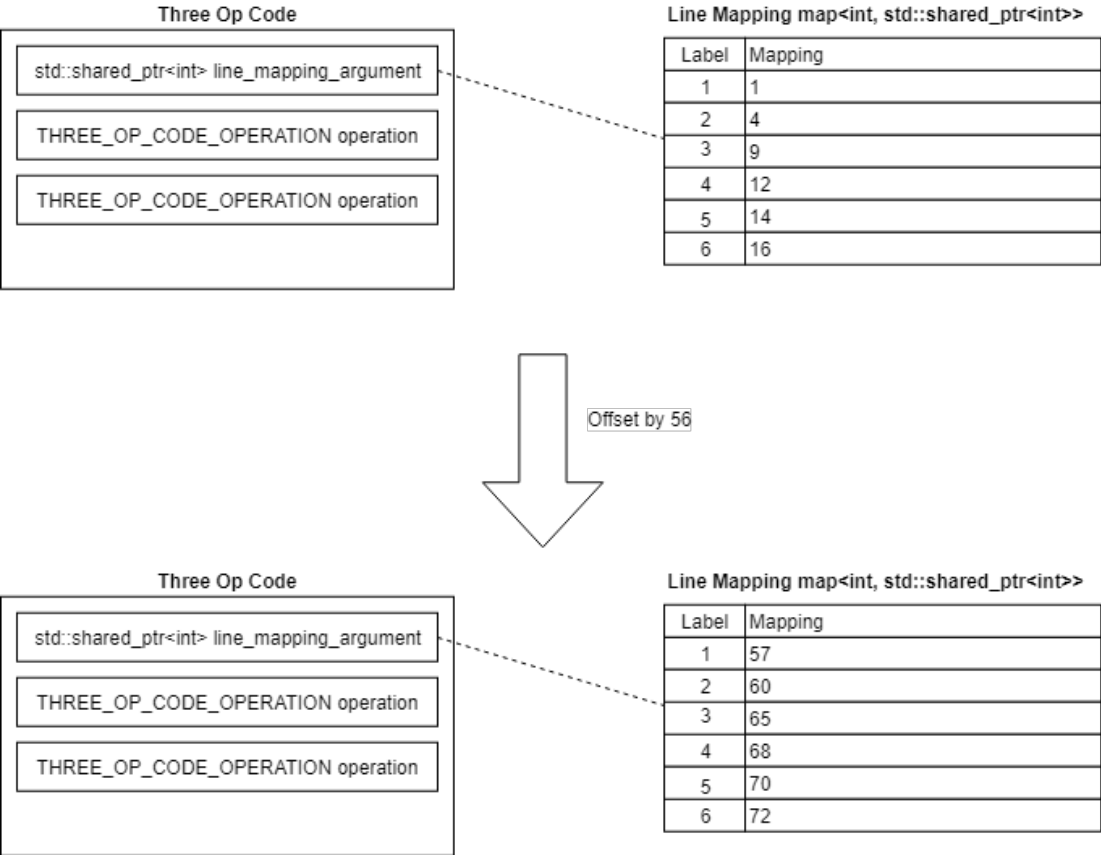


Figure 4.4: A simple illustration of the concept of mappings

An example of a sample DO loop, including input FORTRAN and output EDSAC code, is included in Appendix G.

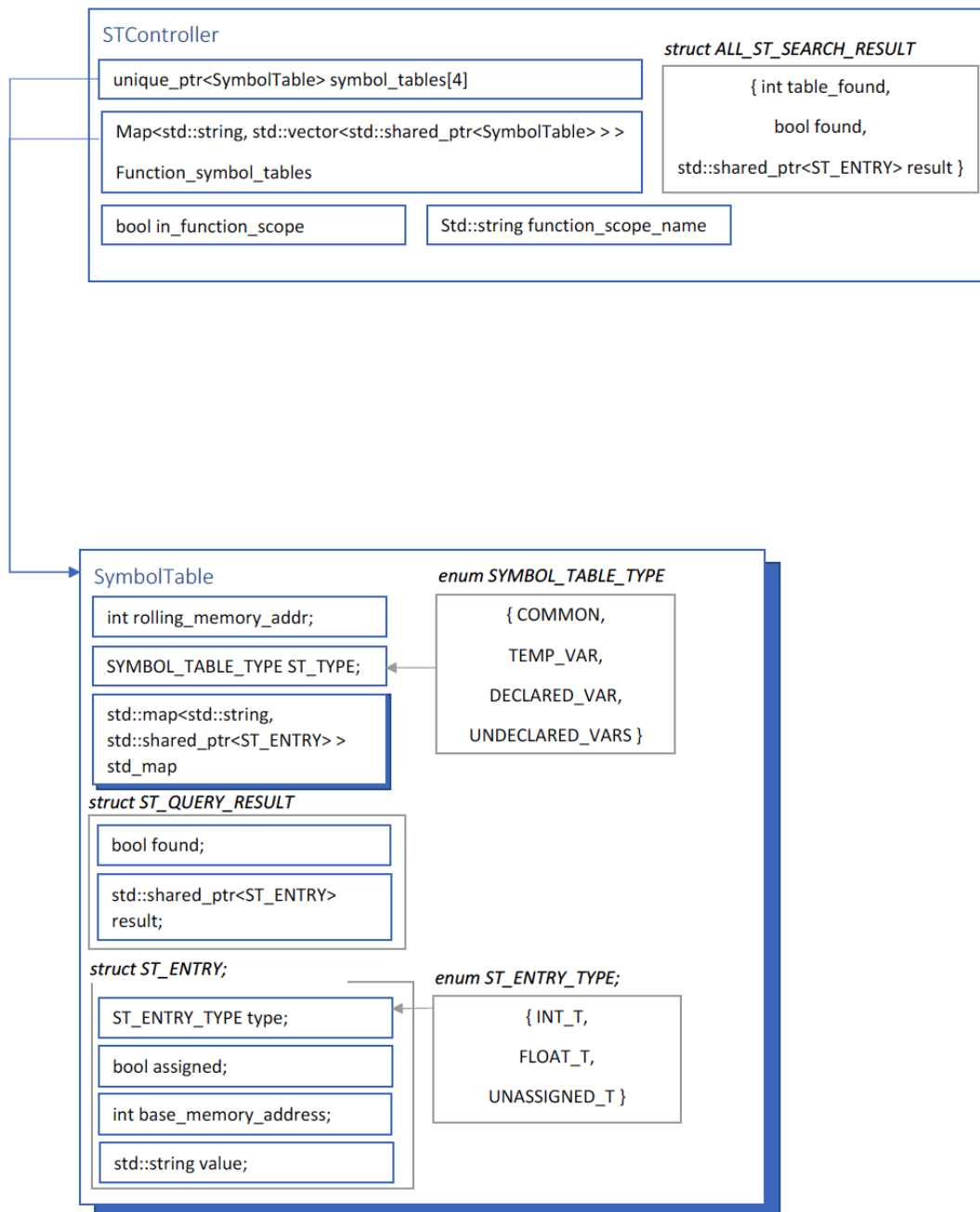


Figure 4.5: The data structures of the Symbol Table and the interfacing controller class

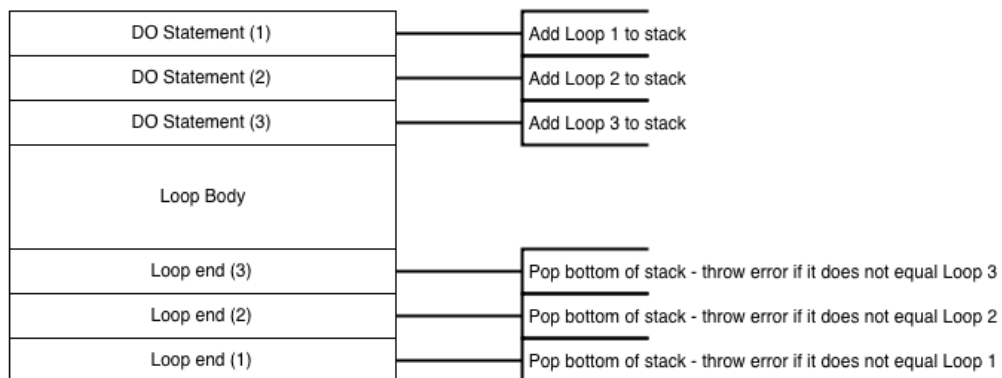
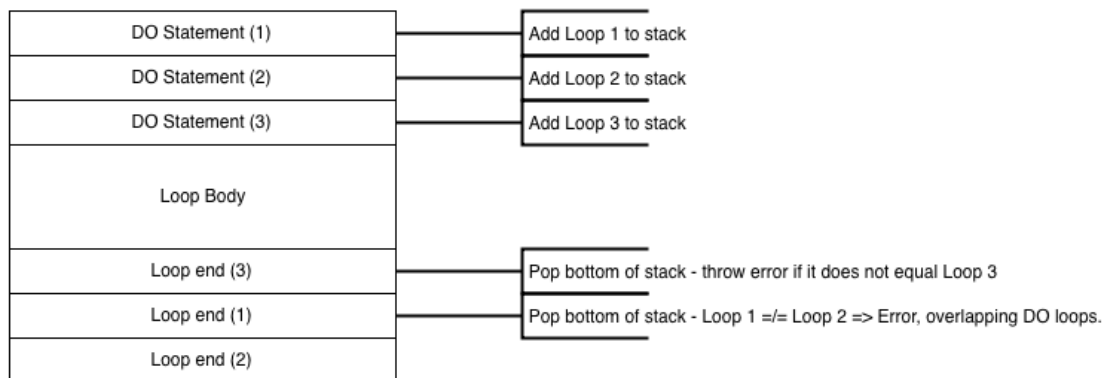
Valid nested DO loops**Invalid overlapping DO Loops:**

Figure 4.6: An example of Overlapping DO loops and their implementation with a First In First Out queue.

5. Justification for approach

5.1 Design considerations

The chosen approach when considering the design of the compiler was affected by several key factors:

1. Resources on the compiling machine are near infinite - relative to the size of memory on EDSAC resources on the compiling machine were extremely unlikely to present a serious issue.
2. The compiler must be well-documented, well structured and maintainable. As such, tidy, structured code and clear data structures are essential.
3. Compilers execute in a highly linear manner, with effectively a single internal state relating to the stage of the compilation process the compiler has reached.

The linear nature of a compiler, as well as the functionally limitless resources enabled by the compiling machine, led the design towards a similarly linear structure. Data structures are designed to map onto, or encapsulate their proceeding stages, ultimately giving a progressive and linear flow of data throughout the program. The benefit of this is clear; the motion of each stage of the compiler from one consistently defined data structure to another gives ample opportunity for output and debugging data to be dumped at every stage of compilation. This is an important factor, both for finding bugs in the code and for debugging compiled code, and supports the projects goal of a high level of documentation and debugging output. Allowing for transparency in the compiler is also a useful tool for anyone attempting to work with the compiler, or indeed anyone looking to improve / expand on the available functionality.

Throughout, an endeavor has been made to ensure that the project documentation is generally informative (both in terms of planning and implementation), without removing the level of granular detail required for another party to attempt to reproduce any work.

5.2 Software considerations

With the project specified in C++, and a minimal use of library functions, considerations for the software are reasonably limited. It is obvious that the software should aim to be modular and maintainable, avoiding complex code and ensuring a high level of documentation throughout - particularly important for large C++ projects. The compiler is targeted at C++14, and makes use of several C++14 features.

FORTRAN II was specified for the project. FORTRAN II is a suitable language for the project as a close relative to EDSAC both in terms of time period and functionality.

5.2.1 Platform agnosticism

The software was intended to be platform agnostic, and that has been achieved through the careful use of standard library functions and platform agnostic libraries. Note that full version information for each compiled binary, in addition to the supporting software, is included in Appendix D, and full details on compilation are included in the Users manual.

5.2.2 Included tools and libraries

In general, usage of libraries has been kept to a minimum, and the following largely outlines reasonably trivial aspects of the project which are generic in nature, or offer some significant improvement over the language-included library functions.

5.2.2.1 Boost

One library used reasonably extensible throughout the project is Boost. Boost offers a variety of expanded library functions to C++, and a more powerful regular expressions engine than included in the standard libraries. As such, some of the more complex statements are dissected using Boost.

5.2.2.2 Flex and Bison

The backbone of the Arithmetic parser is Flex and Bison, modern versions of the classical lex and yacc[31]. Flex intakes a predefined set of tokens, and Bison a predefined grammar, with the combination of the two producing an autogenerated lexing and parsing solution. This is ideal for relatively simple languages, i.e. simple arithmetic statements. In the case of the compiler, the automatic lexer and parser are defined as `lex.1` and `parse.y`.

5.2.2.3 cxxopts.h

Cxxopts is an open source C++ command line argument parsing library. It is header only, and bundled under an MIT license (See Appendix E for license declarations). Cxxopts is useful in providing a simple and easily defined command line argument interface, including properly formatted help pages. cxxopts.h is included in the design archive, having been reproduced from [\[32\]](#).

6. Testing

6.1 Unit testing

6.1.1 Tokenization Testing

Parsing FORTRAN II is a difficult task, one which cannot rely on traditional lexing and parsing tools. With the exception of arithmetic statements, each FORTRAN II statement is identified as a whole, before being dissected in a manner unique to the statement type. As a result, a large volume of testing was performed on this stage of the compiler, ensuring that a FORTRAN II statement can be accurately and correctly loaded into an internal representation. These tests are not designed to produce any meaningful EDSAC output, and may invoke a variety of semantic errors, however they are validated against output from the `--parsedvalues` and `--tokens` flags, which enables statement unique output as to the values extracted in addition to printing the identification of each Statement as it is parsed. A full suite of tokenization tests are included in the design archive, and tokenization testing for an ASSIGN statement is included in Appendix J.

The breakdown of statements parsed following the ‘Beginning Tokenization’ section of the output is the most relevant, showing each parsed statement, a line number, and the identified type, as well as any values extracted from the statement. This is also where errors in the parsing stage are output.

6.1.2 Statement Testing

The semantics of each generated are tested in isolation, with functionality being tested in a series of small programs. The reasoning behind small, simple programs for this is two-fold. The likelihood of interference from other statements masking errors is minimised, in addition to making programs easier to debug, simpler to step through and practical to understand. Included below is an example of such a test, in this instance for a GOTO statement. The statement is tested in isolation with a single, hard coded values, in a simple program. The intention of these tests is not to push edge cases onto statements, but to establish their core functionality running on the EDSAC simulator, and to detect

surface level bugs and implementation issues in individual statements. This also allowed for a consideration of different EDSAC implementations of each statement. A full suite of these tests is included in the Design Archive.

```
1000  STOP
      GOTO 1000
      END
```

Listing 6.1: Statement test for a GOTO statement

6.2 Integration Testing

Integration testing effectively forms an expansion on the previous two forms of testing, with integration tests representing more complex scenarios and functionality tests, as well as the more advanced semantics of the language. This section forms the bulk of the testing of the compiler. An example integration test is included below, as well as the full suite of integration tests included in the design archive. This example tests replicated the semantic functionality of Subroutine scoping, validating that arguments to Subroutines exist in an independent scope, and are passed properly.

Ultimately, integration testing is designed to push the boundaries of the compiler, including the implementation of more difficult semantics and capabilities.

```
C Expected output for this test is two integer PRINTS, '100 200'.
  SUBROUTINE SCOPETEST(X)
  PRINT X
  RETURN
  X = 100
  PRINT X
  CALL SCOPETEST(200)
  END
```

Listing 6.2: Function Scoping Integration Test

7. Future Work

7.1 Common Blocks

The compiler has no support for COMMON blocks, an important method of transferring data around larger FORTRAN programs. The compiler is capable of parsing COMMON blocks, however there is no support in the Symbol Table. This would be a relatively easy implementation, requiring some modification of the symbol table to support proper referencing of COMMON blocks (variables in the COMMON scope are declared by index, not name).

7.2 Floating Points

Floating point support in the compiler would involve another reasonably large modification, as the symbol table currently treats all values as single address values. This would be a significant expansion on the functionality of the compiler in terms of more complex mathematical processing.

7.3 I/O

Currently the compiler only supports basic output, and no input. Ideally, the compiler would make use of a subroutine (for example, M20) to implement reading of data values. This would be a significant improvement to the real-time functionality of the compiler, as well as the demonstrative potential of programs for the EDSAC replica project.

7.4 Division

Division is not implemented within the compiler, and is likely a feature that should be considered after the implementation of floating points. While it would be relatively easy to implement Division operators on the current version of the compiler, it would

be largely redundant, with no method of checking whether a value was divided into a whole or fractional number, and no support for any further operations with fractional numbers.

8. Project Outcome

8.1 Project Documentation

The project has been extensively documented throughout, both in terms of the finished product and supporting documentation. Similarly, a large majority of the research done on EDSAC is centralised in the work for this project, leaving both this document and the supporting files as an essential asset for anyone considering continuing the work done on developing an extensive software platform for EDSAC. A large proportion of time has been dedicated to ensuring the that project is maintainable and widely documented. In these regards, the project has been a success. The project documentation is thorough, cumulatively consisting of:

- A Users Manual for the Compiler itself
- A singular large program flow diagram
- A condensed program flow diagram
- A specification of the Regular Expressions used to identify language constructs, and several smaller state transition diagrams.
- A collection of working examples
- Several smaller flow charts designed to illustrate algorithms for specific processes in the compilers execution.
- The EDSAC disassembler - designed to make the process of dissecting and understanding legacy EDSAC instructions easier.

8.2 Aside - EDSAC disassembler

The EDSAC disassembler was a side project completed with the intention of making EDSAC example programs more accessible, as well as fostering an understanding of the instruction set. This proved particularly useful, taking instructions in in a binary format,

and outputting a mnemonic value as well as a short description of the instructions effect. An example of this is included in Appendix C.

8.3 The EDSAC Compiler

8.3.1 Usefulness of the EDSAC compiler

Overall, the compiler has been a strong success. It remains relatively stable, and decreases the time investment for writing simple programs for the EDSAC machine by an order of magnitude. The compiler supports the majority of language elements, and makes good use of the hardware available. Simple arithmetic programs, manipulating control flow, and basic output are all well supported, as well as more advanced control flow. This includes all three varieties of GOTO statement, and a full conditional IF statement.

The compiler is unlikely to be used in building large programs stretching the capacity of EDSAC, with a limited capacity and relative inefficiency in the generation of programs. The adaptation of FORTRAN II, and indeed a compiled language as a whole, for EDSAC is inevitably an inefficient one. However, the usefulness of the compiler for the rapid development of simpler programs, or even snippets of larger programs, is obvious. This is demonstrated in the included example programs, of varying complexity, which effectively demonstrate the functionality of the compiler. A complete description of the compilers functionality is included in the User Manual, which is bundled in addition to the source code as an element of the Design Archive.

8.3.1.1 Example Programs

Included in Appendix F are a selection of example programs. Additionally, a range of example programs are included in the Design Archive. These range from simple, to more complex. The majority perform simple arithmetic and control flow related tasks on EDSAC, demonstrating jumps in control flow, conditionals, equality operators and simple arithmetics.

8.3.2 Benefits of the Compiler

The compiler abstracts the user away from some of the more difficult elements of EDSAC programming, including the complexity of functions, control flow, basic printing and the need for a constant awareness of the size of the program and the location of all data values. Additionally, the ability to rebuild previously generated code with easy modifications - for example using a different starting address (or offset) is a useful tool.

This includes the pre-bundling of required subroutine libraries, and easy use of computation elements which are significant challenges to program in EDSAC - for example exponentiation, subroutines and print output. The compiler offers a clear and significant benefit, albeit with a functionality ceiling. This is ultimately to be expected, as the true potential of EDSAC (and indeed any machine) is inevitably lost through a process of automation and adoption of higher level languages.

8.3.3 Incompatibilities and performance

As mentioned previously, some elements of the FORTRAN language are particularly difficult to implement on EDSAC. As a part of this, some modifications to the language have been made in order to ensure compatibility, details of which are contained within the Users Guide. In addition, some statements, while semantically correct, are significantly more expensive than alternate methods available to a user programming directly for EDSAC. This is an inevitable trait of adopting a higher level language, however the project has endeavoured to maintain the best possible implementation for EDSAC where available.

9. Critical Evaluation

9.1 Completion of Goals

Overall, the project has been extremely successful in completing the defined goals. The standard and breadth of documentation is high, and while the scope of functionality implemented in the compiler was less than originally targeted, the quality of outputted programs is good. The project encompasses a large subset of FORTRAN II accurately and efficiently. The project faced a considerable challenge both in terms of the scale of implementation, but also the as a research exercise, and is overall regarded as an excellent success.

In particular, the lexing and parsing stage of the compiler has produced extremely satisfactory results, with the process of identifying statement types via regular expressions prior to parsing saving a significant amount of complication. Programming for ED-SAC presented a unique challenge, with a wealth of functionality hidden within the tiny instruction set and the complex semantics it obscures.

In addition to being capable of a large portion of the FORTRAN II language, the compiler remains platform agnostic, and has been built with an outward focus on documentation designed for another party. The intention remains that the project may be expanded upon or continued in the future.

9.2 Scope of implementation

The scope of the project implementation is less than the full scope of FORTRAN II, however implements the majority of features. The breadth of features implemented was controlled, and the projects original outlook of not overreaching for functionality at the expense of quality informed the size of the subset implemented. The planning behind the implementation, particularly the prioritisation of particular language elements or constructs alongside the robust structure of the program, enabled the size of this subset to change almost freely as problems (and solutions) were encountered. Contingency planning in this regard was essential.

9.3 Quality of implementation

The quality of implementation of the compiler is strong, with a purposed and modular structure, high levels of code documentation, and a relatively extensible design. This was best reflected throughout the latter stages of development, when new features for the compiler appeared effortless to implement, a reasonable sign of a strong design and sufficient foresight.

The compiler is functional and free of serious bugs, however there is room for improvement, with small bugs a recurring feature. A strive for durability was made, reflected in the testing performed, although the complexity of specific data structures (namely line / address mappings) remains a source of frequent and small bugs.

Error detection, both semantic and syntactic, is a particular strong point of the compiler, with a variety of both triggered error messages and optional debugging output. The final output of the compiler, with all features enabled, is a wealth of information about the program structure, a feature possible only from the modularity of the compiler's data structures.

9.4 Quality of documentation

Overall, as mentioned throughout this document, the quality of documentation is excellent. Some areas of the projects code base are particularly complex, and an effort has been made to keep these areas in particular documented to a standard where another user could continue to work on the code without significant interruption.

9.5 Project planning

The planning of the project as a whole has been a success, delivering an in scope, functional and genuinely useful product. An overreaching of features was avoided, and documentation has been maintained throughout. While the development stage of the project overran initial projections, contingency planning for the under / overrunning of the development ensured that a deliverable product was completed, without sacrificing quality or testing or documentation. Detailed planning of each stage of the development cycle, on both a time management and implementation level, was essential. Included in the Design Archive is the vast majority of project planning documentation, although Gantt charts for both the overall project and more detailed development cycle are included in Appendix F.

9.6 Conclusion

Throughout, the project has presented a considerable challenges, however with careful planning, management and grounded expectations, it is one that has been well met. The final product is useful, though limited, and well-documented, in addition to offering a genuinely useful functionality for the EDSAC replica project. It is hoped that the project offers value to a potential user base, and is expanded upon in the future to support a wider range of functionality for EDSAC.

Bibliography

- [1] “Recreating edsac.” [Online]. Available: <http://www.tnmoc.org/special-projects/edsac/recreating-edsac>
- [2] A. Herbert and D. Hartley, “Edsac replica project,” *Making the History of Computing Relevant IFIP Advances in Information and Communication Technology*, p. 297–308, Feb 2017.
- [3] M. V. Wilkes and W. Renwick, “The edsac (electronic delay storage automatic calculator),” *Mathematics of Computation*, vol. 4, no. 30, p. 61–61, Jan 1950.
- [4] C.-K. Martin, “Tutorial guide to the edsac simulator.”
- [5] M. Bentham, “Echo edsac simulator.” [Online]. Available: <https://www.cl.cam.ac.uk/events/EDSAC99/simulators/echo/flat.html>
- [6] M. Richards, “Edsac initial orders and squares program.”
- [7] “Edsac 99.” [Online]. Available: <https://www.cl.cam.ac.uk/events/EDSAC99/reminiscences/#EDSAC>
- [8] D. Cohen, “Oxo aka noughts and crosses - the first video game,” Mar 2019. [Online]. Available: <https://www.lifewire.com/oxo-aka-noughts-and-crosses-729624>
- [9] E. E. Osborne, M. V. Wilkes, D. J. Wheeler, and S. Gill, “Preparation of programs for an electronic digital computer,” *Mathematical Tables and Other Aids to Computation*, vol. 12, no. 64, 1951.
- [10] M. Campbell-Kelly, “The edsac simulator,” 2016. [Online]. Available: <https://www.dcs.warwick.ac.uk/~edsac/>
- [11] W. Cunningham, “Self modifying code,” Jul 2014. [Online]. Available: <https://wiki.c2.com/?SelfModifyingCode>
- [12] C. Hadley, “The history of the computing lab,” Aug 2004. [Online]. Available: <https://www.cl.cam.ac.uk/relics/history.html>

- [13] “Rediscovered edsac diagrams reveal secrets,” Jul 2014. [Online]. Available: <http://www.tnmoc.org/news/news-releases/lost-edsac-diagrams-reveal-secrets-one-earliest-computers>
- [14] S. Gill, D. Wheeler, and M. Wilkes, “Edsac initial orders(id:3411/eds002).” [Online]. Available: <http://hopl.info/showlanguage.prx?exp=3411>
- [15] M. V. Wilkes, “The use of the edsac for mathematical computation,” *Applied Scientific Research, Section B*, vol. 1, no. 1, p. 429–438, 1950.
- [16] S. Riley, Mar 2018. [Online]. Available: https://www.youtube.com/watch?time_continue=17&v=nc2q4OOK6K8
- [17] A. Herbert, “Edsac number formats,” Apr 2019.
- [18] M. Wilkes, “Arithmetic on the edsac,” *IEEE Annals of the History of Computing*, vol. 19, no. 1, p. 13–15, 1997.
- [19] J. V. Neumann, “First draft of a report on the edvac,” Jun 1945.
- [20] A. W. Burks, H. H. Goldstine, and J. Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” *The Origins of Digital Computers*, p. 399–413, 1982.
- [21] “Devlin’s angle.” [Online]. Available: https://www.maa.org/external_archive/devlin/devlin_12_03.html
- [22] *General Information Manual: Programmer’s Primer for FORTRAN*, 1957. [Online]. Available: www.ibm.com/ibm/files/T507917N26894N49/us_en_us_ibm100_fortran_programmers_primer.pdf
- [23] M. Campbell-Kelly, “Edsac program documentation,” *University of Warwick Department of Computer Science*, Jul 1996. [Online]. Available: <http://cd.textfiles.com/230/EMULATOR/DIVERSE/EDSAC/EDSACDOC.PDF>
- [24] P. McJones, “History of fortran and fortran ii,” Jul 2016. [Online]. Available: <http://www.softwarepreservation.org/projects/FORTRAN/>
- [25] J. W. Backus, H. Herrick, and I. Ziller, “Specifications for the ibm mathematical formula translating system, fortran,” p. 29, Nov 1954. [Online]. Available: <http://archive.computerhistory.org/resources/text/Fortran/102679231.05.01.acc.pdf>
- [26] D. Harper and L. Stockman, “The history of fortran.” [Online]. Available: <https://www.obliquity.com/computer/fortran/history.html>
- [27] “History of fortran language.” [Online]. Available: <https://www.livephysics.com/computational-physics/fortran/history-fortran-language/>

-
- [28] 1958. [Online]. Available: <http://archive.computerhistory.org/resources/text/Fortran/102653989.05.01.acc.pdf>
- [29] “Fortran standards documents.” [Online]. Available: https://gcc.gnu.org/wiki/GFortranStandards#Fortran-_Automatic_Coding_System_for_the_IBM_704
- [30] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Pearson India Education Services, 2015.
- [31] J. R. Levine, T. Mason, and D. Brown, *Lex and Yacc*. OReilly, 1995.
- [32] jarro2783, “Cxxopts,” Jan 2019. [Online]. Available: <https://github.com/jarro2783/cxxopts/>

Appendices

A. Word Count Considerations

The word count of the submitted version of this report is 9880, a figure calculated using an online PDF word counter on the stripped pdf. Additionally, the word count was validated with ‘texcount’, a popular and well regarded LaTeX word count tool. This figure does includes sections 2 through 9, and does not include the bibliography. The word count inside figures is negligible, and is not considered.

B. Project Planning documentation

The following section includes various Gantt charts. Both follow the same key, included in figure [B.1](#). Also note that the main project management Gantt chart is split across two pages, primarily due to its size. These are labelled ‘Project time plan’ and ‘Project time plan (Continued)’. The Development Time Plan Gantt chart shows the individual stages of development in more detail, in addition to the allotted overflow time for each stage of development.

Figure B.1: Gantt Chart Key





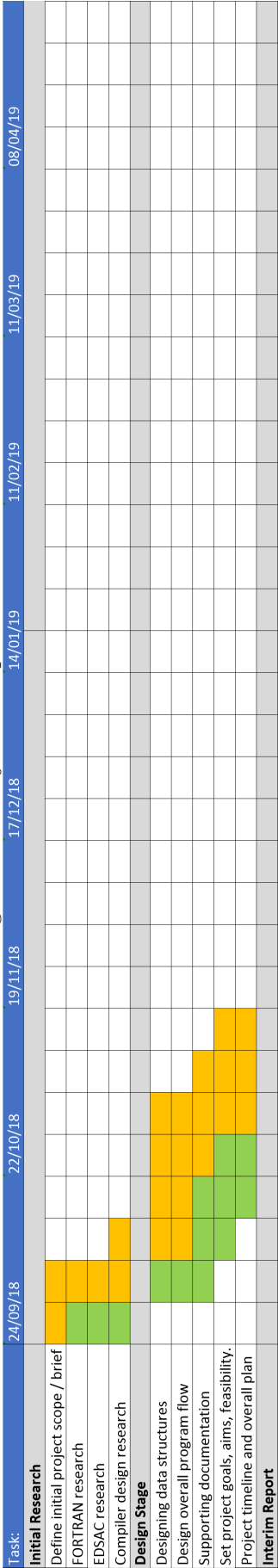
Key:	Colour:
Intended Time period	
Early work	
Late work	
Overflow period	

Figure B.2: Project time plan



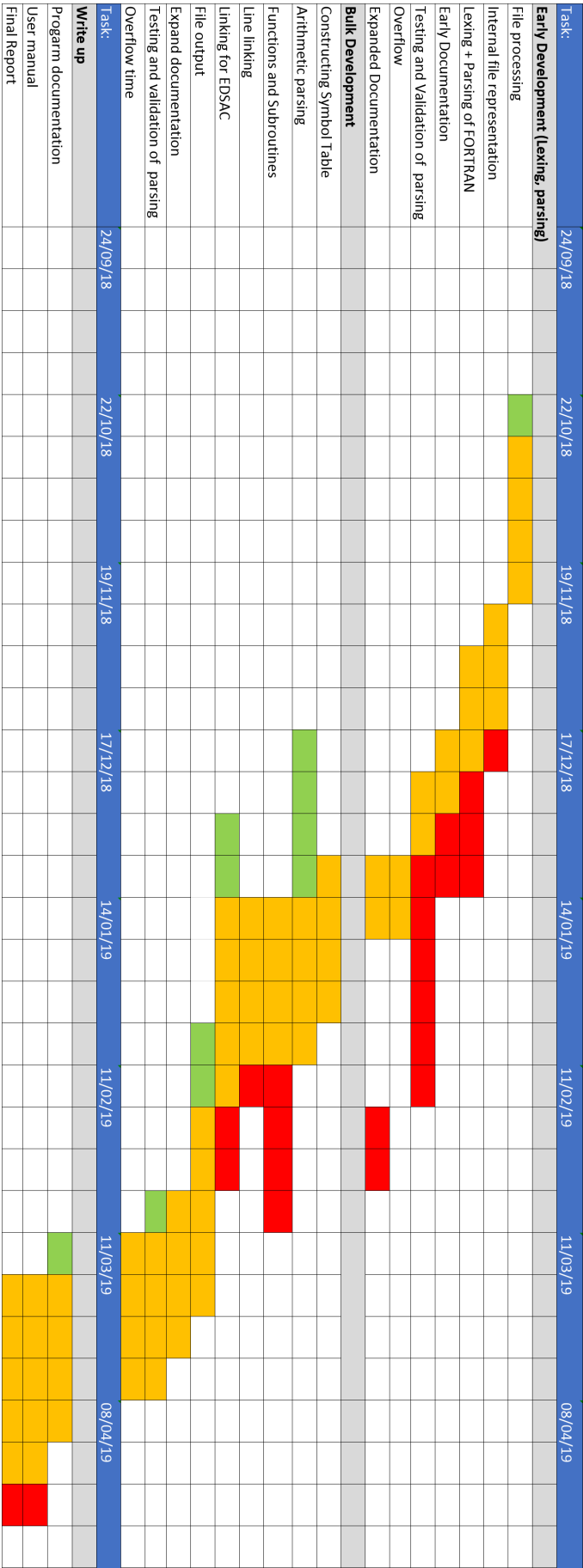
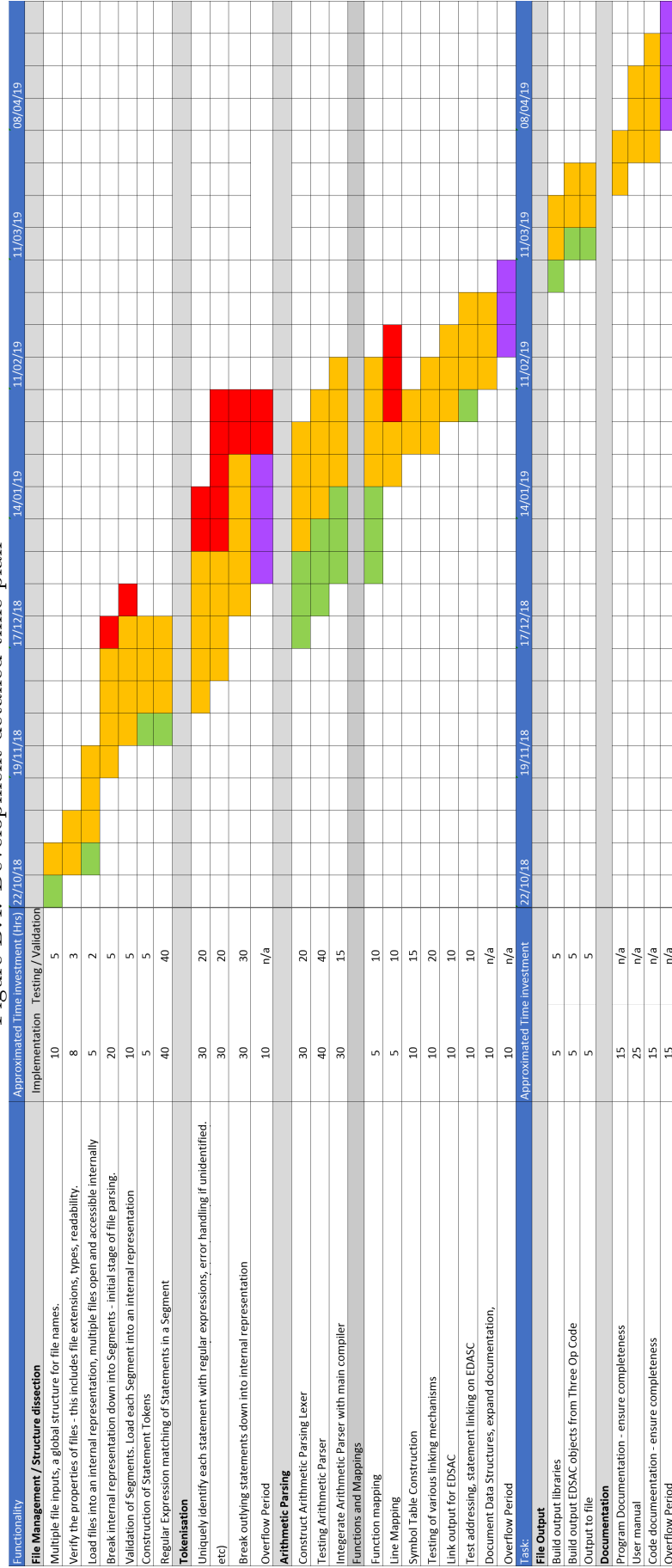


Figure B.3: Project time plan (Continued)

Figure B.4: Development detailed time plan



C. EDSAC Disassembler Output

```
00101 0 0000000000 0 | TOS | Transfer the contents of the accumulator to
      storage location 01 and clear the accumulator

10101 0 0000000010 0 | H2S | Copy the number in storage location 21 into the
      multiplier register

00101 0 0000000000 0 | TOS | Transfer the contents of the accumulator to
      storage location 01 and clear the accumulator

00011 0 0000000110 0 | E6S | If the sign of the accumulator is positive, jump
      to location 61; otherwise proceed serially

00000 0 0000000001 0 | P1S | data 11

00000 0 0000000101 0 | P5S | data 51

00101 0 0000000000 0 | TOS | Transfer the contents of the accumulator to
      storage location 01 and clear the accumulator

01000 0 0000000000 0 | IOS | Read the next character from paper tape, and store
      it as the least significant 5 bits of location 01

11100 0 0000000000 0 | AOS | Add the number in storage location 01 into the
      accumulator

00100 0 0000010000 0 | R16 | Shift the number in the accumulator 16 places to
      the right

00101 0 0000000000 1 | TOS | Transfer the contents of the accumulator to
      storage location 01 and clear the accumulator
```

D. Software Versions

The following table includes the software versions used for compiled binaries on each platform. This represents the target version for each platform, and compilation on other versions comes with an increased chance of errors caused by changes to the used tools.

Software Version	MacOS	Linux	Windows
Flex	2.6.4	2.6.4	2.5.4
Bison	3.3.2	2.3.0	3.0.0
C++14 Compiler	clang 10.0.1	G++ 7.0.3	MSVC 141
Boost	1.69	1.69	1.69

E. License Statement

The following is required under the MIT License covering cxxopts.h, which is included in the provided software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT

OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

F. EDSAC Character Output Codes

This section includes the character codes for EDSAC, from which the bit values for instructions, as well as input and output for data values, can be computed. It is important to note the Figure Shift and Letter Shift characters, which when used on Martin Campbell-Kelly's EDSAC simulator, toggle the teleprinter output between figure and letter shift. This toggle determines whether the numeric or character value of the data value will be outputted to the screen. Reproduced from [\[4\]](#).

Perforator		Teleprinter			
Letter Shift	Figure Shift	Letter Shift	Figure Shift	Binary	Decimal
P	0	P	0	00000	0
Q	1	Q	1	00001	1
W	2	W	2	00010	2
E	3	E	3	00011	3
R	4	R	4	00100	4
T	5	T	5	00101	5
Y	6	Y	6	00110	6
U	7	U	7	00111	7
I	8	I	8	01000	8
O	9	O	9	01001	9
J		J		01010	10
p		Figure Shift		01011	11
S		S	”	01100	12
Z		Z	+	01101	13
K		K	(01110	14
Erase1		Letter Shift		01111	15
Blank Tape 2		(No effect)		10000	16
F		F	\$	10001	17
q		Carriage Return		10010	18
D		D	;	10011	19
f		Space		10100	20
H	+	H	£	10101	21
N	-	N	,	10110	22
M		M	.	10111	23
D		Line Feed		11000	24
L		L)	11001	25
X		X	/	11010	26
G		G	#	11011	27
A		A	-	11100	28
B		B	?	11101	29
C		C	:	11110	30
V		V	=	11111	31

G. DO loop example

The following is an example annotated input and output from a FORTRAN DO loop.

```
x = 0
```

```

        DO 10I = 1,10
10      PRINT I
        END

```

The compiled code is included below, broken down statement by statement. The subroutine P6 is excluded due to its size. Note that [] tags represent comments.

```

T56K          - Set program entry point to address 56.
[P6]          - The start of the P6 subroutine.
[excluded]
[Program Start]
GKZF          - Stop the program, requiring the user to start again
              once loded.

[DO Statement]
T109F          - Construct copies of each loop variable
A114F          - This is done by adding the value into the accumulator,
              and transferring it to a new address.
T110F          -
A115F          -
T112F
T109F          - Clear the accumulator into our buffer flush variable.
A110F          - index = index + 1
A113F          -
U110F          - Transfer out of the accumulator without clearing
S112F          - Subtract the loop end value from the current index
E108F          - If the result is positive, then index = loop end, so
              jump
T109F          - If not - clear the accumulator.
[End DO statement]

[Print Statement]
T109F          - This is the body of the loop, a print statement
              - Add the value to the accumulator, transfer it into
              address zero
A110F
TF
A104F          - Add the return address to the accumulator
G56F          - Jump into the subroutine
[End Print Statement]

[Injected DO Loop Return]

T0F           - Jump back to the top of the loop
E94F          - Clear the accumulator to zero, then jump.

[End Injected DO loop return]

[End Program and Symbol Table]
ZF
POF
POF
POF
P5D
PD
POF
P5D

```



```
[End Tape Token]
E
Z
PF
```

Listing 1: DO Loop annotation - Annotated EDSAC output

Note how the print statement is constructed, and immediately following the PRINT statement, control flow returns to the top of the loop. This return is guaranteed to complete, and exiting the loop is handled by the instructions at the top of the loop. [E108F](#) handles escaping the loop, jumping past the final injected DO instruction. In this case, the final instruction is ZF (Stop program), but in the case of a longer example, this would jump into the first instruction of the next statement processed.

H. Subroutine jump example

The following is an example of a subroutine jump from instruction 106 to 56, and then a return back to 107 once the subroutine has completed.

```
[105]    A105F
[106]    G56F
```

The accumulator after instruction 105 has the value (shortened):

```
[Acc]    11100 0 0001101001 0
```

The subroutine then immediately executes the following:

```
[56]     GK
[57]     A3F
[58]     TxF
```

This adds the value of address 3 ([U2F](#)), to the accumulator. This results in the value [E107F](#) being constructed in the accumulator. Instruction 58 then copies the value of the accumulator to x , the final instruction of the subroutine. Provided the subroutine leaves

the accumulator clear, this instruction forms a link return, jumping program flow back to instruction 107 [9].

I. Arithmetic Parser Example

The following is an annotated example of FORTRAN II code utilising the Arithmetic Parser, which allows inline computation of values. The following simple example program demonstrates this:

```
X = 100
Y = 42
PRINT X + 1 - 2 * Y
END
```

The statement `X + 1 - 2 * Y` is passed to the arithmetic parser, which constructs a parse tree representing the expression. In this example, the parse tree would be similar to that included in Figure 4.1. The output is then prepended to the calling function, leaving computed variables available to the Statement. In the above example, the following code would be outputted.

```
T56K
[P6]
[P6 omitted for brevity]
GKZF          - Main program start
T109F          - Print statement start (Arithmetic operations begin here)
H114F
V111F
L64F
L64F
T115F
T109F
A110F
A112F
T113F
T109F
A113F
S115F
T116F          - Arithmetic operations end, output is in 116.
T109F          - Print statement operations begin here, having completed
arithmetic preprocessing.
A116F          - Print the value of address 116 (the computed arithmetic
)
TF
A106F          - Jump into print subroutine
G56F
ZF            - Symbol table begins here.
```

[Symbol table omitted for brevity]

Listing 4: Annotated output from a PRINT statement, demonstrating the inclusion of arithmetic processing.

J. Testing examples

I.1 Tokenization testing

The following is an example of tokenisation testing for ASSIGN statements. This process is not automated, mainly due to the ‘intermediary’ stage of the Compiler which is being tested.

```
C      Tokenization testing for ASSIGN statements.
C
C
C      Expected to pass
C          ASSIGN Y TO 5
C      Expected to pass
C          ASSIGN X TO 5
C      Expected to pass
C          ASSIGN TO TO 5
C      Expected error case - We need to pass an int into the assignment.
C          ASSIGN X TO X
C      Expected Error case - assignment needs to be an integer.
C          ASSIGN TO TO TO
C      Expected to fail, assignment must be an integer.
C          ASSIGN TO TO 5+5
C      Not expected to pass
C          ASSIGN TO TO TOT0
C      This is expected to parse as ASSIGN TO TO 5
C          ASSIGN
C          T0
C          T0
C          55
C      This won't parse, ASSIGN will catch the size of the integer and overflow.
C          ASSIGN ASSIGNASSIGNASSIGNASSIGNASSIGNASSIGNASSIGNASSIGN TO
33252971317821423126254115152107205183189187255255291233424418512616313916112
C      This is expected to pass - Negative numbers are valid
C          ASSIGN NEWVARIABLE TO -500
C      This shouldn't pass - we don't allow arithmetic
C          ASSIGN NEWVARIABLE TO -5-100
C          END
```

Listing 5: ASSIGN Tokenization Tests (input FORTRAN)

The following is the output from the above test input.

```

===== EDSAC FORTRAN II Compiler =====

:: Loading Command Arguments ::

+parsed-values
+dump tokens
+load: Tests/Stmt_Tests/ASSIGN_tests.f

:: Loading File Inputs ::

--- Tests/Stmt_Tests/ASSIGN_tests.f ---

:: Beginning preliminary parsing of files ::

:: Beginning Tokenization ::

+PROGRAM [1,29]

--- New Segment type=(PROGRAM) ---

[5][ASSIGN_TOKEN] { ASSIGN Y TO 5 }
[INFO] Loaded variable name : Y
[INFO] Loaded variable assignment : 5
[7][ASSIGN_TOKEN] { ASSIGN X TO 5 }
[INFO] Loaded variable name : X
[INFO] Loaded variable assignment : 5
[9][ASSIGN_TOKEN] { ASSIGN TO TO 5 }
[INFO] Loaded variable name : TO
[INFO] Loaded variable assignment : 5
[ERROR] Cannot find valid token for line [11]{ASSIGN X TO X}
[ERROR] Failed to identify token.
[ERROR] Cannot find valid token for line [13]{ASSIGN TO TO TO}
[ERROR] Failed to identify token.
[15][ASSIGN_TOKEN] { ASSIGN TO TO 5+5 }
[ERROR] Value 5+5 could not be parsed - it is not a valid number. Arithmetic
statements are not allowed in ASSIGNMENT operations.
[ERROR] Failed to initialise token.
[ERROR] Cannot find valid token for line [17]{ASSIGN TO TO TOTOT}
[ERROR] Failed to identify token.
[19][ASSIGN_TOKEN] { ASSIGNTOTOT5 }
[INFO] Loaded variable name : TO
[INFO] Loaded variable assignment : 5
[INFO] Remapping variable TO
[24][ASSIGN_TOKEN] { ASSIGN ASSIGNASSIGNASSIGNASSIGNASSIGNASSIGNASSIGNASSIGN
T033252971317821423126254115152107205183189187255255291233424418512616313916112
}

```

```

[ERROR] Value
33252971317821423126254115152107205183189187255255291233424418512616313916112
could not be parsed - it is not a validnumber. Arithmetic statements are not
allowed in ASSIGNMENT operations.
[ERROR] Failed to initialise token.
[26][ASSIGN_TOKEN] { ASSIGN NEWVARIABLE TO -500 }
[INFO] Loaded variable name : NEWVARIABLE
[INFO] Loaded variable assignment : -500
[28][ASSIGN_TOKEN] { ASSIGN NEWVARIABLE TO -5-100 }
[ERROR] Value 5-100 could not be parsed - it is not a valid number. Arithmetic
statements are not allowed in ASSIGNMENT operations.
[ERROR] Failed to initialise token.
[29][END_TOKEN] { END }

--- End Tokenization ---

:: Standard Library Initialisations ::

[INFO] Building bundled libraries.
[INFO] Skipping M20, remains disabled
[INFO] Skipping P6, remains disabled
[INFO] Total library size: 0

:: File Outputs ::

[INFO] Skipping file output for Three Op Code - dump_three_op_code is disabled.
[INFO] Writing EDSAC output to out.edsac
[INFO] Finished writing out.edsac

Time taken: 0.01s

Output Summary:
Finished with 12 errors, 0 warnings and 20 info messages.

==== End of runtime ====

```

Listing 6: ASSIGN Tokenization Testing User Output

K. Example programs

The following section contains a variety of example programs compiled for EDSAC. This includes the FORTRAN program, as well as the expected output and compiled EDSAC program.

K.1 Cubes

One of the earliest programs written for EDSAC was a program which calculated the cubes from numbers $1 - > N$. This is reproducible in FORTRAN as follows (where N is 10):

```

        ASSIGN X TO 2
100    CONTINUE
        PRINT X ^3
        X = X + 1
        GOTO 100
        END

```

Listing 7: Cubes - FORTRAN input sourcee

```

T56K
[P6]
GKA3FT25@H29@VFT4DA3@
TFH30@S6@T1FV4DU4DAFG
26@TFT05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF031@E
20@J995FJF!F.PZ
GKZF
T128F
A129F
T134F
A130F
T135F
A135F
S132F
U135F
T128F
A135F
S132F
U135F
G109F

```

```

T128F
H129F
V134F
L64F
L64F
U134F
E97F
T128F
A134F
T131F
T128F
A131F
TF
A115F
G56F
T128F
A129F
A136F
T137F
T128F
A137F
T129F
T128F
A138F
E89F
ZF
P0F
P1F
P1D
PF
PD
P1F
PF
PF
PD
PF
PD
E
Z
PF

```

Listing 8: Cubes - Compiled EDSAC output

The output of this is:

8	27	64	125	216	343	512	729	1000	1331	1728	2197	2744	3375	4096	4913	5832
---	----	----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------

K.2 IF Statements

The following is a simple example of IF statements implemented in FORTRAN, translated across to EDSAC.

```

C Example Program - IF Statements
  ASSIGN X TO -1
  IF(X) 10, 15, 20

```

```

        PRINT 0
        STOP
20      PRINT 5
        STOP
10      PRINT 10
        STOP
15      PRINT 15
        STOP
        END

```

Listing 9: IF Statements Demonstration - FORTRAN input

The output of this is:

```
10
```

```

T56K
[P6]
GKA3FT25@H29@VF
T4DA3@TFH30@S6@T1F
V4DU4DAFG26@TF
TF05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF
031@E20@J995FJF!F..PZ
[Main Program]
GKZF
T120F
A121F
G107F
S122F
G113F
E101F
T120F
A123F
TF
A98F
G56F
ZF
T120F
A124F
TF
A104F
G56F
ZF
T120F
A125F
TF
A110F
G56F
ZF
T120F
A126F
TF
A116F
G56F
ZF
ZF
POF

```



```

P-1D
PD
P0F
P2D
P5F
P7D
E
Z
PF

```

Listing 10: IF statements demonstration - Compiled EDSAC output

K.3 Descending Integers

The following is a slightly more complex IF Statement example, including the use of GOTO statements and self-referential arithmetic.

```

      X = 10
5     IF (X) 20, 10, 10
10    PRINT X
      X = X - 1
      GOTO 5
20    STOP
      END

```

Listing 11: Descending Integers - FORTRAN input source

```

T56K
[P6]
GKA3FT25@H29@VF
T4DA3@TFH30@S6@T1F
V4DU4DAFG26@TF
TF05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF
031@E20@J995FJF!F..PZ
[Main Program]
GKZF
T112F
A113F
G110F
S114F
G95F
E95F
T112F
A113F
TF
A98F
G56F
T112F
A113F
S115F
T116F
T112F
A116F
T113F

```

```

T112F
A117F
E89F
ZF
ZF
POF
P5F
PD
PD
PF
PD
E
Z
PF

```

Listing 12: Descending Integers - EDSAC Output

The output of this is:

10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---

K.4 Sum of cubes

This example demonstrates an integration of self-referential arithmetic, exponents, PRINT statements and DO loops terminating back into control flow.

```

      X = 0
      DO 1000 I = 1,10
1000  X = X + I^3
      PRINT X
      END

```

Listing 13: Sum of cubes to N - FORTRAN input

```

T56K
[P6]
GKA3FT25@H29@VF
T4DA3@TFH30@S6@T1F
V4DU4DAFG26@TF
TF05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF
O31@E20@J995FJF!F..PZ
[Main Program]
GKZF
T139F
A144F
T140F
A145F
T142F
T139F
A140F
A143F
U140F
S142F

```

```
E133F
T139F
T139F
A140F
T151F
A146F
T152F
A152F
S149F
U152F
T139F
A152F
S149F
U152F
G121F
T139F
H140F
V151F
L64F
L64F
U151F
E109F
T139F
A151F
T147F
T139F
A141F
A147F
T148F
T139F
A148F
T141F
T139F
E94F
T139F
A141F
TF
A136F
G56F
ZF
P0F
P0F
P0F
P5D
PD
P0F
P5D
P1D
PF
PF
PD
P1F
PF
PF
E
Z
PF
```

Listing 14: Sum of cubes to N - EDSAC Output

The output of this, where $N = 10$, is:

3035

K.5 Sum of natural numbers up to N

This example briefly demonstrates the power of IF statements when combined with GOTO statements for more advanced control flow and conditional checking.

```

      X = 100
      RESULT = 0
      DO 10 INDEX=1,1000
5       IF(INDEX - X) 10, 10, 15
15      GOTO 20
10      RESULT = RESULT + INDEX
      GOTO 5
20      PRINT RESULT
      END

```

Listing 15: Sum of naturals - FORTAN input

```

T56K
[P6]
GKA3FT25@H29@VF
T4DA3@TFH30@S6@T1F
V4DU4DAFG26@TF
TF05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF
031@E20@J995FJF!F..PZ
[Main Program]
GKZF
T132F
A138F
T133F
A139F
T136F
T132F
A133F
A137F
U133F
S136F
E123F
T132F
T132F
A133F
S134F
T140F
T132F
A140F
G114F
S141F
G114F
E111F
T132F
A142F

```

```

E126F
T132F
A135F
A133F
T143F
T132F
A143F
T135F
T132F
E94F
T132F
A144F
E101F
T132F
A135F
TF
A129F
G56F
ZF
POF
POF
P50F
POF
P500D
PD
POF
P500D
PF
PD
PD
PF
PD
E
Z
PF

```

Listing 16: Sum of naturals - EDSAC Output

The output of this (where $X = 100$) is:

```
5050
```

K.6 Subroutine sum of Naturals up to N

This example is almost identical to the previous, except the sum of naturals is included as a subroutine.

```

SUBROUTINE SUMOFNATURALS(X)
  RESULT = 0
  DO 10 INDEX=1,1000
5    IF(INDEX - X) 10, 10, 15
15   GOTO 20
10   RESULT = RESULT + INDEX
    GOTO 5
20   PRINT RESULT

```

```

RETURN
CALL SUMOFNATURALS(200)
END

```

Listing 17: Subroutine sum of Naturals upto N - FORTRAN input

```

T56K
[P6]
GKA3FT25@H29@VFT4DA3@
TFH30@S6@T1FV4DU4DAFG
26@TFT05FA4DF4FS4FL4F
T4DA1FS3@G9@EFSF031@E
20@J995FJF!F.PZ
GKA3F
T133F
T143F
A149F
T145F
A150F
T147F
T143F
A145F
A148F
U145F
S147F
E124F
T143F
T143F
A145F
S144F
T151F
T143F
A151F
G115F
S152F
G115F
E112F
T143F
A153F
E127F
T143F
A146F
A145F
T154F
T143F
A154F
T146F
T143F
E95F
T143F
A155F
E102F
T143F
A146F
TF
A130F
G56F
T0F
ZF

```

```

GKZF
T0F
A142F
T144F
A138F
G88F
ZF
P0F
P100F
P0F
P0F
PF
P0F
P500D
PD
P0F
P500D
PF
PD
PD
PF
PD
PF
E
Z
PF

```

Listing 18: Subroutine sum of Naturals upto N - EDSAC Output

The output of this, where $N = 200$, is:

```
20100
```

K.7 Calculation of Factors

The following example is a simple algorithm for calculating Factors of a number. In this case, the number set is X.

```

      X = 55
      DO 15 K = 2, 100
      DO 15 I = 2, 100
      IF(X - K*I) 15, 30, 15
15    CONTINUE
      STOP 1
30    PRINT K
      PRINT I
      STOP 5
      END

```

Listing 19: Calculation of the factors of X- FORTRAN input

```

T56K
[P6]
GKA3FT25@H29@VFT4DA3@

```

```
TFH30@S6@T1FV4DU4DAFG
26@TFTF05FA4DF4FS4FL4
FT4DA1FS3@G9@EFSF031@
E20@J995FJF!F..PZ
GKZF
T150F
A156F
T151F
A157F
T154F
T150F
A151F
A155F
U151F
S154F
E133F
T150F
T150F
A160F
T152F
A161F
T158F
T150F
A152F
A159F
U152F
S158F
E131F
T150F
T150F
H151F
V152F
L64F
L64F
T162F
T150F
A153F
S162F
T163F
T150F
A163F
G129F
S164F
G136F
E129F
T150F
E106F
T150F
E94F
T150F
A165F
ZF
T150F
A151F
TF
A139F
G56F
T150F
A152F
```



```

TF
A144F
G56F
T150F
A166F
ZF
ZF
P0F
PD
PD
P27D
P7D
PD
PD
P8F
P7D
PD
PD
P8F
PF
PF
PD
PD
P2D
E
Z
PF

```

Listing 20: Calaulation of the factors of X - EDSAC Output

The output of this program is, where $N = 55$:

```
5 11
```

K.8 Prime Numbers

The following example demonstrates a simple Prime Number calculator written in FORTRAN. This example further demonstrates the capability of Functions, Loops and GOTO statements, in addition to the use of 'Continue' as a placeholder.

```

      FUNCTION ISPRIME(X)
      Y = 1
      DO 15 K = 2,20
      DO 15 I = 2,20
      IF(X - I*K) 15, 30, 15
      C If prime, fall out the bottom of this loop
15    CONTINUE
      Y = 0
      GOTO 40
      C Not prime - exit the loop
30    Y = 5
      GOTO 40
40    RETURN Y
      DO 50 INDEX = 1, 15

```

```

      K = ISPRIME(INDEX)
      IF(K) 50, 45, 50
45    PRINT INDEX
50    CONTINUE
      END

```

Listing 21: Prime numbers upto N- FORTRAN input

```

T56K
[P6]
GKA3FT25@H29@VFT4DA3@
TFH30@S6@T1FV4DU4DAFG
26@TFTF05FA4DF4FS4FL4
FT4DA1FS3@G9@EFSF031@
E20@J995FJF!F..PZ
GKA3F
T146F
T187F
A195F
T189F
A196F
T193F
T187F
A189F
A194F
U189F
S193F
E134F
T187F
T187F
A199F
T190F
A200F
T197F
T187F
A190F
A198F
U190F
S197F
E132F
T187F
T187F
H190F
V189F
L64F
L64F
T201F
T187F
A188F
S201F
T202F
T187F
A202F
G130F
S203F
G140F
E130F
T187F
E107F

```

T187F
E95F
T187F
A204F
T192F
T187F
A205F
E146F
T187F
A206F
T192F
T187F
A207F
E146F
ZF
TOF
GKZF
T180F
A184F
T181F
A185F
T182F
T180F
A181F
A183F
U181F
S182F
E179F
T180F
TOF
A181F
T188F
A164F
G88F
T180F
A192F
G177F
S186F
G172F
E177F
T180F
A181F
TF
A175F
G56F
T180F
E154F
ZF
POF
POF
P8F
PD
POF
P8F
PD
POF
PD
PD
PF

```
PF
PD
P10F
PD
PD
P10D
P10F
PD
PD
P10D
PF
PF
PD
P0F
PD
P2D
PD
PF
E
Z
PF
```

Listing 22: Prime numbers upto N - EDSAC Output

The output of this program is:

1	2	3	5	7	11	13
---	---	---	---	---	----	----

The inclusion of 2 is a side effect of the particular algorithm used.

L. Design Archive Index

The design archive includes four folders:

- **Project Files:** The complete source of the compiler, including the necessary build tools.
- **Documentation:** The complete documentation of the compiler, including the ‘Users Manual’.
- **Binaries:** This folder contains compiled executables for the compiler on each supported platform.
- **Edsac Disassembler:** This folder contains the aside project, the EDSAC Disassembler.

M. Original Project Brief

Included in this section is the original project brief. The brief is included on a new page for readability.

EDSAC rebuild project: FORTRAN compiler

Elliot Alexander (28561716)
Project Supervisor: Andrew Brown

1 Problem Description

The National Museum of Computing, situated near Bletchley Park, has been attempting the reconstruction of EDSAC (Electronic Delay Storage Automatic Calculator); widely considered one of the first programmable computers. The project is attempting to rebuild the machine back to its original state at completion in 1947. Traditionally, running programs on the EDSAC initially involved defining programs by hand as a set of 'initial orders', however at a later date assembler functionality was added, allowing instructions to be defined by a set of mnemonics. Memory was a serious constraint as to the size of initial orders.

The problem is as such, currently there is no uncomplicated or efficient way to construct initial orders for the machine without an intimate knowledge of both the machines hardware, and instruction set.

2 Project Description

The immediate solution for this problem, as is common on embedded machines today, is a cross compiler capable of compiling a higher level language into machine executable code from within the comforts of a modern system. This compiler will be written in C++, designed to compile Fortran II code into a set of orders for the EDSAC machine, greatly reducing the likelihood of syntactic errors, as well as reducing both the barrier to entry and time expenditure required when programming the machine. The final cross compiler should be relatively easy to use, allowing several files to be loaded into the compiler at once, and outputting code appropriate to be loaded onto the EDSAC machine. The compiler should run as an executable, and may have several options for tweaking the properties of the EDSAC machine (i.e. expanded memory capacity).

3 Project Goals

The goals of the project are as such:

- The Compiler should interpret Fortran II code and output machine code for the EDSAC machine.
- The compiler should run on modern x86 machines as an executable
- The compiler should remain accurate and durable, with a focus on error handling and accuracy

4 Project Scope

The scope of the project will remain reasonably constricted, with accuracy and informative compiler errors and warnings being an essential part of the project. Therefore, the focus will remain quality, with the value of any extra features being carefully considered before an attempt to implement them is made. The goals of the project are well constrained, with a functional, accurate and reliable compiler taking precedent. In the instance that there is additional time or the project progresses ahead of schedule, a focus should remain on improving and expanding error detection, correction, and user warnings.