VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Debugging 42

Aden Kenny

Supervisor: Dr. Marco Servetto

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

42 is a statically typed, high level programming language which allows the composition and cooperation of millions of libraries at once. The 42 compiler, like the vast majority of software products, is not bug-free and therefore requires significant testing. This project has produced tests that have lowered the number of unknown bugs in the 42 compiler.

Acknowledgments

Contents

Figures

Chapter 1

Introduction

42 is a statically typed, high-level programming language designed to allow the use and composition of millions libraries simultaneously and [1]. The compiler is implemented in Java and is relatively untested. This project's aim is to design and implement a testing regime in order to identify bugs in the current implementation of the 42 compiler. Generally compiler bugs are often found through language usage but this does not mean that highly popular languages are free from them [2] [3] [4] [5]. This shows that we cannot guarantee bug free code in even the most used languages.

Compiler testing is generally considered difficult and time consuming, and testing by itself is unlikely to be adequate to guarantee correctness, [6] let alone being free of harmful bugs. This means that the best we can do without resorting to formal methods is simply finding as many bugs as possible with the resources available, which is what this project attempts to do. Therefore the overall goal is to create as many short 42 programs as possible and attempt to compile them. The programs that cause the compiler to throw an error or simply not behave as expected will be examined for the possibility of a bug. If a bug is found, a bug report will be created and sent to Dr Servetto where it will either be fixed or added to a to-do list of bugs to fix. Therefore this project has the singular goal of finding as many bugs as possible but it must be noted that goal of this project is not prove the absence of bugs in the 42 compiler, as testing cannot show or prove the absence of bugs [7].

2

Background and Related Work

2.1 The 42 Language

42 has a set of guarantees that are key to the language and and its use in practically any situation. These guarantees are also useful, as a user language knows that they will always hold true, except in the case of bug interfering with the language guarantees. The following section will quickly introduce the most important guarantees provided by 42.

2.1.1 Language Guarantees

Immutability

In 42 an immutable object, as the name suggests, is always immutable and will stay immutable for the rest of its life cycle [1]. A reference to an object is immutable by default which gives us some guarantees surrounding objects and makes it easier to reason about them when contrasted to objects that are mutable, or could become mutable in the future [8].

```
A: {
   S out = S"Hello, World!"
   out := S"Hello, 42!" // This is not allowed as out is immutable.

   var S out2 = S"Hello, World!"
   out2 := S"Hello, 42!" // This is allowed, out2 is mutable.
}
```

Encapsulation

For an encapsulated, mutable, object, a single ”capsule binding” to that object is the one and only means of accessing the encapsulated object and by extension its graph of mutable objects [1]. Immutable objects do not require this guarantee for obvious reasons.

Error safety

In 42 we have try blocks with catch clauses just like Java and C++. The main difference is that in 42 the any code that is ”guarded by a catch error” [1] (in a catch clause) cannot modify an external state. This allows much easier reasoning about code inside a catch clause, especially when combined with immutable references to objects.

Subtyping

Java code
In Java it is always possible to create a subtype of a class assuming the class is not declared final.

```
class A {
   A() {
      A a = B.b(new B()); // Won't work in 42. An exact type was not passed.
      B c = B.b(new A()); // Won't work either. Return type is not exact.
      A b = B.b(new A()); // This will work.
   }
}
class B extends A {
   static A b(A a) {return a;}
}
```

42 code
In 42 we cannot create a subtype of a class through extending it, and subtyping is significantly more restrictive [1]. The only way to get a subtype is through an interface, as all classes must be of an exact type therefore subtyping can only be accomplished through inheriting an interface. This means that we know that any class reference is of an exact type with one important exception to this is through interfaces where a reference to an interface type could be a type of any implementing class.

```
A : {interface
   {...}
}
B: {implements A
   class method // class keyword is equivalent to static.
   B b(B that) {return that}
}
C: {implements A
   class method
      A c(B that) { // We're returning an A.
         B known = that.b(that) // This must be of exactly type B.
         return known // This is okay as A is an interface, not a class.
      }
   }
}
D: {extends B // This won't work. We can't extend classes.
   {...}
}
```

2.2 Compiler Correctness

A compiler is an extremely large and complex piece of software; it is highly likely that bugs exist in the implementation of practically any language [9] and 42 is no exception to this.

Complier testing can be a difficult and time consuming process and Leroy [10] states that compiler-introduced bugs are widely considered to be extremely difficult to find and track down. Below we will explore some of the current approaches to compiler testing and debugging that we consider most important to this project, and explore how these ideas introduced in these approaches can best be applied to this project.

2.2.1 CompCert

The only way to truly prove a piece of software is bug free is through formal verification and Miller [11] states that the best way to make a systematic statement about the correctness of the code is to use formal methods, but the work required to create a formally verified of any real scale is a major barrier to using formal methods in compilers. The only example of a large scale compiler being fully formally verified is CompCert as presented by Leroy [10]. CompCert is a formally verified compiler for a large subset of the C language [12] [13]. CompCert is a verified compiler that provides some semantic preservation and has "a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program" [14].This, theoretically, rules out the possibility of bugs being introduced by the compiler and means that any bug free code that is compiled will produce a bug free program.

The CompCert compiler is an interesting project and contains useful information for this project, but to carry out similar project for 42 language is unrealistic for multiple reasons. First and foremost is that the writing of a formally verified compiler is significantly out of scope of this project, especially considering the CompCert compiler has had over 6 man years of effort spent on it. Additionally if the approach used by Leroy was to be applied to a language different than the C subset introduced by Blazy [13] some issues may arise. The CompCert compiler, as of publication of Leroy's paper [10], only produces code for PowerPC, a RISC ISA. This is in direct contrast with 42 which targets the x86 ISA, and targeting x86 may be more difficult due to it being a CISC ISA. Finally, the 42 language could prove to be more difficult to produce a verified compiler for due to difficult to verify features such as metaprogramming and large numbers of libraries in a 42 project.

2.3 Automated Testing

2.3.1 Fuzzing

Fuzzing is the technique of supplying random input into a program or application. This technique is quite often considered to be quite simple [15] but can be effective as it is fairly easy to implement and can find a large number of bugs[11] [15] [16] [9]. An example of a modern, well used fuzzing tool is *american fuzzy lop* (AFL) [17]. AFL differs from other older and less advanced fuzzing tools such as *fuzz* [11] by the use of compile-time instrumentation and genetic algorithms. This allows AFL to "discover clean, interesting test cases that trigger new internal states in the targeted binary" [17]. AFL has been used to find bugs in major programs such as *Apple Safari*, *llvm*, *Adobe Reader*, and *VLC* [17].

Potential Fuzzer Usage

A potential issue with using AFL is that the interface to run it on Java programs [18] is less used and less well documented than the standard AFL version. While fuzzers are a useful tool that have worked well on similar projects such as *llvm*, using a fuzzer on the 42 compiler may present some issues. The most pressing issue would perhaps be compilation

time of a 42 program using the 42 compiler. Compiling a 42 program currently takes a significant period of time (around 45 seconds on fairly standard consumer hardware) and if compiling large numbers of programs, this compilation time would quickly add up to a large period of time. If an attempt was made to fuzz the compiler by providing random 42 programs it would require writing a fuzzer from scratch which is almost certainly beyond the scope of this project, as well as still having the previously mentioned disadvantage of significant compilation times. These issues do not preclude the use of a fuzzer but a more careful approach would have to be taken perhaps with the use of a different fuzzer [19] [20] focusing more on fuzzing distinct parts of the compiler rather than fuzzing the entirety of the compiler.

2.4 General Testing Literature

2.4.1 Compiler Testing

Automated Testing

In a paper presented by Chen et al. the topic of automated compiler testing[21] is introduced. They note that traditional automated testing techniques such as fuzzing have performance issues, with many tests and a significant amount of time needed to find bugs. They then introduce the idea of *learning to test* and propose LET, an approach to compiler testing that allows the prioritisation of test programs through a learning process, where a model is trained. This model is then used to prioritise test programs based on their likelihood of finding a bug and their performance costs. The results are quite impressive with around 60% of test cases being sped up by at least 25%. While the results are significant, the usage of the findings in regards to this project seems unlikely. This is primarily due to the approach being cutting edge, and seemingly difficult to replicate or even use if it were to be available.

In another paper Chen et al. present a comparison of compiler testing techniques [22]. They compare three different compiler testing techniques, Randomized Differential Testing (RDT), Different Optimization Levels (DOL) (a variation of RDT), and Equivalence Modulo Inputs (EMI). The results show that DOL is effective at finding bugs relating to optimisation, and RDT is generally more effective at detecting other types of bugs, but all three techniques can be used together to complement each other to a certain extent. The results presented here are not particularly useful to this project. 42 does not have a heavily optimising compiler or different levels of optimisation, therefore DOL would not be of much use if applied to this project. RDT is used with two different compilers implementing the same specification and 42 only has a single complier implementing the specification therefore RDT is not useful to us. In contrast, EMI could be useful to this project. EMI is fairly similar to LET presented in another paper by Chen et al. [21] therefore the viability of using EMI and LET should be explored for this project.

Kossatchev and Posypkin present a survey of ”Compiler Testing Methods” [23] from which we can extract some useful information. A key idea that can be taken from this paper is that ”the algorithms for generating syntax-oriented test suites are time-tested ones. There remains a question of whether the use of automated methods for generating semantics-oriented tests ... is justified from the practical standpoint, since the approaches discussed have been tested on subsets of real languages or on model programming languages”. This can be related to 42, where it is clear that automated methods of generating simple programs that test the syntax of the language is a fairly easy task. In contrast it is likely that automated testing of of non-syntax, or more ”semantic-orientated” parts of the language would be

significantly harder. This is due to the fact that it could potentially require the automated tool to ”learn” how to generate 42 programs that are valid and complex enough to test anything beyond the parser. While it is certainly possible that bugs exist in the 42 parser, and it is known that parsing bugs can be extremely serious, especially from a security perspective [24], we feel that testing of the language features is a higher priority at the present time.

This does raise some doubts over the usefulness of automated compiler testing in this project, especially the use of fuzzers as discussed previously. It is felt at this time that while parser bugs are not our priority, the use automated testing tools such as fuzzers will still be pursed. It is unlikely that they will find many bugs beyond parsing related issues, but the usage of them will have minimal impact upon the manual testing, therefore they can still be useful tools in identifying bugs in the 42 compiler.

2.4.2 Traditional Testing

Black-box testing

Ostrand defines black-box testing as ”any method of generating testcases that is independent of the software’s internal structure” [25]. Nidhra [26] states that ”the main advantage” of black-box testing is that a tester is not required to have knowledge of the programming language a system is implemented in, nor knowledge of the system itself. Nidhra also states that black-box testing aids ”in overall functionality validation of the system” [26]. This is important for a compiler as it would prove difficult to have complete knowledge of the overall system of a non-trivial compiler. Therefore, in the case of black-box testing a compiler, a tester would not have specific knowledge of the compiler or the source code. They would construct programs and attempt to compile them with the compiler that is being tested. The user would then compare the expected output of running the compiled code with the actual output, and if the actual output differs from the expected output, a bug has been identified. An issue that must be kept in mind with this style of testing on a compiler is that it is reasonable that the tester will write code that is incorrect, and if the tester does not realise this, it is possible that an identified bug is a false positive.

Black-box testing will be used for this project. It is probably the best choice, at least for the start of the project, primarily because of the lack of system or implementation knowledge needed. Black-box testing will also be used because the initial brief for the project stated that the majority of testing would be carried out by writing many small programs. This means that black-box testing is particularly suited as it focuses on writing programs that test the whole system, and writing programs that test the entire system of a compiler is fairly easy, as programs can be written and compiled through the compiler.

Orthagonal Latin Squares

A paper by Mandl [27] introduces the concept of experiment design on testing. He states when given a test objective, “we identify a state space spanned by a finite number of variables with a number of allowable values “ [27]. There are then two typical techniques for testing given this state space. The first option is to exhaustively cover the entire state space, and the second option is a random walk through the state space. Both of these techniques have rather major downsides. The exhaustive approach will present issues when the state space is large, as the number of tests required will be large. The random walk approach also presents problems, as the state space will not be fully tested, and “it is difficult to assess the level of confidence to be derived from the apparently successful testing. ” [27]. Mandl then introduces a third approach. This approach uses orthogonal Latin squares [28] to test “k variables each admitting n values” [27] (assuming finite values of k and n). We can create

a set of $k-2$ orthogonal $n \times n$ Latin squares, and from this implement n^2 test cases corresponding to the entries of the squares, rather than the n^k test cases that would be required for an exhaustive test. Mandl states that the we gain "essential exhaustiveness" at a lower cost. Mandl then cites an example of $n = 4$ and $k = 4$, where the orthogonal Latin squares approach requires 4^2 test cases compared to the 4^4 test cases required if we were to perform an exhaustive test. As a final note Mandl states that the "orthogonal-Latin-squares method being proposed seems to strike a very efficient compromise between the level of effort required and the amount of information obtained: At a cost commensurate with that of a traditional set of test cases selected at random, it yields about as much useful information as the prohibitively expensive exhaustive testing." [27].

This style of testing could prove useful for this project. For rather obvious reasons, exhaustive testing would not be viable for this project, but orthogonal Latin square testing as presented by Mandl could be modified to suit this project. These modifications would involve choosing test cases and setting an upper bound on the k and n values. This would be due to the fact that for compiler input it is very easy to imagine scenarios where k and n are extremely large and would involve very large squares. This proposed modification would mean that the testing would probably not yield as much information as exhaustive testing but it is likely that it would still be more cost-effective than a random walk through the state space.

White-box testing

Ostrand defines white-box testing as "test methods that rely on the internal structure of the software" [25].

Grey-box testing

Expand this section

3

Design

The overall design of the solution for this project mainly focused on the creation of short 42 programs. These 42 programs were then turned into tests that were added to the automated test suite. The goal of the project was to discover bugs in the 42 implementation, and a black-box testing style was considered to be the best way of testing the implementation. The project followed engineering and testing best-practices which helped to ensure that the outcome and deliverables of the project were of as a high of a standard as possible.

This chapter will cover the requirements that were set out for the project, the design of the system, process of design, and the alternative designs and approaches that were considered.

3.1 Requirements Analysis

The requirements for this project were gathered from Dr. Servetto, the supervisor for this project. The initial project outline stated that "the purpose of this project will be to write many super short 42 programs trying to leverage to corner cases of the implementation and discover bugs." Further requirements were then gathered through discussion with Dr. Servetto during the initial planning phase of the project.

The requirements were quite wide in scope which meant there was significant freedom in how they were interpreted. The overall scope of the project was stated as;

- Small 42 programs will be created
- These programs will be turned into tests and added to the 42 test suite
- These programs will generally be manually written
- These programs will generally be written in a "black-box" style

and from this scope, the requirements of the project were extracted. This included further discussion with Dr. Servetto, the primary stakeholder, especially when clarification was needed. The overall requirements of the project were stated as;

- Write small 42 programs
- Turn programs that display certain characteristics into automated tests

Due to the wide scope, the requirements could be considered to be vague, but they are as specific as possible for the scope of the project. This is due to the project being one of testing

a compiler in a black-box style. This scope impacts the requirements as the black-box testing does not have be a single method, and the tests do not have to focus on a specific part of the compiler. This leads to the requirements being very simple and straightforward, as seen above.

3.2 System Design

3.2.1 Black-box Testing

A black-box testing methodology was the main methodology applied to this project. This methodology was chosen for two major reasons. Firstly, the outline of the project stated that "small 42 programs will be created", which is a goal that can only be fulfilled with a testing methodology that is at least partially of a black-box style.

Secondly, discussion with Dr. Servetto led to the conclusion that a black-box style should be used for this project. A major reason for this was that as the time frame for this project was limited to two trimesters (approximately six months), it was felt to be infeasible for a newcomer to the project to gain enough system knowledge to be able test effectively test in a white-box style, and still write enough tests to make the time spent worthwhile. The time spent to gain a high level of system knowledge would be likely to be a significant period of time, as like most compilers, the 42 compiler is a complex piece of software. It was felt that the time required would probably be better spent writing more tests.

Black-box testing has a significant number of techniques to choose from, and a subset of the techniques were chosen for use in this project;

- Boundary testing
- Error guessing
- Fuzzing
- Modified orthogonal squares
- Syntax testing

These techniques were chosen as they provided a good blend of coverage, ease of use, and suitability for the given project. Boundary testing was chosen as compilers are a system where boundary issues such as buffer overflow, and integer overflow frequently manifest, and can cause major problems. Error guessing, is sometimes seen as primitive and not very useful, was chosen primarily for its value during the initial phase of the project, while the language was still being learnt. It can also be useful while testing compilers as many compilers have suffered from the same kinds of bugs or undefined behaviour, and error guessing allows a tester to implement lessons learned from past experience and mistakes [29]. Fuzzing was chosen as it is has proven to be quite effective for testing compiler testing [17] [30]. The orthogonal square testing method used in this project was a modified version of Mandl's method [27]. It was modified to more closely suit the project, as even with the reduction in tests required by using Mandl's method, exhaustively testing the 42 compiler would be impossible. The method was modified by limiting the size of the square, and by combining multiple test cases into one to reduce compilation times.

Expand this section...

While white-box testing was rejected as the main methodology used for this project, some aspects of it were used in this project, primarily through the usage of grey-box testing. Grey-box testing can be a rather broad term, but in this specific case it meant blending black-box testing techniques with system knowledge. This system knowledge was not to the level that would be expected for white-box testing, but it still provided valuable insight while testing. The idea behind grey-box testing was that source code for a certain part of the compiler would be read, and then black-box style tested would be written that were specifically targeted at the chosen part. The motivation was that these tests could be more effective than purely black-box tests, due to the fact that vulnerabilities in the code could be identified and then exploited.

3.3 Alternative Approaches

Alternative approaches to this project were considered, but for various reasons these alternative approaches were found to be inadequate, inferior to the chosen approach, or not suited to the requirements of the project. When the project was initiated, it was decided that broadly scoped, manual testing would form the majority of the tests written for this project.

While a black-box methodology was chosen for this project, a subset of testing methods had to be chosen, as some methods proved to be unsuitable for the goal of testing a compiler, and the outcomes of other styles were considered to be outweighed by the cost of implementation. Two other major types of testing were considered, white-box testing, and a more automated testing regime.

3.3.1 White-box Testing

White-box is a testing methodology that focuses on "test methods that rely on the internal structure of the software" [25]. This means that the author of the tests has significant knowledge of the inner workings of the system that they are testing. Major white box testing styles include structured testing [31], branch testing, path testing, and statement coverage [32] [33].

Advantages

One of the main advantages of a white-box testing methodology is that a tester is required to have a good knowledge of the system. This means that the tester may have a good idea of where bugs could be found in the system, as they understand the code, and should therefore understand where testing effort would be best applied.

White-box testing is said to be easy to automate [32], which can be a major advantage, especially when testing a large scale project such as a compiler.

Disadvantages

One of the main disadvantages of white-box testing is that high levels of system knowledge are required. This means that before starting testing, the tester will have to spend time familiarising themselves with the system instead of testing.

Table 3.1: A brief comparison of the advantages and disadvantages of testing methodologies considered for this project

Testing style	Pros	Cons
Black-box	No knowledge of system required Unbiased view of codebase Tests are relatively uncoupled from system implementation	Practically infinite test state space Lack of knowledge of system may hinder efforts Can lead to large numbers of tests testing the exact same code
White-box	Knowledge of system may help find bugs Test state space is smaller than black-box Tooling may exist to help testing	Requires system knowledge Scope of testing can be extremely large Tests may be strongly coupled to implementation
Grey-box	The majority of pros of both black and white-box testing	The majority of cons of both black and white-box testing

3.3.2 Automated Testing

Advantages

Disadvantages

4

Implementation

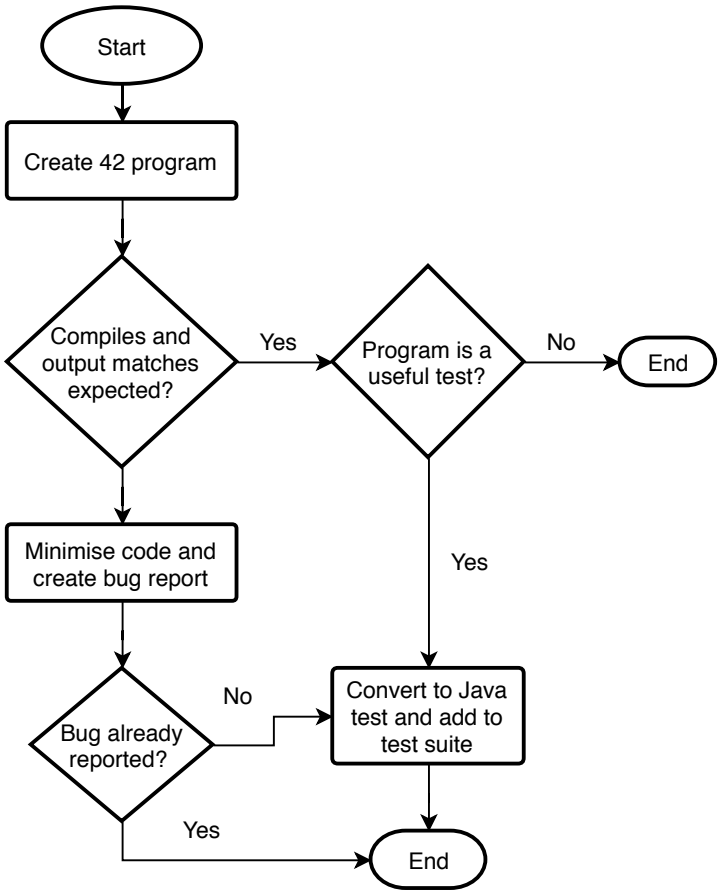

Figure 4.1: A flowchart of creating a standard 42 test

5

Evaluation

5.1 Introduction

Evaluation of this project is difficult as it is impossible to know exactly how many of bugs exist in the 42 compiler as it is impossible to prove that a piece of software is bug-free [7]. It is possible, but probably not likely, that the vast majority of bugs were found, and also possible, and probably more likely, that this project has barely scratched the surface of the bugs that could be found.

5.2 Evaluation Criteria

A fairly simple, but effective set of evaluation criteria have been chosen. When a bug is found three variables must be identified. These variables are the severity of the bug, the likelihood of encountering the bug, and how difficult it would be to work around the bug once a user is aware of it. Each of these variables are assigned a value of 1 to 3, with a bug with all variables as 1 being a non-severe, unlikely to encounter, and easy to work around bug. These scores are then summed to get a final value of how high the "quality" is of a bug, the higher the better for evaluation purposes.

For an example of evaluating a bug, A test with the following code is written.

```
C: {
   Num n = 1Num / 0Num // n has the value of 1/0.
   Debug(n) // This will print 1/0 to the console.
} // No error is thrown by this code.
```

Figure 5.1: 42 division by zero. This should not be allowed and we would expect it throw some sort of math error.

This is clearly a bug as division of any number by zero is undefined, and we would expect some kind of error to be thrown. In this case an error was not thrown and "1/0" was printed, and a bug report will be created, but firstly we need to figure out what values to assign to the evaluation variables. This bug isn't too severe, but it still could be a nasty surprise for a user so we'll assign it a 2 on the severity scale. The bug is pretty unlikely to be encountered on a regular basis, so it will be assigned a 1 on the likelihood scale, and it would be pretty simple to work around with some kind of check for a result equalling "1/0 so we'll assign it a 1 on the work around scale. This gives us an overall "quality" score of 4, so a slightly below medium quality bug, which seems fairly accurate.

TODO add some sort of diagram showing evaluation of a bug.
TODO add table of values.
TODO expand upon evaluation
TODO update bug overview.

5.3 Bug overview

Table 5.1: The categories and quantity of bugs found

Category	**Quantity**
Syntax (too liberal)	2
Syntax (too restrictive)	1
Type system (too liberal)	2
Type system (too restrictive)	1
Metaprogramming (too liberal)	0
Metaprogramming (too restrictive)	0
Reduction	3
Miscellaneous	5
Total quantity	**14**

6

Conclusions

6.1 Future Work

Ideas for future projects expanding upon this work or complementing include;

- A specific fuzzer for 42
- Wide scale user testing
- A transpiler to turn other code into 42 code
- Some sort of formal verification???

7

Work Done

7.1 Initial steps

The initial stages of the project consisted of familiarising myself with the 42 language and as I had no previous knowledge of the language this would prove to be extremely important. I started by reading the 42 tutorial [1]. After this I started to get some hands on experience by writing very basic and small 42 programs. Writing these programs uncovered a few small bugs but the primary value of them was learning how to use the language which would allow me to create more complex programs and push the boundaries of the language later on.

7.1.1 Bug areas

Once I had familiarised myself with the language, Dr Servetto suggested a few areas of the language where he thought it likely that it would be possible to find bugs.

- Parser
- Type system
- Metaprogramming
- Reduction
- Native interfacing

7.1.2 Bug types

Dr Servetto also laid out the criteria for what constituted a bug and these criteria are fairly simple and straightforward;

- A Java error is produced
- 42 behaviour does not match specifications

Java errors

The first criterion is fairly self-explanatory. If stderr reports a Java exception or error we classify it as a 42 bug as all errors or exceptions should be handled by the 42 compiler. This not true in all cases, and in sometimes a Java error being produced reveals a deeper problem. An example of this is in Figure 3.1. We would expect the code in Figure 3.1 to

throw some sort of type error as *s* is an immutable object as it is not prefaced with the *var* keyword. However instead of this expected behaviour, this code snippet will throw a Java AssertionError. If we look at the Java stack trace we can see that an assertion is violated in the type system. The purpose of this assertion is to make sure that a state that we should never reach (when behaviour is as expected) is not reached. Therefore we can come to the conclusion that since this assertion is violated we have reached a state that should not be reached. We can then come to the conclusion that we have an error that is not just 42 failing to handle its own error, it also shows there is an implementation error in the compiler.

```
C: {
   s = 12Num // s is immutable as variables are immutable by default.
   s := 13Num // Note Pascal style for updating variables.
   return ExitCode.normal()
}
```

Figure 7.1: The update immutable bug. This bug throws a Java AssertionError and is on the TODO list to be fixed.

Behaviour mismatch

This criterion is significantly more broad than the previously mentioned one. This includes 42 behaviour that does not match with the specifications laid out for 42 (Figure 3.2 is an example of this) [1], and behaviour that is not mentioned in the specifications (Figure 3.3 is an example of this).

```
C: {
   s = return ExitCode.normal()
}
```

Figure 7.2: Assigning an exit code to a variable. According to the specifications this should work correctly

```
C: {
   Num n = 1Num / 0Num // n has the value of 1/0.
   Debug(n) // This will print 1/0 to the console.
} // No error is thrown by this code.
```

Figure 7.3: 42 division by zero. This should not be allowed and we would expect it throw some sort of math error.

The behaviour for the snippet of code in figure 3.3 is not specifically defined in the 42 specifications [34]. Therefore we must err to our best judgment on a case by case basis. In this specific case it is clear that this is not how the code should behave, it is nonsensical to allow division by zero therefore this should result in both the specifications and implementation being changed.

```
class A {
   static void a() {
      int n = 1 / 0;
      System.out.println(n);
   }
}
```

Figure 7.4: Java division by zero. The equivalent code of figure 3.3. This will throw an ArithmeticException which is the behaviour we would expect in 42.

7.2 Testing

For this project two main types of tests were written. Firstly, various general tests that did not target any specific area of the compiler were written. This kind of test would generally be implementing simple, well known program in 42. Examples of this include finding the maximum value in an array or finding if an array contains a value (see Appendix A). These tests found roughly half the bugs (6) but made up around 70% of the total tests written (196). This means that compared to targeted tests which found 8 bugs, the more general tests have a lower chance of exposing a bug. Though it must be noted that this is skewed by the fact that many of the general tests were written when initially familiarising myself with the language.

After a potential bug was found I would attempt to minimise the code required to replicate bug then a bug report (an example of a bug report can been seen in Appendix B) would be sent to Dr Servetto who would give feedback on the bug, such as if it was already know, the likely cause, and how difficult it would be to fix. If a bug was of high priority Dr Servetto would fix it and the bug report could be closed. If it was not of a high enough priority it would be put in the "TODO list". This meant the bug would then be known and could be fixed in the future.

Some comments can be made about the quantity of bugs (see table 3.1) found over the course of this project. The first thing that stands out is that the total quantity of bugs found is not as high as hoped. This is primarily due to the absence of automated testing of the compiler, all testing was done manually. On average around 10 tests have been written for each bug found (142 tests written, 14 bugs found). An additional interesting point to note includes the total lack of metaprogramming bugs discovered. This is due to no tests having been written that use the 42 metaprogramming system. Reasons for this include not having very much experience with metaprogramming and not being needed in writing simple programs. Future testing of metaprogramming in 42 will require specifically tailored tests and more in depth research into metaprogramming. Another topic worthy of discussion of the high quantity of bugs in the "Miscellaneous" category. This category, as described earlier, includes all the bugs that were easily classified into the other categories or simply did not fit. Examples of this include the division by zero bug discussed earlier in this chapter (see figure 3.3). Therefore one could probably expect a high number of this category of bugs, especially due to the lack of automated testing so far.

7.3 Conclusion

A lesson that has been learnt over the course of this project is that compiler testing is very different when contrasted to more normal testing. Compiler bugs are considered to be diffi-

cult to find [10], and attempting to find bugs in the 42 compiler was no exception. Some bugs were found by programming fairly simple programs, but the majority were found through pushing the boundaries of what one might expect a programmer to do in the normal usage of a language. Examples of this include the "underscore bug". In 42 an underscore character is a valid variable name, and if an object that was passed to a method as a parameter was called "_" and "_" was not in scope when the method was called, a Java NullPointerException would be thrown (see Figure 3.5 for full code snippet). This is clearly unintentional behaviour and a bug in the 42 compiler.

```
A: {
   class method Any m(Any that) that // Any is supertype of all objects,
      like Object in Java.
}

B: {
   a = a.m(_) // Underscore is a valid variable name but there is no
      variable with this name in scope.
   return ExitCode.normal() // We need to declare an exit status on any
      program.
}
```

Figure 7.5: The underscore bug. This code would throw a NullPointerException in Java and it was subsequently fixed to produce the correct 42 error.

This project has shown, that while writing simple programs may find some fairly simple bugs, a far larger quantity of bugs will be found by specifically targeting different areas of the compiler or by writing unusual code. An additional reason to focus on these two methods is that many (but most certainly not all) of the bugs that could be found by writing simple or standard programs have already been found and fixed.

An advantage of working with the 42 compiler versus a C or C++ compiler is that a significant number of compiler bugs can be attributed to aggressive compiler optimisation [3][35][36][37]. At the current time optimisation is not a priority for the 42 compiler at the present time. A major reason for this is that some language features are still being implemented and numerous bugs are still being found, as can be seen by the results of this project. The lack of aggressive compiler optimisation meant that some fairly common classes of compiler bugs have not found at all. This mainly includes bugs that are present at higher levels of optimisation but not at lower levels. This is clearly seen in bug reports for *gcc* where bugs are found when a higher levels of optimisation flag is passed e.g. *-O3* and then the bugs are not found at lower level e.g. *-O0* or *-O1*.

After looking over the bugs found so far, Dr Servetto and I came up with eight different categories for the classification of bugs found so far. These categories are roughly equivalent to the areas where Dr Servetto thought it likely I would find bugs in the compiler as per the list at the start of this chapter. This list of categories can be seen in the list below.

- Syntax is too liberal
- Syntax is too restrictive
- Type system is too liberal

- Type system is too restrictive
- Metaprogramming is too liberal
- Metaprogramming is too restrictive
- Reduction
- Miscellaneous

Most of these categories are fairly self-explanatory but two of the categories require a brief explanation. Firstly a bug that came under the ”Metaprogramming is too liberal” category would generally mean that the code produced unexpected behaviour as a result of metaprogramming. An example of this would a metaprogramming operation O that takes well-typed code and produces non-well-typed code. Correct behaviour would result in O taking well-typed code and producing well-typed code. A further category that requires some clarification is the ”Reduction” category. This includes the 42 compiler producing wrong Java behaviour and the compiler producing non-well-typed code. Finally, the ”Miscellaneous” category includes all the bugs that did not fit nicely under any other category. Note that these categories are not final and could change in the future. Additionally some of the bugs uncovered over the course of this project do not fit nicely into a single category or would be hard to categorise accurately so are generally placed in the miscellaneous category.

8

Future Work

A significant amount of work has been accomplished so far in finding bugs in the 42 compiler. Although the quantity of bugs found is perhaps not as high as hoped, but the quality of bugs found, if all factors are considered, is high. One of the mains reason the quantity of bugs found is quite low is due to the fact that no automated testing tools have been thus far. Which means that the overall quantity of tests written so far is not very large. All tests have been written manually, and the majority of these manually written tests have been fairly general programs which have been written in a black-box testing style. These black-box tests did not take into account the compiler code or previous knowledge of the compiler. A minority of programs were written in a grey-box testing style, and these programs proved to be the most effective at finding bugs. They had a probability of 9.2% of finding a bug compared to the black-box style tests which had a probability of 3.1%. We can therefore conclude that the grey-box testing style was more effective when contrasted to the black-box testing style. A conclusion that can be drawn from these results is that a focus should probably be placed upon a grey-box testing style. It must be noted that this would not exclude the possibility of black-box tests being written in the future, as they could still be useful and black-box testing has found bugs so far.

The next step for this project is to incorporate automated testing tools such as fuzzers. These tools may potentially uncover a large amount of bugs by virtue of allowing far more tests to be created than would be possible with purely manually creating tests. It is a possibility that automated test may not be as effective as hoped in this scenario due to various factors discussed in Section 2.4.2 of this report. The possibility of this will be mitigated by still writing manual tests. Automated testing will take place in parallel with manual testing. The future manual testing will almost certainly take on a more grey-box testing strategy, as so far, it has proven to be both more effective at finding bugs on the whole, and more effective per test written.

Overall, the future work on this project will consist of more testing, specifically targeted tests with knowledge of the code of the compiler, as well as the use of automated testing tools and techniques [15] [18] [20] [21] [22] to increase the quantity of tests produced. The combination of high quality targeted testing, and more general automated tests will provide a significantly more rounded testing strategy than has been used so far over the course of this project.

Appendix A

Short 42 programs

```
Nums: Collections.vector(of: Num) // Define a vector of Num.
// Gets the max value in an array.
C: {
   class method Num max(Nums arr) {

      var Num max = Num"-9999" // Set a low number. Note no Num.Min_VALUE

      with e in arr.vals() ( // Like for(int e : arr)
         if (e > max) (
            max := e
         )
      )
      return max
   }
}
```

```
Nums: Collections.vector(of: Num)

C: {
   class method Bool contains(Nums arr, Num target) {
      with e in arr.vals() (
         if (e.equals(target)) (
            return Bool.true()
         )
      )
      return Bool.false()
   }
}
```

Appendix B

Sample bug report

VectorBlankVariable.L42

```
reuse L42.is/AdemTowel02
CacheAdamTowel02:Load.cacheTowel()

Nums : Collections.vector(of: Num)
Main: {
   Nums arr = Nums[_;_;_]
}
```

Expected behaviour: Some sort of 42 error stating that variable "_" is not in scope or cannot be found.

Actual behaviour: java.lang.NullPointerException

```
Caused by: java.lang.NullPointerException
        at newTypeSystem.AG.g(TIn.java:92)
        at newTypeSystem.TIn.g(TIn.java:1)
        at newTypeSystem.TsOperations.tsX(TsOperations.java:46)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:137)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$X.accept(ExpCore.java:318)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsMCall.tsMCall(TsMCall.java:73)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:141)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$MCall.accept(ExpCore.java:50)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsBlock.dsType(TsBlock.java:139)
        at newTypeSystem.TsBlock.dsType(TsBlock.java:163)
        at newTypeSystem.TsBlock.tsBlockBase(TsBlock.java:69)
        at newTypeSystem.TsBlock.tsBlock(TsBlock.java:28)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:142)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$Block.accept(ExpCore.java:408)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsMCall.tsMCall(TsMCall.java:73)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:141)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$MCall.accept(ExpCore.java:50)
```

Bibliography

[1] "42 - Metaprogramming as default - Tutorial." `http://l42.is/tutorial.xhtml`, 2018. [Online; accessed 19-May-2018].

[2] N. Amin and R. Tate, "Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 838–848, ACM, 2016.

[3] "Bug List." `https://gcc.gnu.org/bugzilla/buglist.cgi?chfield=\%5BBug\%20creation\%5D&chfieldfrom=7d`, 2018. [Online; accessed 07-June-2018].

[4] "The Unsound Playground." `https://lwn.net/Articles/342330/`, 2018. [Online; accessed 06-June-2018].

[5] "Known Issues for JDK 8." `http://www.oracle.com/technetwork/java/javase/8-known-issues-2157115.html`, 2018. [Online; accessed 08-June-2018].

[6] J. Harrison, "Formal verification at intel," in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 45–54, June 2003.

[7] E. W. Dijkstra, "Notes on Structured Programming - 249." circulated privately, apr 1970.

[8] "Mutability & Immutability." `http://web.mit.edu/6.005/www/fa15/classes/09-immutability/`, 2015. [Online; accessed 05-June-2018].

[9] F. Arnaboldi, "Exposing Hidden Exploitable Behaviors in Programming Languages Using Differential Fuzzing," p. 19.

[10] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[11] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, pp. 32–44, Dec. 1990.

[12] S. Blazy, Z. Dargaye, and X. Leroy, "Formal Verification of a C Compiler Front-end," in *Proceedings of the 14th International Conference on Formal Methods*, FM'06, (Berlin, Heidelberg), pp. 460–475, Springer-Verlag, 2006.

[13] S. Blazy and X. Leroy, "Mechanized Semantics for the Clight Subset of the C Language," *Journal of Automated Reasoning*, vol. 43, pp. 263–288, Oct 2009.

[14] "CompCert." `http://compcert.inria.fr`, 2018. [Online; accessed 26-May-2018].

[15] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," tech. rep., 1995.

[16] J. E. Forrester and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 2000.

[17] "american fuzzy lop." `http://lcamtuf.coredump.cx/afl/`, 2018. [Online; accessed 27-May-2018].

[18] "Afl-based fuzzing for Java." `https://github.com/isstac/kelinci`, 2018. [Online; accessed 28-May-2018].

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84, May 2007.

[20] "Javafuzz is a java classes fuzzer based on the the Java Reflection API." `https://github.com/cphr/javafuzz`, 2016. [Online; accessed 27-May-2018].

[21] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to Prioritize Test Programs for Compiler Testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 700–711, May 2017.

[22] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An Empirical Comparison of Compiler Testing Techniques," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 180–190, May 2016.

[23] A. S. Kossatchev and M. A. Posypkin, "Survey of compiler testing methods," *Programming and Computer Software*, vol. 31, pp. 10–19, Jan 2005.

[24] G. Hoglund, "Exploiting Parsing Bugs - Black Hat."

[25] T. Ostrand, *Black-Box Testing*. American Cancer Society, 2002.

[26] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[27] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Commun. ACM*, vol. 28, pp. 1054–1058, Oct. 1985.

[28] E. T. Parker, "Orthogonal Latin Squares," *Proceedings of the National Academy of Sciences*, vol. 45, no. 6, pp. 859–862, 1959.

[29] B. Homs, *Fundamentals of Software Testing*. Wiley & Sons, 2013.

[30] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," *SIGPLAN Not.*, vol. 48, pp. 197–208, June 2013.

[31] A. H. Watson and T. J. McCabe, "Structured testing: a software testing methodology using the cyclomatic complexity metric," 01 1996.

[32] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2017.

[33] M. E. Khan, F. Khan, *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[34] "42 - Metaprogramming as default." http://l42.is.xhtml, 2018. [Online; accessed 19-May-2018].

[35] R. Seacord, "Dangerous Optimizations and the Loss of Causality." http://www.eng.utah.edu/~cs5785/slides-f10/Dangerous+Optimizations.pdf, 2010.

[36] "Can compiler optimization introduce bugs." https://stackoverflow.com/questions/2722302/can-compiler-optimization-introduce-bugs, 2010. [Online; accessed 07-June-2018].

[37] "Fun with NULL pointers, part 1 [LWN.net]." https://lwn.net/Articles/342330/, 2009. [Online; accessed 07-June-2018].