Evaluation Report:
Implementation of a compiler for Ralang

Daniel Pacheco
School of Computing and Digital Technology
Faculty of Computing, Engineering and Built Environment
Birmingham City University
Daniel.Pacheco@mail.bcu.ac.uk

May 20, 2016

Contents

List of Tables

List of Figures

Listings

Abstract

The purpose of this report is to evaluate the design and implementation of Ralang, an educational programming language designed to simplify and compliment the teaching of functional programming concepts to Computer Science undergraduates.

The project lasted 12 consecutive weeks, involving research, design, implementation and testing, which included developing a product, a presentation and a viva of the overall project, a demonstration of the finished software and the final work: this report, which critically analysis and discusses the overall project.

1 Introduction

This evaluation report for the implementation of a compiler identifies and evaluates the successes and failures of the last 12 weeks of designing, developing and testing the product proposed in week 4. Part of the proposal is appended in appendix E and appendix G.

Section 2 analyses project proposal, specifically the aims and objectives, then it attempts to identify if the initial system requirements were met within the amount of time available. This section will also be looking at the successes and failures of the project as a whole, nevertheless an in-depth analysis of improvements and recommendations will only be discussed in section 5.

In the methodology, design and implementation section, we will briefly talk about some of the problems faced during the implementation and how the research replaced the designing stages and it affected the work flow of the project. The report covers some of the tools which were of use in the design and development stages: these tools were vital to the development and completion of Ralang.

Section 3 also mentions some of the most important problem we faced during the testing phase, nonetheless it's important to mention that some of these issues were found during the development process, others during the presentation and demonstration. An in-depth analysis of this was very important and will be discussed in section 5.

The testing section covers 2 very import types of testing used during the development of the project as well as a covering the evaluation of the results obtained from a questionnaire which were used to analyse the non-function requirements of Ralang. Most of the functional requirements will be analysed in section 3 and 5 therefore it wasn't the focal point of this section.

Until this point, the report has been preparing the material required to create a very critical and comprehensive analysis of the overview of the product and also the project. Section 5 proposes changes which will affect Ralang in the future. To create an in-depth discussion only a few important selected points will be discussed. In this section, the report will attempt to suggest ways of improving the methodologies, implementation or design, and also their approaches to each individual problem.

Section 5, may attempt to provide guidance to take on a similar project, or may attempt to suggest what other approaches might have been more suitable for a particular problem. Not only this but section 5 will be justifying reasons to why certain decisions had to be made, for instance why the testing methodology had to be adapted and why we were unable to use the proposed implementation methodology.

Part of section 5, will be reviewing the product and also the project management process. This will be based on the testing and evaluation, and will be concluding this section.

Section 6 will be looking at the highlights of this project by referring to the original aims, objectives and requirements which were added to appendix G. We will use this section to create an overview of the project as a whole and also making recommendations for further research. Most importantly ensuring the objectives were met to the highest standard possible.

2 Achievement of Objectives

The software part of this project, to design and implement a programming language, has been mostly successful, as section 4 will be able to demonstrate in more detail that the 3 main objectives were met within the time frame specified by the project proposal. For more details about the aims and objectives in the proposal see appendix E.

A small group of students were asked to look at the source code of a program written in Ralang and were asked to answer a questionnaire to find out if the syntax of the language is readable and easy to follow. The results of this questionnaire, as depicted in figure 2.1 shows that 3 out of 9 strongly agree that Ralang's syntax is readable and easy to follow, whilst 5 out of 9 agree that Ralang's syntax is readable and easy to follow, and 1 person does not have an opinion.

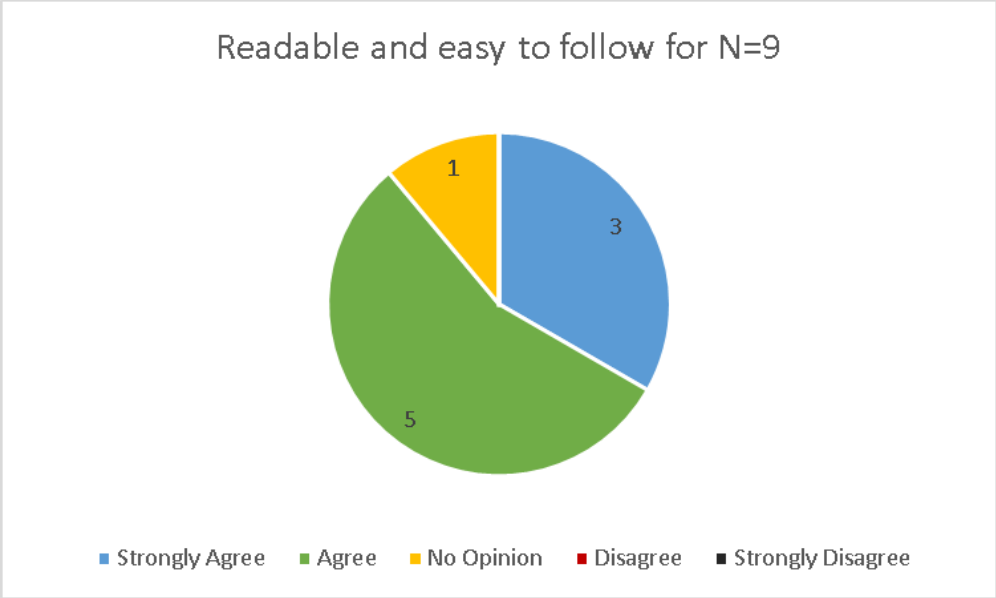


Figure 2.1: Survey results for Ralang appears to be readable and easy to follow

Ralang compiler is able to generate JVM bytecode and run programs written in Ralang both on Windows 10 32-bit and Ubuntu 15.10 64-bit. Seventeen different test cases were used to test the compiler working on these 2 operating systems. For more details about compiling and running these test cases see appendix B. There are test cases which were identified to fail, these test cases will be studied in greater detail in section 5, listed below are known reasons why some test cases are failing:

- Last In First Out (LIFO) Operand Stack limit set too low
- Local variables limit set too low
- JVM level stack overflow

The tests mentioned above can be classified as system testing or black-box testing, where given a certain input then the output could be predicted and tested against. This type of testing was part of the methodology used in development, more details in section 4.

Unit tests were also developed to test specific functionality within the compiler which are known to break very easily for example the tokens obtained from the parser, when the BNF file is changed. A total number of 72 different unit tests, to 1 of which is currently failing, as illustrated in figure 2.2.

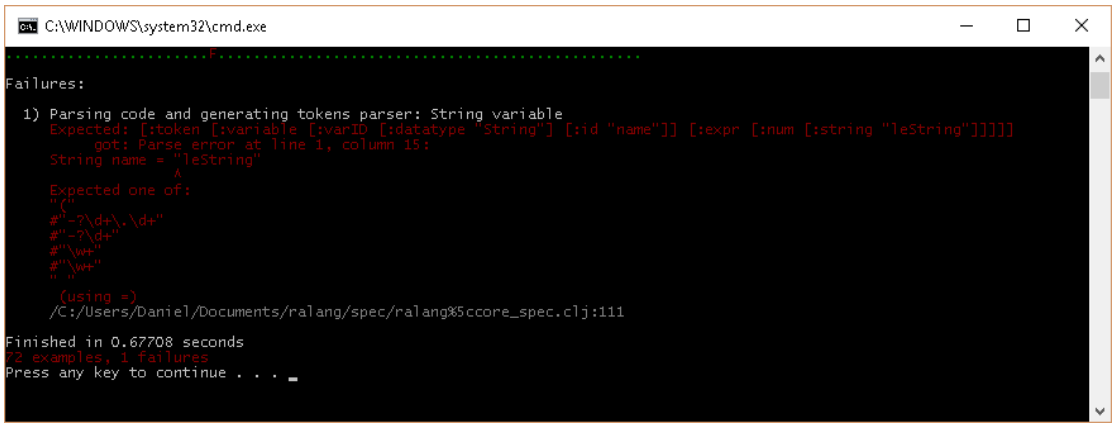


Figure 2.2: Running 72 different unit test on a Windows machine

The first 4 weeks of this project were used to plan and understand the feasibility of the project, by the fourth week a proposal was written formulating what system to target, as well as programming language choice to develop the compiler.

Language design was planned to take place between week 4 and 6. Then the implementation of the compiler would be between week 6 and 10. The last 2 weeks would be used to test the software and to ensure the language was consistent throughout. Due to lack of knowledge of compiler writing, the research had to take the place of design stage. Design of the language was worked out during the development stages instead.

There was some aspects of the proposal which weren't well thought out, for example language design was poorly researched and it meant to finish the project by the twelfth week, a new approach to the problem had to take place. Overall, the 3 main objectives were met, nevertheless the development methodologies had to be adaptive.

3 Methodology, Design and Implementation

As briefly discussed in appendix G.2 about the proposed development methodologies, Rapid Application Development (RAD) software development methodology would be suitable for a project this size limited by the amount of time given if had it been researched and if appropriate designs for the language had been done prior to development. The project had to adapt a new methodology called Test-Driven Development (TDD) in order to achieve the goals proposed.

Figure 3.1 shows the development stages of the compiler. Test cases are created to demonstrate and define the syntax of program's written in Ralang. One of the first test cases created was a hello world (testcases/helloworld.ra) which can be found in appendix B. In the test phase, it was also important to define what the outcome of that program after being executed should be. For instance, the hello world program was going to be able to display a message on the screen with the text "Hello World".

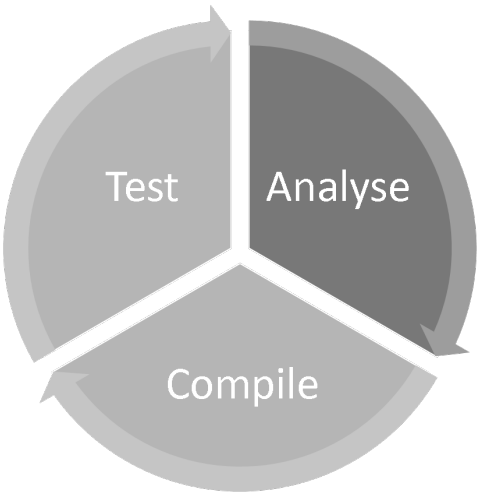


Figure 3.1: Test-driven development diagram showing development methodology

The next 2 stages are more challenging because it requires understanding of the JVM specification, the current state of the parser's grammar and to ensure the implementation is consistent throughout and that new test cases won't change the meaning of the language.

Analyse stage consisted of looking for ways to interpret source code and trying to understand what the JVM bytecode would have to look like in order to obtain the expected results from a particular test case. For that to work, different tools were used:

- Javap, is the Java class file disassembler. Sample Java programs could be compiled to a class file, and then JVM bytecode could be studied.
- Jasmin, is an assembler for the JVM, that allowed the creation of sample programs written in bytecode to be assembled into a class file to be executed.
- Bytecode Viewer, another Java class file disassembler with features to view Java code, to assist comparing the two high level languages, Java and Ralang.

Compiling stage involved generating bytecode from the sample test. This stage alone required several attempts until a workable version was produced. The testing part of development was only a small part of what the system required to be tested. Other testing methodologies had to be put in place to guarantee the system works as expected.

A short survey was carried out in week 11, this helped to get feedback from people who have never seen this programming language before. There was not much time for planning and designing a comprehensive questionnaire but it allowed people to give quick feedback based on some sample code presented to them. A comprehensive qualitative and quantitative analysis of the results will be presented in greater details in section 4. This questionnaire will be of use to further improve Ralang in the future.

Testing was overall challenging, considering the fact that to meet the essential requirements of the project, the compiler had to be capable of working out the correct output for various inputs.

Frequently during the development of this project, test cases which were able to compile properly previously, were unable to generate the correct bytecode after additions or changes in the system, and therefore the compiler would have to be fixed and all the prior test cases had to be tested against repeatedly throughout which slowed down the development of the project and that's when automated testing was introduced to tackle this issue.

Unit tests were mostly focused around the parser, because those were the changes most problematic. For example if the grammar of the BNF file had been changed to classify an Integer as a Number, and a Float was added to the grammar, but the Float was not added to be classified as a Number, the parser would produce the wrong set of tokens but the error would only be detected during the code generation.

Unit tests coverage are very extensive around the parser and the core of the compiler, nevertheless there is a lack of unit tests for the code generator. It would be useful to have more unit tests for the generator but for the purpose of this project it wouldn't be particularly important as Clojure would provide long stacktraces to allow the debugging of the software, as well as wrong generated bytecode could very easily be identified.

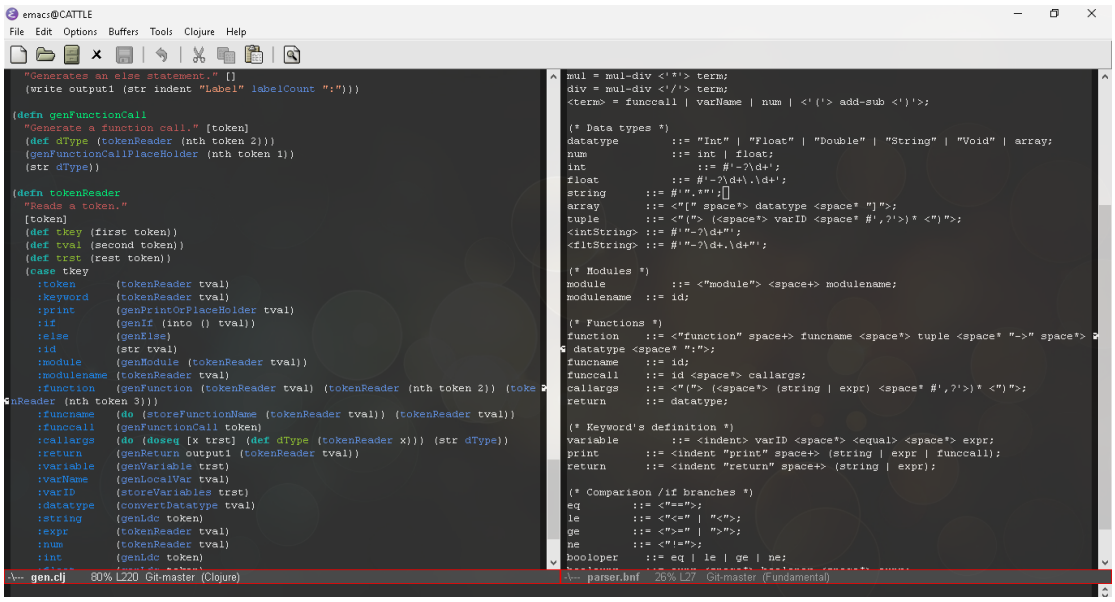


Figure 3.2: Emacs used as a development environment for developing Ralang

The project was developed on an Intel i3-2330M processor, with 4GB of RAM and on a 64-bit Windows 10 operating system, where certain Integrated Development Tools are unable to perform well such as Visual Studio and IntelliJ. Figure 3.2, shows Emacs text editor which allowed Ralang to be developed. It doesn't provide auto-completion or visual debuggers as expressed by Batsov (*Why Emacs?*), but it works well with Clojure and performance wise is fast for the

machine used to develop the system. Advanced techniques involved using keyboard shortcuts which increased performance immensely. GNU (*GNU Emacs Manuals Online*) was very useful for that purpose, and their software is very well documented.

Clojure, as briefly mentioned in appendix G, was a good choice of programming language to be used in this project because it is a functional programming language which ensures functions do not have side effects, data structure manipulation is incredibly powerful in Clojure and it already runs on the JVM, so for example, testing Ralang on a Linux machine works exactly the same way, after all the tools have been installed. Nevertheless learning a new language in such a small amount of time can be challenging, but there were many resources available online:

- StackOverflow.com had very helpful answers to questions which have been asked repeatedly over time. There was transferable knowledge from other programming languages, but it was more about learning how to do it the way Clojure compiler likes to interpret it. For instance, writing *cond (conditions)* or *case (switch cases)* were preferable to using *if*.
- Clojure.org has got a detailed documentation with a lot of sample code. In the early stages of the development this was the place to learn the language, mainly Clojure's rich set of immutable data structures and extensive ways to manipulate these structures.

The two mostly used data structures were maps and lists. The parser returned a map of tokens as a binary tree, the generator would walk the tree and generate bytecode from the tokens. Variables, functions and their corresponding values, were stored in lists, and Clojure was able to provide a good toolkit to perform operations on these data structures.

4 Testing

The testing stage of the project was briefly discussed in the previous section, mainly the functional requirements. This section will attempt to give an overview of the functional requirements and explore in more detail the non-function requirements instead.

4.1 Functional requirements

In section 3 it was discussed that testing was part of the development methodology, where a test case was created and then the implementation of the system resolved around that one scenario. After the system had been developed to work for that particular test case, testing every other scenario prior to that one was must, because making changes to the compiler may break some of the other parts which was previously working.

This way of testing was not very time efficient, so unit tests were developed to check for changes in the parser which should remain the same, but in a more automatic and time efficient way. So when a change takes place, the unit tests are able to detect those changes and warn of certain problems the compiler may have.

Section 5 will be evaluating and discussing these 2 different types of testing mentioned above and also the questionnaire results, in appendix H, in more detail.

4.2 Non-functional requirements

In this section, the report focuses on understanding the user needs rather than the requirements set in the proposal. Below maintainability, reliability and scalability will be analysed in greater detail how they are used to test these user requirements.

- **Maintainability**, is important when a project starts to grow, to allow further development to give users the changes they need and want in the system. For the maintainability of the compiler to improve, documentation would have to be also improved, as the users of this system would be relying on the latest information provided to them via the documentation.
- **Reliability**, ensures that after each change to the compiler, it remains to work the way that people already using it are accustomed to. This is very important in compiler writing, because if the syntax of the language keeps changing constantly, projects depending on the language would have to keep updating to follow the current standards of the language, in which case users would find a more stable programming language to run their services.
- **Scalability**, allows the product to grow. During the development of this project, we added the building blocks for this programming language, but left enough room for improvements: to create a strong and static data typing system, more functional than it currently is, code-generation optimisation and tail call recursion optimisations, would be prioritised.

There was only one other non-functional testing which involved asking a group of 9 people to look at the source code for a program written in Ralang code and to give their views. The results are shown in table J.2 in appendix H and are visually represented in figure 4.1.

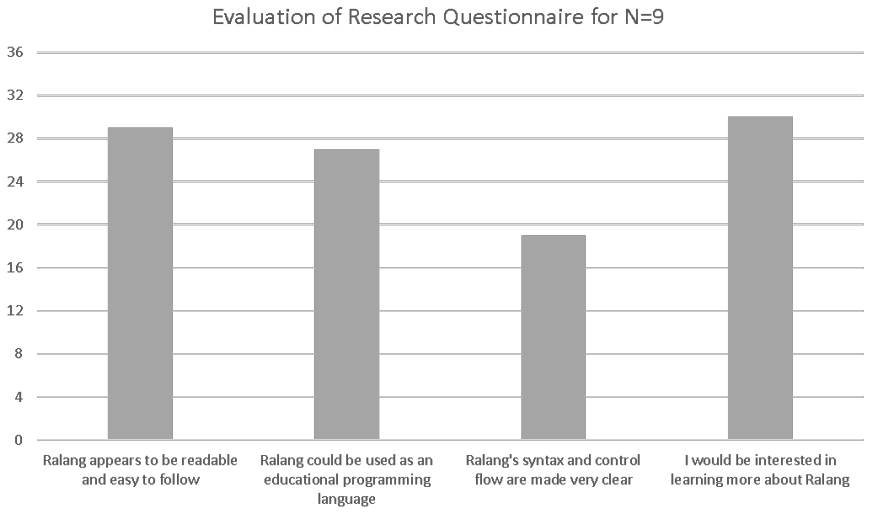


Figure 4.1: Evaluation of Research Questionnaire for a group of 9 students

The results shown in figure 4.1 are very optimistic, as over 80% of the participants, have said the syntax of Ralang is readable and easy to follow and also that they would like to learn more about the programming language. Section 5 will be looking in more detail into the quality of this research.

5 Analysis, Discussion and Critical Appraisal

Until now the report has discussed what has been achieved in the last 12 weeks on this project, from the research and design stages, to the implementation and testing. The report covered the major problems encountered along the way but has not made any suggestions for improvements as of yet. In this section we will try to address anything that might have been overlooked, critically evaluating what has been done and suggesting alternatives to these issues.

One of the problems which were briefly mentioned in section 2 and also during the product demonstration was the problems with stack and local variable limits being set too low and also JVM level stack overflow. The reason for the stack and local variables limits being set too low is because each frame in the JVM which has a LIFO operand stack and local variables, its maximum size need to be set at compile-time. (Lindholm and Yellin, 1999)

```
.method public static factorial(I)I
    .limit stack 50
    .limit locals 50
    ldc 2
    iload_0
    if_icmple Label1
    ldc 1
    ireturn
    Label1:
    iload_0
    iload_0
    ldc 1
    isub
    invokestatic factorial/factorial(I)I
    imul
    ireturn
.end method
```

Figure 5.1: An example of a factorial function written in JVM bytecode

During the implementation of the compiler, for quick prototyping, this operand stack and local variable limit was set to a static value of 50, as shown in figure 5.1, to give enough room to allow the test cases to run. Now this is a problem because if a program uses more space than it is allocated, it will still compile properly but the JVM will not be able to run the program. There are a few approaches to tackle this problem:

1. **Creating a tag** in the place of *.stack* and *.local* (similar to the way function calls were dealt with), count the number of instructions, pushes, stores, etc. and then in the second-pass of the code-generation, updating these tags with their appropriate values.

2. Generating **first-pass with default values of 0** being attributed to *.stack* and *.local*, calculating their true values during the second-pass of the code-generation and and lastly changing the default values to their correct values in a third-pass of the code-generation. With another pass, the compiler becomes slower because of the I/O stream speeds. It would be preferable to write/read as less as possible from the bytecode already generated.

3. **Calculating the size required in the first-pass** by the *.stack* and *.local* before it gets written to the file. This is possible, if the data of each frame gets appended to a list, and after being able to calculate the size of the *.stack* and *.local*, only then the frame (method) gets written to the file all at once. This requires more memory to store the frame (from *.method* to *.endmethod*), and only then writing to the file.

The first approach seems more reasonable than the other two. On one hand, the second option, requires too many reads and writes of the same file whilst the third option requires storing the

whole method in memory and this could be critical in certain systems.

The compiler already uses a two-pass code-generation, and the approach number 1 could take advantage of this. The size limit of *.stack* and *.local* would be calculated during the first-pass and would be written to the file in the second pass. This method would be saving memory in comparison with *third* approach, and reducing the number of times writing to the same file in comparison with *second* approach, increasing the overall performance of the compiler and its effectiveness.

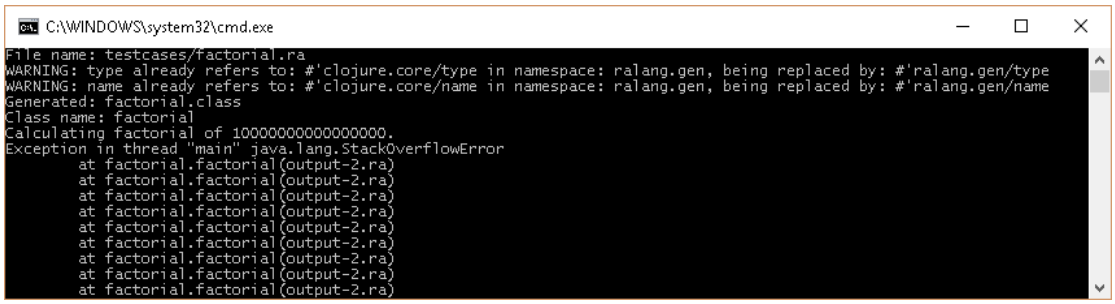


Figure 5.2: An example of a stack overflow

The other problem which was identified during the demonstration was a StackOverflowError, as shown in figure 5.2. The JVM throws a StackOverflowError if the computation in a thread requires a larger stack than the allowed amount, as described by Lindholm and Yellin, 1999. There are a few approaches to tackle this problem:

1. Use **Clojure's approach** to the same problem by using *java.lang.Long* to deal with integers and at the same time providing capabilities to use the JVM primitive data types as shown in table 5.1. Arguably though, if memory could be viewed as a critical in certain systems, using long to store a byte, short or int, may also cause some problems in the future. (Duey, 2013)

2. Use ***java.lang.Integer* by default**, and automatically change the data type to a *java.lang.Long* if the value is too large for an integer. This means the compiler will have to do extra checks and it also means we are moving away from a statically typed language, which would be one of the requirements for Ralang.

For the time being, the first approach is easier to implement, nevertheless it would still be important to allow the creation of variables with different data types, in a similar way that Clojure does it. By doing so, the static typing is not being compromised and there is no need for extra checks in the compiler.

Primitive data types	Start	End
Byte	-128	127
Short	–32,768	32,767
Int	–2,147,483,648	2,147,483,647
Long	–9,223,372,036,854,775,808	9,223,372,036,854,775,807

Table 5.1: JVM Numeric integral primitive data types excluding Char. Values obtained from Lindholm and Yellin, 1999

One last problem, which is important to mention and is not very obvious just by looking at figure 5.2 and table 5.1 is that even though Ralang generates an integer by default changing it from a *java.lang.Integer* to a *java.lang.Long*, would not solve the stack overflow problem. It would still end up in a stack overflow error, as shown in figure 5.3.

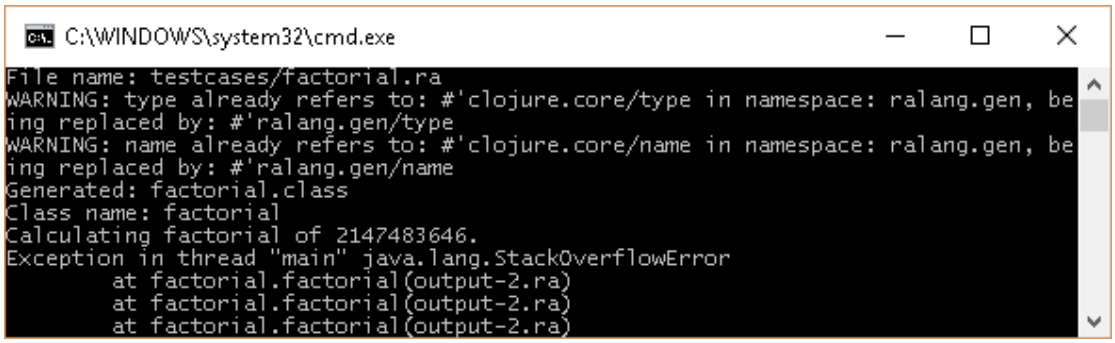


Figure 5.3: An example of an Int resulting in a stack overflow

The number (2147483646) used for this example is an integer still within the boundary limits of its type, nevertheless it is still causing a *java.lang.StackOverflowError*. This happens because a recursive function which is not being optimised by the compiler, for instance a simple *factorial(5)* calling factorial function in listing B.4 would generate similar results to the *Without Optimisation* in table 5.2:

Without optimisation	With optimisation
factorial(5) =5 * factorial(4) =5 * 4 * factorial(3) =5 * 4 * 3 * factorial(2) =5 * 4 * 3 * 2 * factorial(1) =5 * 4 * 3 * 2 * 1 =5 * 4 * 3 * 2 =5 * 4 * 6 =5 * 24 =120	factorial(5) =call_factorial(5, 1) =call_factorial(4, 3) =call_factorial(3, 6) =call_factorial(2, 24) =call_factorial(1, 120) =120

Table 5.2: Comparing with and without optimisation code.

So not only, the approaches mentioned above would have to be implemented, but also tail call recursion optimisation, to prevent recursive calls from generating large chains of recursive calls. Now the problem with the number (2147483646) will generate such a long tail it will blow the stack. Since the JVM does not implement tail call recursion, this would have to be implemented at the compiler level, a representation of this is shown in *With optimisation* in table 5.2 as described by Bruno, 2014, which was one of the suggestions for the Clojure compiler too.

Jelvis, 2015 also warns that various modern programming languages don't implement because of the legacy tools and software, others because they were short-term thinking and others simply because there is not enough pressure to implement tail call optimisations. For Ralang, this was a design problem for not wanting to use looping techniques such as *for* and *while* loops, on the other hand it means that implementing appropriate recursive calls optimisations are essential for Ralang.

The reason why Ralang has decided to stick to recursive calls only instead of *for* and *while* loops which are very common in modern programming languages, is because iteration over recursive calls would require the language to be in a mutable state, in which case it would be moving away from a purely functional programming language as intended for this project.

In section 3 and 4 it was covered 2 main types of testing and also a questionnaire which can be found in appendix H. It is important to mention that the design of the questionnaire is not thoroughly thought out and it was planned to remove as much ambiguity as possible,

nevertheless it is important to keep in mind that all participants were people from the same faculty and it may still contain ambiguous responses. Hence why it is important to not rely too much on this particular questionnaire but instead to discuss how we might have been able to improve it for a different study:

1. Allow people to try out the software for themselves for a short period of time, this may require some form of tutorial to begin with, and then to allow them answering a questionnaire of the same sort. This will give the correspondents enough time and experience with the language before they are asked for their opinion.

2. The questionnaire presented to these students, had a sample code in Ralang and also in C#. As C# was used at our university some students might have already created some sort of preconception about C#, and this might have affected the results. This links back to point no. 1, if the students were allowed to try the software before they answered the questionnaire, this might have affected the results.

3. Instead of a questionnaire, a simple test could be put in place, to see if the students actually learned how the programming language works, and based on these results then we could take the students feedback on board.

These could be some potential solutions if a new research was to take place. Nevertheless, even though the other 2 tests were much more accurate, there still was a lot of room for improvements, mainly with the unit testing, but also with the TDD, as can be shown in appendix B.

The system tests, were missing a lot of important features and it did not test for system failures. This is really important, because during the demonstration and viva of the project several problems were discussed which were not identified during the system testing. It's arguable that during the development phase, these system tests were used to develop the product and not exactly to test every possible case scenario, and that is the reason why test cases were developed in the first place.

Though, not even the unit tests can provide the most accurate results. The reason for this is because the coverage of the unit tests are very tiny for a project of this size. The unit tests coverage is about $^1/_5$ of the entire project. It does cover one of the most important parts which is the parser, which was one of the most difficult parts of the compiler to debug, nevertheless there is still a lot of ground to cover. The main reason why this was not completed was because there simply was not enough time.

In terms of achieving the main objectives, three of the most important challenges faced, was researching compiler writing, the time frame allocated to the project and being able to quickly adapt to a new implementation methodology. As mentioned previously in section 3, replacing the design process with the research of compiler design, meant that there was not enough time to plan Ralang's language, therefore the approach taken was to design test cases and make the compiler to work for those particular test cases.

The language slowly started to become very inconsistent throughout, and it meant re-thinking and re-implementing several parts. As a future reference, it would be very important to have the language design done first and peer-reviewed for inconsistencies and then implemented into the compiler, this would have saved half of the time, and the time lost could have been used improving the test cases, unit tests and looking into improving features in the language.

Non-functional requirements in section 4 is a very important part of this project but it's one of the most difficult ones to test. This essentially is supposedly testing the user requirements for the language, and at this point is very difficult to carry out such type of testing. One of the point which was not mentioned in section 4 was that carrying out the following types of non-functional requirements would be very important:

- **Platform compatibility** would mean testing the compiler in other Operating Systems, like other versions of Windows and Linux, as well as Mac OS, possibly also targeting the Android platform.
- **Documentation**, to provide the users with access to an in-depth documentation, guides, and how-to, sample code, etc. This is very important to allow the users to feel empowered, when they are stuck to enable them to find the information as quickly as possible.
- **Performance and response time**, as in how fast does it take to perform a large task. This would require testing the system under a lot of stress and seeing how it performs, in a way or another, it could be considered also a stress test.

One last non-functional requirement that should have been tested more thoroughly was the **extensibility and reusability** test. As shown in appendix C, the Java integration is something that would be very useful to integrate Ralang applications with Java applications. It would also be very useful to have it the other way around, and to allow the extensive Java Class Library to be imported into Ralang.

It was a great accomplishment to enable the Ralang compiler to create a *package* which then a Java program could import and make use of, as demonstrated in appendix C. The next step would be to allow the Ralang compiler to import classes from Java Class Library, but this would require focusing on the language's current issues before adding additional features.

6 Conclusion

The project design, implementation and testing followed the proposal (appendix E and G) in almost every aspect it could. The time frame was a bit off because of the research which had to be done in order to begin the design and implementation.

The questionnaire involved handing out printed questionnaires and talking to people directly about the project. It was intended for the participants to get a tutorial and then attempt to program in Ralang, before answering the questionnaire. Due to the amount of time this was not possible, but may be considered in the future.

It would be possible to use online surveys to get a lot more feedback, nevertheless for this project it was necessary to have a face-to-face conversation with people to make sure they understand what this project is all about and also to answer any questions. This process obviously is more time consuming but the results seem to be more genuine since the people who took part, seemed very engaged and wanting to know more about Ralang and also the project overall.

Similarly, the development methodologies had to be changed in order to finish the product within the time frame. This was specially important because the proposal did not take the time it would take to research the implementation, and it took too long. Using the TDD instead of the RAD, was not a terrible change, though in the future it may even be considered as a first option.

With the testing, we went beyond what was in the initial specification and developed additional unit tests to provide a more efficient way of testing and guaranteeing the software works each time after any changes. Unit tests helped greatly in developing the system, as prior to week 9 all the manual tests were becoming harder to maintain. It greatly contributed towards providing quick feedback loops and enough debugging information when anything stopped working.

The project management process, focused on getting the most basic test cases scenario to work first. The first test case scenario was one of the hardest because the JVM was a new system which had to be learnt, as well as Clojure, the programming language of choice to write the compiler, had to be learnt from the ground up also, and writing a compiler was no easy task mainly to learn it and develop one at the same time. There was a basic idea of how it all was going to work together to read a source file and compile it into a JVM class file, it is now represented in appendix D, and it is essentially the high-level design.

This design was to keep in mind that a few compiler writing techniques were to be skipped in order to have enough time to create a basic one that works well. For instance, the parser was created using InstaParse for Clojure and the code optimisations would be a step which ought to be skipped to keep the project within the time frame.

In the future, it would be great to work on optimised and specialised parser for Ralang and to have an optimised compiler which could generate optimised code for the JVM, and perform compile-time function execution.

References

Aho, A. V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811.

Batsov, B. *Why Emacs?* URL: http://batsov.com/articles/2011/11/19/why-emacs/ (visited on 05/15/2016).

Bruno, E. (2014). *Tail Call Optimization and Java*. URL: http://www.drdobbs.com/jvm/tail-call-optimization-and-java/240167044 (visited on 05/19/2016).

Chakravarty, M. M. T. and G. Keller (2004). "The risks and benefits of teaching purely functional programming in first year". In: *Journal of Functional Programming* 14 (01), pp. 113–123. ISSN: 1469-7653. DOI: 10.1017/S0956796803004805. URL: http://journals.cambridge.org/article_S0956796803004805.

Daly, T. (2011). "Minimizing to Maximize: An Initial Attempt at Teaching Introductory Programming Using Alice". In: *J. Comput. Sci. Coll.* 26.5, pp. 23–30. ISSN: 1937-4771. URL: http://ezproxy.bcu.ac.uk:2132/citation.cfm?id=1961574.1961578.

Duey, J. (2013). *Typed Clojure*. URL: http://www.clojure.net/2013/03/14/Typed-Clojure/ (visited on 05/17/2016).

GNU. *GNU Emacs Manuals Online*. URL: https://www.gnu.org/software/emacs/manual/ (visited on 05/15/2016).

Jayal, A. et al. (2011). "Python for teaching introductory programming: A quantitative evaluation". In: *Innovation in Teaching and Learning in Information and Computer Sciences* 10.1, pp. 86–90.

Jelvis, T. (2015). *Why is tail recursion optimisation not implemented in languages like Python, Ruby, and Clojure?* URL: https://www.quora.com/Why-is-tail-recursion-optimisation-not-implemented-in-languages-like-Python-Ruby-and-Clojure-Is-it-just-difficult-or-impossible (visited on 05/19/2016).

Kölling, M. (1999). "The problem of teaching object-oriented programming". In: *Journal of Object Oriented Programming* 11.8, pp. 8–15.

Krpan, D. and I. Bilobrk (2011). "Introductory programming languages in higher education". In: *MIPRO, 2011 Proceedings of the 34th International Convention*, pp. 1331–1336.

Lindholm, T. and F. Yellin (1999). *Java Virtual Machine Specification*. Java SE 8 edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201432943.

Nikula, U. et al. (2007). "Python and roles of variables in introductory programming: experiences from three educational institutions". In: *Journal of Information Technology Education* 6, pp. 199–214.

Vihavainen, A., J. Airaksinen, and C. Watson (2014). "A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success". In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. Glasgow, Scotland, United Kingdom: ACM, pp. 19–26. ISBN: 978-1-4503-2755-8. DOI: 10.1145/2632320.2632349. URL: http://ezproxy.bcu.ac.uk:2253/10.1145/2632320.2632349.

Watson, C. and F. W. Li (2014). "Failure Rates in Introductory Programming Revisited". In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. Uppsala, Sweden: ACM, pp. 39–44. ISBN: 978-1-4503-2833-3. DOI: 10.1145/2591708.2591749. URL: http://doi.acm.org/10.1145/2591708.2591749.

Yadin, A. (2011). "Reducing the Dropout Rate in an Introductory Programming Course". In: *ACM Inroads* 2.4, pp. 71–76. ISSN: 2153-2184. DOI: 10.1145/2038876.2038894. URL: http://ezproxy.bcu.ac.uk:2253/10.1145/2038876.2038894.

Appendices

Appendix A Source code for Ralang

A.1 src/ralang/core.clj

```
(ns ralang.core
  (:use [clojure.algo.generic.functor :only (fmap)])
  (:use [clojure.java.io])
  (:require [clojure.string :as string])
  (:require [instaparse.core :as insta])
  (:require [clojure.walk :as walk])
  (:require [ralang.gen :as gen])
  (:gen-class))

(def parser
  "EBFN parser for RaLang."
  (insta/parser (clojure.java.io/resource "parser.bnf")))

(defn debugMessage
  "Shows debugging messages."
  [message]
  (println (string/join message)))

(defn removeEmptyLineOrComment
  "Find and remove empty lines and comments from the source code."
  [source]
  (def reComment #"\s*#.*")
  (def reEmptyLine #"^\s+$")
  (def sourceWithoutComment
    (clojure.string/replace source reComment ""))
  (clojure.string/replace sourceWithoutComment reEmptyLine ""))

(defn readSource
  "Read source file and remove empty lines."
  [file]
  (with-open [rdr (reader file)]
    (def source
      (for [line (line-seq rdr)]
        (removeEmptyLineOrComment line)))
    (doseq [x (remove empty? source)]
      (def parse (parser x))
      (gen/tokenReader parse)))
  (gen/genEndMethod)
  (gen/genOutput2))

(defn -main
  "RaLang's main function."
  [& args]
  (if (= (count args) 0) (throw (Exception. "Wrong number of arguments.")))
  (def fileName (nth args 0))
  (cond
    (.exists (as-file fileName)) (readSource fileName)
    :else (throw (Exception. "Source file does not exist."))))
```

Listing A.1: "Source code for the main file of Ralang compiler"

A.2 src/ralang/gen.clj

```
(ns ralang.gen
  (:use [clojure.java.io])
  (:require [clojure.string :as string])
  (:gen-class))

(declare tokenReader)
```

```
(declare genReturn)
(def output1         "output1")
(def output2         "output2")
(def indent          (string/join (repeat 4 " ")))
(def functionsTable (hash-map))
(def j_string        "Ljava/lang/String;")
(def j_print         "invokevirtual java/io/PrintStream/println")

(defn initOutput
  "Deletes output1 and output2 iff they already exist." []
  (.delete (clojure.java.io/file output1))
  (.delete (clojure.java.io/file output2)))

(defn write
  "Write bytecode to the output file."
  [file, bytecode]
  (spit file (str bytecode "\n") :append true)
  (def lastWritten bytecode))

(defn getType
  "Returns JVM type."
  [type]
  (case (str type)
    ":int"    (str "i")
    ":string" (str "a")
    (str type)))

(defn getMethodType
  "Returns JVM type for methods."
  [type]
  (case (str type)
    ":string" (str j_string)
    ":int"    (str "I")
    ":float"  (str "F")
    (str "")))

(defn convertDatatype
  "Converts datatype from ralang to JVM."
  [type]
  (cond
    (= (first type) :array) (str "[" (tokenReader (second type)))
    :else (case type
            "String" (str j_string)
            "Int"    (str "I")
            "Void"   (str "V")
            (str type))))

(defn storeArgs
  "Read function arguments and store them."
  [args]
  (def localCount 0)
  (def localVariables {})
  (doseq [x args]
    (def aType (tokenReader (nth x 1)))
    (def aName (tokenReader (nth x 2)))
    (cond
      (= aType j_string) (def aType "a"))
    (def localVariables (merge localVariables {aName (str (string/lower-case
    aType) "load_" localCount)}))
    (def localCount (inc localCount))))

(defn storeVariables
  "Read and store variable."
  [variables]
  (doseq [x variables]
    (def aType (tokenReader (nth x 1)))
    (def aName (tokenReader (nth x 2)))
    (def localVariables (merge localVariables {aName (str (string/lower-case
    aType) "load_" localCount)})))
```

```
74      (write output1 (str indent (string/lower-case aType) "store_" localCount))
75      (def localCount (inc localCount))))
76
77  (defn readTuple
78    "Function argument's tuple reader."
79    [tuple]
80    (storeArgs tuple)
81    (def args (apply str (map #(tokenReader (nth % 1)) tuple)))
82    (str "(" args ")"))
83
84  (defn genModule
85    "Generates a module."
86    [id]
87    (initOutput)
88    (write output1 (str ".class public " id))
89    (write output1 ".super java/lang/Object")
90    (write output1 ".method public <init>()V")
91    (write output1 "    aload_0")
92    (write output1 "    invokespecial java/lang/Object/<init>()V")
93    (def moduleName id))
94
95  (defn genEndMethod
96    "Generates a end method." []
97    (cond
98      (not (.contains lastWritten "return")) (do (genReturn output1 nil) (genReturn
       output1 "")))
99    (write output1 ".end method"))
100
101 (defn genFunction
102   "Generates a new function.
103     name  - Function's name.
104     ar    - Function's arguments.
105     rt    - Function's return type."
106   [name, ar, rt]
107   (cond
108     (not (= lastWritten ".end method")) (do (genReturn output1 nil) (genEndMethod
      )))
109   (write output1 (str ".method public static " name ar rt))
110   (write output1 "    .limit stack 50")
111   (write output1 "    .limit locals 50")
112   (def functionsTable (merge functionsTable {name (str ar rt)}))
113   (def labelCount 1))
114
115 (defn storeFunctionName
116   "Stores the current function name to enable us to create local variables."
117   [name]
118   (def functionName name))
119
120 (defn genReturn
121   "Generates the ending of the function.
122     rt    - Function's return type."
123   [file, rt]
124   (def returnType (getType rt))
125   (write file (str indent returnType "return")))
126
127 (defn genFunctionCallPlaceHolder
128   "Generates a placeholder for function calls."
129   [token]
130   (def fName (tokenReader token))
131   (write output1 (str "->fc" fName))
132   fName)
133
134 (defn genOutput2
135   "Generates:
136     Function calls.
137     Missing print statements." []
138   (with-open [rdr (reader output1)]
139     (doseq [line (line-seq rdr)]
140       (def oPart (split-at 4 line))
```

```
141       (def oCode (apply str (first oPart)))
142       (def oArgs (apply str (second oPart)))
143       (case oCode
144         "->fc" (write output2 (str indent "invokestatic " moduleName "/" oArgs (
      first (map functionsTable [oArgs]))))
145         "->pr" (write output2 (str indent j_print "(" (second (string/split (
      first (map functionsTable [oArgs])) #"\(.*\)")) ")V"))
146         (write output2 line))))
147   ; The following checks for proper ending to a simple program with only 1 main
      function.
148   ; It checks if the last line ends in ".end method"
149   (with-open [rdr (reader output2)]
150     (cond
151       (not (= (last (line-seq rdr)) ".end method")) (genReturn output2 nil))))
152
153 (defn genLdc
154   "Generates a LDC. Returns type (string, int)."
155   [content]
156   (write output1 (str "    ldc " (second content)))
157   (first content))
158
159 (defn genPrintOrPlaceHolder
160   "Generates a print statement or a placeholder for a print statement."
161   [content]
162   (write output1 "    getstatic java/lang/System/out Ljava/io/PrintStream;")
163   (comment (cond
164     (= (first content) :funccall) (genFunctionCallArgs (rest (nth content 2)))))
165   (def type (tokenReader content))
166   (case (str (first content))
167     ":funccall" (write output1 (str "->pr" (second (second content))))
168     (write output1 (str indent j_print "(" (getMethodType type) ")V"))))
169
170 (defn genArithmetic
171   "Generates an arithmetic expression for a particular type."
172   [arith, numbers]
173   (tokenReader (first numbers))
174   (def type (tokenReader (second numbers)))
175   (def dType (getType type))
176   (write output1 (str indent dType arith))
177   (str type))
178
179 (defn genVariable
180   "Reads and stores a variable."
181   [variable]
182   (def name (conj () (first variable)))
183   (def value (second variable))
184   ; Value is the expression of the variable which gets calculated before
      assignment.
185   (tokenReader value)
186   (storeVariables name))
187
188 (defn genLocalVar
189   "Gets a local variable by name."
190   [variable]
191   (def expression (map localVariables [variable]))
192   (write output1 (str indent (first expression)))
193   (first (first expression)))
194
195 (defn genIf
196   "Generates an if statement."
197   [token]
198   (tokenReader (nth token 0)) ; left expression
199   (tokenReader (nth token 2)) ; right expression
200   ; bool operation, eg. eq, ge, le, ne
201   (def boolOperation (apply str (rest (str (first (second (nth token 1)))))))
202   (write output1 (str indent "if_icmp" boolOperation " Label" labelCount)))
203
204 (defn genElse
205   "Generates an else statement." []
```

```
206   (write output1 (str indent "Label" labelCount ":")))
207
208 (defn genFunctionCall
209   "Generate a function call." [token]
210   (def dType (tokenReader (nth token 2)))
211   (genFunctionCallPlaceHolder (nth token 1))
212   (str dType))
213
214 (defn tokenReader
215   "Reads a token."
216   [token]
217   (def tkey (first token))
218   (def tval (second token))
219   (def trst (rest token))
220   (case tkey
221     :token      (tokenReader tval)
222     :keyword    (tokenReader tval)
223     :print      (genPrintOrPlaceHolder tval)
224     :if         (genIf (into () tval))
225     :else       (genElse)
226     :id         (str tval)
227     :module     (genModule (tokenReader tval))
228     :modulename (tokenReader tval)
229     :function   (genFunction (tokenReader tval) (tokenReader (nth token 2)) (
      tokenReader (nth token 3)))
230     :funcname   (do (storeFunctionName (tokenReader tval)) (tokenReader tval))
231     :funccall   (genFunctionCall token)
232     :callargs   (do (doseq [x trst] (def dType (tokenReader x))) (str dType))
233     :return     (genReturn output1 (tokenReader tval))
234     :variable   (genVariable trst)
235     :varName    (genLocalVar tval)
236     :varID      (storeVariables trst)
237     :datatype   (convertDatatype tval)
238     :string     (genLdc token)
239     :expr       (tokenReader tval)
240     :num        (tokenReader tval)
241     :int        (genLdc token)
242     :float      (genLdc token)
243     :double     (genLdc token)
244     :tuple      (readTuple trst)
245     :add        (genArithmetic "add" trst)
246     :sub        (genArithmetic "sub" trst)
247     :mul        (genArithmetic "mul" trst)
248     :div        (genArithmetic "div" trst)
249     token))
```

Listing A.2: "Source code for Ralang's token generator"

A.3 spec/ralang/core_spec.clj

```
1  (ns ralang.core-spec
2      (:require [speclj.core :refer :all])
3      (:require [ralang.core :refer :all])
4      (:require [ralang.gen :refer :all]))
5
6  (defn specljTest [] true)
7
8  (describe "Unit tests"
9      (it "Tests if Speclj works properly."
10         (should specljTest)))
11
12 (describe "Initialise compiler"
13     (it "-main: With 0 arguments"
14         (should-throw (-main))))
15     (it "-main: With 1 argument and a file that does not exist"
16         (should-throw (-main "does-not-exist.ra")))
17     (it "-main: With 1 argument and a file that exists"
```

```
18          (should= nil (-main "testcases/factorial.ra")))
19
20  (describe "Read source file"
21      (it "readSource: Reads a file that exists"
22          (should= nil (readSource "testcases/factorial.ra"))))
23
24  (describe "Remove empty line or comment"
25      (it "removeEmptyLineOrComment: A comment"
26          (should= "" (removeEmptyLineOrComment "# Comment")))
27      (it "removeEmptyLineOrComment: Code and a comment"
28          (def sampleCode "print 5")
29          (should= sampleCode
30              (removeEmptyLineOrComment (str sampleCode "# Comment"))))
31      (it "removeEmptyLineOrComment: 1 space"
32          (should= "" (removeEmptyLineOrComment " ")))
33      (it "removeEmptyLineOrComment: Empty string"
34          (should= "" (removeEmptyLineOrComment ""))))
35
36  (describe "Parsing code and generating tokens"
37      (it "parser: An integer"
38          (should= [:token [:keyword [:expr [:num [:int "5"]]]]] (parser "5")))
39      (it "parser: A float"
40          (should= [:token [:keyword [:expr [:num [:float "3.14"]]]]] (parser
        "3.14")))
41      (it "parser: Print integer"
42          (should= [:token [:keyword [:print [:expr [:num [:int "20"]]]]]] (parser
        "print 20")))
43      (it "parser: Print float"
44          (should= [:token [:keyword [:print [:expr [:num [:float "129.34"]]]]]] (
        parser "print 129.34")))
45      (it "parser: Print string"
46          (should= [:token [:keyword [:print [:string "\"Hello World\""]]]] (parser
         "print \"Hello World\"")))
47      (it "parser: Print result of adding two integer"
48          (should= [:token [:keyword [:print [:expr [:add [:num [:int "50"]] [:num
        [:int "25"]]]]]]] (parser "print 50+25")))
49      (it "parser: Print result of subtracting two integer"
50          (should= [:token [:keyword [:print [:expr [:sub [:num [:int "50"]] [:num
        [:int "25"]]]]]]] (parser "print 50-25")))
51      (it "parser: Print result of multiplying two integer"
52          (should= [:token [:keyword [:print [:expr [:mul [:num [:int "50"]] [:num
        [:int "25"]]]]]]] (parser "print 50*25")))
53      (it "parser: Print result of dividing two integer"
54          (should= [:token [:keyword [:print [:expr [:div [:num [:int "50"]] [:num
        [:int "2"]]]]]]] (parser "print 50/2")))
55      (it "parser: Print result of adding and multiplying"
56          (def e [:token [:keyword [:print [:expr [:add [:num [:int "50"]] [:mul [:
        num [:int "10"]] [:num [:int "2"]]]]]]]])
57          (should= e (parser "print 50+10*2")))
58      (it "parser: Print result of adding, multiplying and subtracting"
59          (def e [:token
60                  [:keyword
61                   [:print
62                    [:expr
63                         [:sub
64                              [:add
65                                   [:num [:int "50"]]
66                                   [:mul
67                                        [:num [:int "10"]]
68                                        [:num [:int "2"]]]]
69                              [:num [:int "30"]]]]]]]])
70          (should= e (parser "print 50+10*2-30")))
71      (it "parser: Print result of adding, multiplying, subtracting and dividing"
72          (def e [:token
73                  [:keyword
74                   [:print
75                    [:expr
76                         [:sub
77                              [:add
```

```
78                              [:div
79                                  [:num [:int "50"]]
80                                  [:num [:int "2"]]]
81                              [:mul
82                                  [:num [:int "10"]]
83                                  [:num [:int "2"]]]]
84                          [:num [:int "30"]]]]]]])
85      (should= e (parser "print 50/2+10*2-30")))
86  (it "parser: Print result of adding, multiplying, subtracting, dividing and
    parentheses"
87      (def e [:token
88              [:keyword
89              [:print
90              [:expr
91                  [:sub
92                      [:div
93                          [:mul
94                              [:num [:int "2"]]
95                              [:add
96                                  [:num [:int "20"]]
97                                  [:num [:int "30"]]]]
98                          [:num [:int "5"]]]
99                      [:num [:int "5"]]]]]]])
100     (should= e (parser "print 2*(20+30)/5-5")))
101 (it "parser: New module"
102     (should= [:token [:module [:modulename [:id "NewModule"]]]] (parser "
    module NewModule")))
103 (it "parser: Int variable"
104     (def e [:token [:variable [:varID [:datatype "Int"] [:id "age"]] [:expr
    [:num [:int "55"]]]]])
105     (should= e (parser "Int age = 55")))
106 (it "parser: Float variable"
107     (def e [:token [:variable [:varID [:datatype "Float"] [:id "speed"]] [:
    expr [:num [:float "32.5"]]]]])
108     (should= e (parser "Float speed = 32.5")))
109 (it "parser: String variable"
110     (def e [:token [:variable [:varID [:datatype "String"] [:id "name"]] [:
    expr [:num [:string "leString"]]]]])
111     (should= e (parser "String name = \"leString\"")))
112 (it "parser: New function with 0 args and returns Void"
113     (def e [:token [:function [:funcname [:id "leFunction"]] [:tuple] [:
    datatype "Void"]]])
114     (should= e (parser "function leFunction() -> Void:")))
115 (it "parser: New function with 0 args and returns Int"
116     (def e [:token [:function [:funcname [:id "leFunction"]] [:tuple] [:
    datatype "Int"]]])
117     (should= e (parser "function leFunction() -> Int:")))
118 (it "parser: New function with 0 args and returns Float"
119     (def e [:token [:function [:funcname [:id "leFunction"]] [:tuple] [:
    datatype "Float"]]])
120     (should= e (parser "function leFunction() -> Float:")))
121 (it "parser: New function with 0 args and returns String"
122     (def e [:token [:function [:funcname [:id "leFunction"]] [:tuple] [:
    datatype "String"]]])
123     (should= e (parser "function leFunction() -> String:")))
124 (it "parser: New function with 1 arg and returns Void"
125     (def e [:token
126             [:function
127             [:funcname
128                 [:id "leFunction"]]
129                 [:tuple
130                     [:varID [:datatype "Int"] [:id "a"]]]
131                 [:datatype "Void"]]])
132     (should= e (parser "function leFunction(Int a) -> Void:")))
133 (it "parser: New function with 1 arg and returns Int"
134     (def e [:token
135             [:function
136             [:funcname
137                 [:id "leFunction"]]
```

```
138                     [:tuple
139                         [:varID [:datatype "Int"] [:id "a"]]]
140                     [:datatype "Int"]]])
141         (should= e (parser "function leFunction(Int a) -> Int:")))
142     (it "parser: New function with 1 arg and returns Float"
143         (def e [:token
144                 [:function
145                 [:funcname
146                     [:id "leFunction"]]
147                     [:tuple
148                         [:varID [:datatype "Int"] [:id "a"]]]
149                 [:datatype "Float"]]])
150         (should= e (parser "function leFunction(Int a) -> Float:")))
151     (it "parser: New function with 1 arg and returns String"
152         (def e [:token
153                 [:function
154                 [:funcname
155                     [:id "leFunction"]]
156                     [:tuple
157                         [:varID [:datatype "Int"] [:id "a"]]]
158                     [:datatype "String"]]])
159         (should= e (parser "function leFunction(Int a) -> String:")))
160     (it "parser: New function with 2 args and returns Void"
161         (def e [:token
162                 [:function
163                 [:funcname
164                     [:id "leFunction"]]
165                     [:tuple
166                         [:varID [:datatype "Int"] [:id "a"]]
167                         [:varID [:datatype "Int"] [:id "b"]]]
168                     [:datatype "Void"]]])
169         (should= e (parser "function leFunction(Int a, Int b) -> Void:")))
170     (it "parser: New function with 2 args and returns Int"
171         (def e [:token
172                 [:function
173                 [:funcname
174                     [:id "leFunction"]]
175                     [:tuple
176                         [:varID [:datatype "Int"] [:id "a"]]
177                         [:varID [:datatype "Int"] [:id "b"]]]
178                     [:datatype "Int"]]])
179         (should= e (parser "function leFunction(Int a, Int b) -> Int:")))
180     (it "parser: New function with 2 args and returns Float"
181         (def e [:token
182                 [:function
183                 [:funcname
184                     [:id "leFunction"]]
185                     [:tuple
186                         [:varID [:datatype "Int"] [:id "a"]]
187                         [:varID [:datatype "Int"] [:id "b"]]]
188                     [:datatype "Float"]]])
189         (should= e (parser "function leFunction(Int a, Int b) -> Float:")))
190     (it "parser: New function with 2 args and returns String"
191         (def e [:token
192                 [:function
193                 [:funcname
194                     [:id "leFunction"]]
195                     [:tuple
196                         [:varID [:datatype "Int"] [:id "a"]]
197                         [:varID [:datatype "Int"] [:id "b"]]]
198                     [:datatype "String"]]])
199         (should= e (parser "function leFunction(Int a, Int b) -> String:")))
200     (it "parser: Function call with 0 args"
201         (should= [:token [:keyword [:expr [:funccall [:id "leFunction"] [:
    callargs ]]]]] (parser "leFunction()")))
202     (it "parser: Function call with 1 arg"
203         (def e [:token [:keyword [:expr [:funccall [:id "leFunction"] [:callargs
    [:expr [:num [:int "5"]]]]]]]])
204         (should= e (parser "leFunction(5)")))
```

```
205     (it "parser: Function call with 2 args"
206         (def e [:token
207                 [:keyword
208                 [:expr
209                 [:funccall
210                     [:id "leFunction"]
211                     [:callargs
212                         [:expr [:num [:int "30"]]]
213                         [:expr [:num [:int "15"]]]]]]]]])
214         (should= e (parser "leFunction(30, 15)")))
215     (it "parser: Function call with arithmetic as 1 arg"
216         (def e [:token
217                 [:keyword
218                 [:expr
219                 [:funccall
220                     [:id "leFunction"]
221                     [:callargs
222                         [:expr
223                             [:add
224                                 [:num [:int "7"]]
225                                 [:num [:int "8"]]]]]]]]])
226         (should= e (parser "leFunction(7+8)")))
227     (it "parser: Function call with arithmetic as 1 arg and 1 variable"
228         (def e [:token
229                 [:keyword
230                 [:expr
231                 [:funccall
232                     [:id "leFunction"]
233                     [:callargs
234                         [:expr
235                             [:add
236                                 [:num [:int "7"]]
237                                 [:num [:int "8"]]]]
238                         [:expr
239                             [:varName "leVariable"]]]]]]])
240         (should= e (parser "leFunction(7+8, leVariable)")))
241     (it "parser: Function call with arithmetic as 1 arg, 1 variable and 1 string"
242         (def e [:token
243                 [:keyword
244                 [:expr
245                 [:funccall
246                     [:id "leFunction"]
247                     [:callargs
248                         [:expr
249                             [:add
250                                 [:num [:int "7"]]
251                                 [:num [:int "8"]]]]
252                         [:expr
253                             [:varName "leVariable"]]
254                         [:string "\"leString\""]]]]]])
255         (should= e (parser "leFunction(7+8, leVariable, \"leString\")")))
256     (it "parser: If Int less than another Int"
257         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "2"]]] [:
    booloper [:le]] [:expr [:num [:int "10"]]]]]]])
258         (should= e (parser "if 2 < 10:")))
259     (it "parser: If Int less than or equal to another Int"
260         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "3"]]] [:
    booloper [:le]] [:expr [:num [:int "9"]]]]]]])
261         (should= e (parser "if 3 <= 9:")))
262     (it "parser: If Int is equal to another Int"
263         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "4"]]] [:
    booloper [:eq]] [:expr [:num [:int "4"]]]]]]])
264         (should= e (parser "if 4 == 4:")))
265     (it "parser: If Int greater than or equal to another Int"
266         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "5"]]] [:
    booloper [:ge]] [:expr [:num [:int "4"]]]]]]])
267         (should= e (parser "if 5 >= 4:")))
268     (it "parser: If Int greater than another Int"
```

```
269         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "6"]]] [:
    booloper [:ge]] [:expr [:num [:int "3"]]]]]]])
270         (should= e (parser "if 6 > 3:")))
271     (it "parser: If Int is different from another Int"
272         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "7"]]] [:
    booloper [:ne]] [:expr [:num [:int "1"]]]]]]])
273         (should= e (parser "if 7 != 1:")))
274     (it "parser: If Int less than another Variable"
275         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "2"]]] [:
    booloper [:le]] [:expr [:varName "leVariable"]]]]]])
276         (should= e (parser "if 2 < leVariable:")))
277     (it "parser: If Int less than or equal to another Variable"
278         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "3"]]] [:
    booloper [:le]] [:expr [:varName "leVariable"]]]]]])
279         (should= e (parser "if 3 <= leVariable:")))
280     (it "parser: If Int is equal to another Variable"
281         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "4"]]] [:
    booloper [:eq]] [:expr [:varName "leVariable"]]]]]])
282         (should= e (parser "if 4 == leVariable:")))
283     (it "parser: If Int greater than or equal to another Variable"
284         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "5"]]] [:
    booloper [:ge]] [:expr [:varName "leVariable"]]]]]])
285         (should= e (parser "if 5 >= leVariable:")))
286     (it "parser: If Int greater than another Variable"
287         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "6"]]] [:
    booloper [:ge]] [:expr [:varName "leVariable"]]]]]])
288         (should= e (parser "if 6 > leVariable:")))
289     (it "parser: If Int is different from another Variable"
290         (def e [:token [:keyword [:if [:boolexpr [:expr [:num [:int "7"]]] [:
    booloper [:ne]] [:expr [:varName "leVariable"]]]]]])
291         (should= e (parser "if 7 != leVariable:")))
292     (it "parser: If Variable less than another Int"
293         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:le]] [:expr [:num [:int "10"]]]]]]])
294         (should= e (parser "if leVariable < 10:")))
295     (it "parser: If Variable less than or equal to another Int"
296         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:le]] [:expr [:num [:int "9"]]]]]]])
297         (should= e (parser "if leVariable <= 9:")))
298     (it "parser: If Variable is equal to another Int"
299         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:eq]] [:expr [:num [:int "4"]]]]]]])
300         (should= e (parser "if leVariable == 4:")))
301     (it "parser: If Variable greater than or equal to another Int"
302         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ge]] [:expr [:num [:int "4"]]]]]]])
303         (should= e (parser "if leVariable >= 4:")))
304     (it "parser: If Variable greater than another Int"
305         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ge]] [:expr [:num [:int "3"]]]]]]])
306         (should= e (parser "if leVariable > 3:")))
307     (it "parser: If Variable is different from another Int"
308         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ne]] [:expr [:num [:int "1"]]]]]]])
309         (should= e (parser "if leVariable != 1:")))
310     (it "parser: If Variable less than another Variable"
311         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:le]] [:expr [:varName "raVariable"]]]]]])
312         (should= e (parser "if leVariable < raVariable:")))
313     (it "parser: If Variable less than or equal to another Variable"
314         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:le]] [:expr [:varName "raVariable"]]]]]])
315         (should= e (parser "if leVariable <= raVariable:")))
316     (it "parser: If Variable is equal to another Variable"
317         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:eq]] [:expr [:varName "raVariable"]]]]]])
318         (should= e (parser "if leVariable == raVariable:")))
319     (it "parser: If Variable greater than or equal to another Variable"
```

```
320         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ge]] [:expr [:varName "raVariable"]]]]]])
321         (should= e (parser "if leVariable >= raVariable:")))
322     (it "parser: If Variable greater than another Variable"
323         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ge]] [:expr [:varName "raVariable"]]]]]])
324         (should= e (parser "if leVariable > raVariable:")))
325     (it "parser: If Variable is different from another Variable"
326         (def e [:token [:keyword [:if [:boolexpr [:expr [:varName "leVariable"]]
    [:booloper [:ne]] [:expr [:varName "raVariable"]]]]]])
327         (should= e (parser "if leVariable != raVariable:")))
328     (it "parser: Else"
329         (should= [:token [:keyword [:else]]] (parser "else:")))
330     (it "parser: Return an Int"
331         (should= [:token [:keyword [:return [:expr [:num [:int "68"]]]]]] (parser
     "return 68")))
332     (it "parser: Return an Float"
333         (should= [:token [:keyword [:return [:expr [:num [:float "68.8"]]]]]] (
    parser "return 68.8")))
334     (it "parser: Return an String"
335         (should= [:token [:keyword [:return [:string "\"leString\""]]]] (parser "
    return \"leString\"")))
336     (it "parser: Return the result of adding 2 Ints together"
337         (should= [:token [:keyword [:return [:expr [:add [:num [:int "65"]] [:num
     [:int "9"]]]]]]] (parser "return 65+9")))
338     (it "parser: Return the result of function call"
339         (def e [:token
340                 [:keyword
341                 [:return
342                 [:expr
343                     [:mul
344                         [:varName "n"]
345                         [:funccall [:id "factorial"]
346                             [:callargs
347                                 [:expr
348                                     [:sub
349                                         [:varName "n"]
350                                         [:num [:int "1"]]]]]]]]]])
351         (should= e (parser "return n*factorial(n-1)"))))
352
353 (run-specs)
```

Listing A.3: "Source code for Ralang's unit tests"

A.4 resources/parser.bnf

```
1  token    ::= keyword  | module   | function | variable;
2  keyword      ::= print    | return   | if       | else     | expr;
3
4  (* Building blocks *)
5  space    ::= " ";
6  indent       ::= #'\s{4}'*;
7  equal        ::= "=";
8  id         ::= #'\w+';
9  varName      ::= #'\w+';
10 varID        ::= datatype <space+> id;
11
12 (* Maths expressions - InstaParse example *)
13 expr = add-sub;
14 <add-sub> = mul-div | add | sub;
15 add = add-sub <'+'> mul-div;
16 sub = add-sub <'-'> mul-div;
17 <mul-div> = term | mul | div;
18 mul = mul-div <'*'> term;
19 div = mul-div <'/'> term;
20 <term> = funccall | varName | num | <'('> add-sub <')'>;
21
```

```
(* Data types *)
datatype  ::= "Int" | "Float" | "Double" | "String" | "Void" | array;
num       ::= int | float;
int       ::= #'-?\d+';
float     ::= #'-?\d+\.\d+';
string    ::= #'".*"';
array     ::= <"[" space*> datatype <space* "]">;
tuple     ::= <"("> (<space*> varID <space* #',?'>)* <")">;
<intString> ::= #"-?\d+"';
<fltString> ::= #"-?\d+.\d+"';

(* Modules *)
module    ::= <"module"> <space+> modulename;
modulename  ::= id | #"\w+.\w+";

(* Functions *)
function    ::= <"function" space+> funcname <space*> tuple <space* "->" space*>
    datatype <space* ":">;
funcname    ::= id;
funccall    ::= id <space*> callargs;
callargs    ::= <"("> (<space*> (string | expr) <space* #',?'>)* <")">;
return      ::= datatype;

(* Keyword's definition *)
variable  ::= <indent> varID <space*> <equal> <space*> expr;
print       ::= <indent "print" space+> (string | expr | funccall);
return      ::= <indent "return" space+> (string | expr);

(* Comparison /if branches *)
eq        ::= <"==">;
le        ::= <"<=" | "<">;
ge        ::= <">=" | ">">;
ne        ::= <"!=">;
booloper    ::= eq | le | ge | ne;
boolexpr    ::= expr <space*> booloper <space*> expr;
if          ::= <indent "if" space+> boolexpr <space* ":">;
else        ::= <indent "else:">;
```

Listing A.4: "Ralang's compiler BNF context-free grammar"

Appendix B Running test cases

B.1 Sample program's Source Code

B.2 resources/parser.bnf

```
module callhello

function main([String] args) -> Void:
    print message("Hello World") # Call 'message' and print output

# A function that takes a string and returns a string
function message(String text) -> String:
    return text
```

Listing B.1: "Program written in Ralang: testcases/callhello.ra"

```
module compare

function main([String] args) -> Void:
    print compare(2016, 2016)
    print compare(2015, 2017)

function compare(Int year1, Int year2) -> String:
    if year1 == year2:
        return "Same year."
    else:
        return "Not same year."
```

Listing B.2: "Program written in Ralang: testcases/compare.ra"

```
module complex

function main([String] args) -> Void:
    print maths()

function maths() -> Int:
    return 10*(2*3-2*2)/2+50
```

Listing B.3: "Program written in Ralang: testcases/complex.ra"

```
module factorial

function main([String] args) -> Void:
    print "Calculating factorial of 5."
    print factorial(5)
    # 120

function factorial(Int n) -> Int:
    # Calculates the factorial of n
    if n < 2:
        return 1
    else:
        return n*factorial(n-1)
```

Listing B.4: "Program written in Ralang: testcases/factorial.ra"

```
module funccall

function main([String] args) -> Void:
    print sum()

function sum() -> Int:
    return 5+(10-2)/4*10
```

Listing B.5: "Program written in Ralang: testcases/funccall.ra"

```
module helloworld

function main([String] args) -> Void:
    print "Hello World"
```

Listing B.6: "Program written in Ralang: testcases/helloworld.ra"

```
module legal

function main([String] args) -> Void:
    print legal(17)
    print legal(18)
    print legal(19)

function legal(Int age) -> String:
    if age < 18:
        return "Not legal."
    else:
        return "Legal."
```

Listing B.7: "Program written in Ralang: testcases/legal.ra"

```
module mulcomp

function main([String] args) -> Void:
    print compare(2016, 2016)
    # "Not same year."

    print compare(2015, 2017)
    # "Same year."

    print notCompare(2016, 2016)
    # "Not same year."

    print notCompare(2015, 2017)
    # "Same year."

function notCompare(Int year1, Int year2) -> String:
    if year1 != year2:
        return "Not same year."
    else:
        return "Same year."

function compare(Int year1, Int year2) -> String:
    if year1 == year2:
        return "Same year."
    else:
        return "Not same year."
```

Listing B.8: "Program written in Ralang: testcases/mulcomp.ra"

```
module mulfuncs

function main([String] args) -> Void:
    print hello()
    print someNumber()

function hello() -> String:
    return "Hello World!"

function someNumber() -> Int:
    return 20
```

Listing B.9: "Program written in Ralang: testcases/mulfuncs.ra"

```
module mulhello

function main([String] args) -> Void:
    print "Multi line hello world!"
    print "Hi there."
```

Listing B.10: "Program written in Ralang: testcases/mulhello.ra"

```
module multimaths

function main([String] args) -> Void:
    print "Addition, subtraction, multiplication and division."
    print 10+10*(6-2)/2
```

Listing B.11: "Program written in Ralang: testcases/multimaths.ra"

```
module mulvars

function main([String] args) -> Void:
    Int a = 60
    print sum(a, 30)

function sum(Int a, Int b) -> Int:
    Int c = a+b*2
    return c
```

Listing B.12: "Program written in Ralang: testcases/mulvars.ra"

```
module notcompare

function main([String] args) -> Void:
    print notCompare(2016, 2016)
    print notCompare(2015, 2017)

function notCompare(Int year1, Int year2) -> String:
    if year1 != year2:
        return "Not same year."
    else:
        return "Same year."
```

Listing B.13: "Program written in Ralang: testcases/notcompare.ra"

```
module printnum

function main([String] args) -> Void:
    print "Simple addition."
    print 150+50
```

Listing B.14: "Program written in Ralang: testcases/printnum.ra"

```
module subtwo

function main([String] args) -> Void:
    print 2016-1993
```

Listing B.15: "Program written in Ralang: testcases/subtwo.ra"

```
module vars

function main([String] args) -> Void:
    print sum(10, 30)

function sum(Int a, Int b) -> Int:
    Int c = b+a
    return c
```

Listing B.16: "Program written in Ralang: testcases/vars.ra"

```
module varstring

function main([String] args) -> Void:
    print message("Hello there!")

function message(String text) -> String:
    return text
```

Listing B.17: "Program written in Ralang: testcases/varstring.ra"

B.3 Linux - Ubuntu 15.10 64-bit

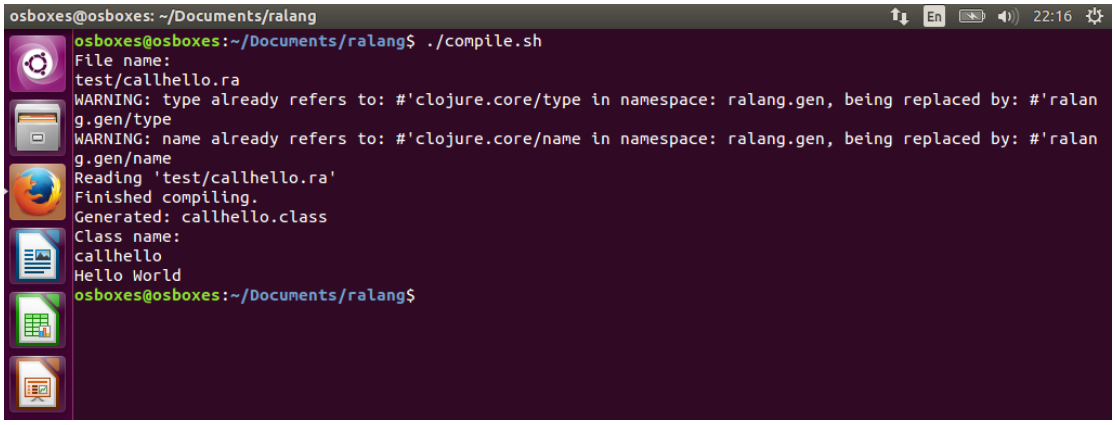


Figure B.1: Compiling and running callhello.ra on Ubuntu

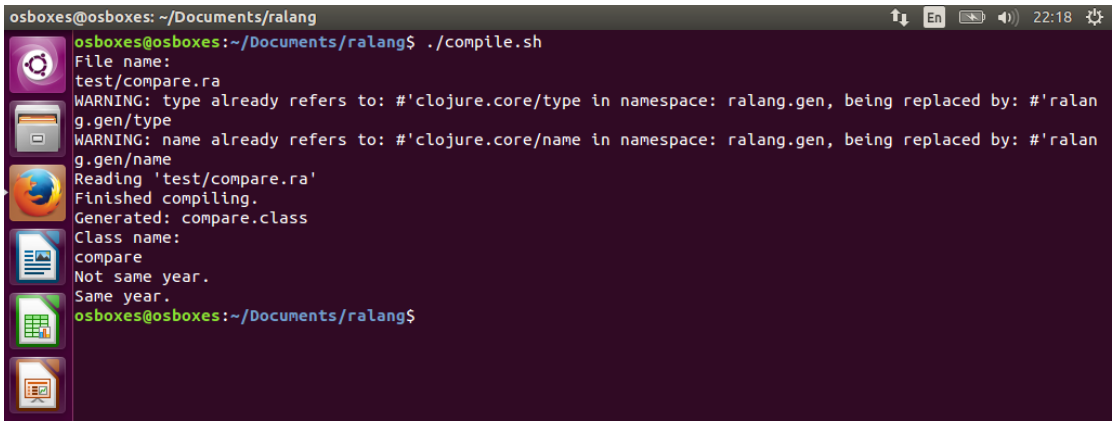


Figure B.2: Compiling and running compare.ra on Ubuntu

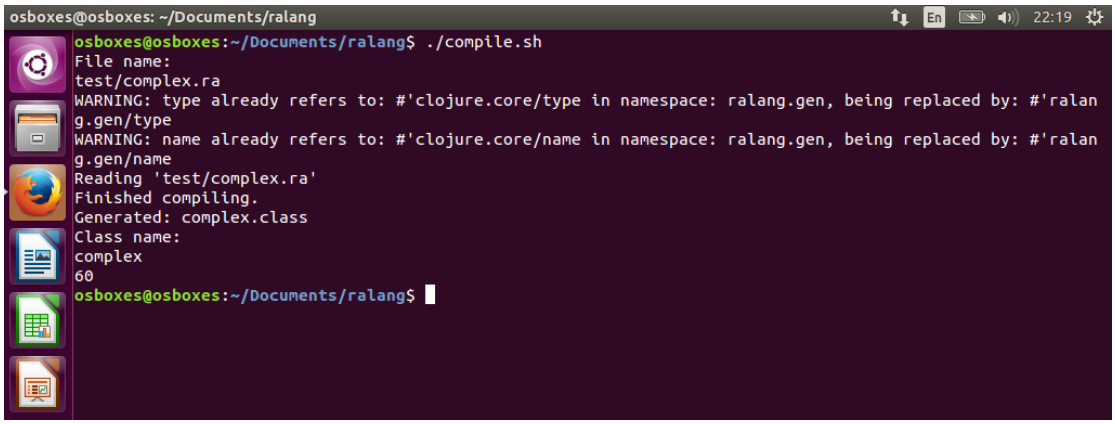


Figure B.3: Compiling and running complex.ra on Ubuntu

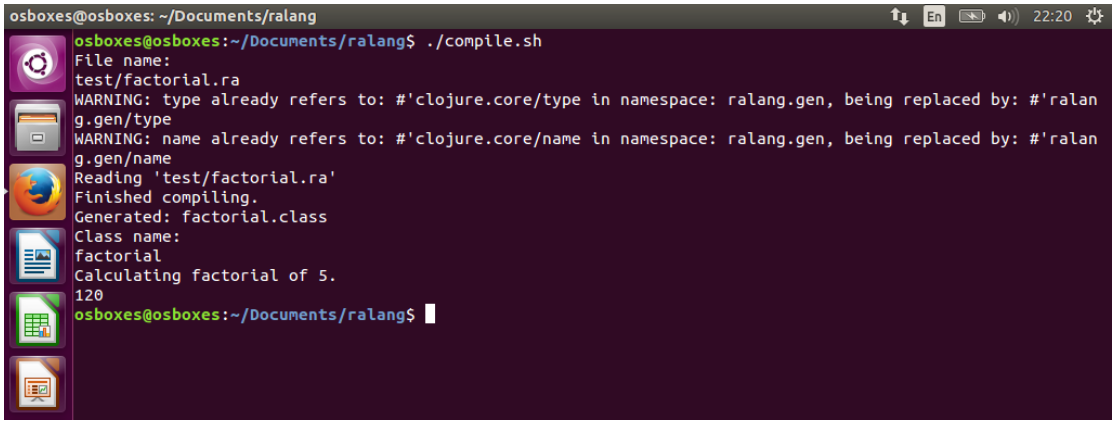


Figure B.4: Compiling and running factorial.ra on Ubuntu

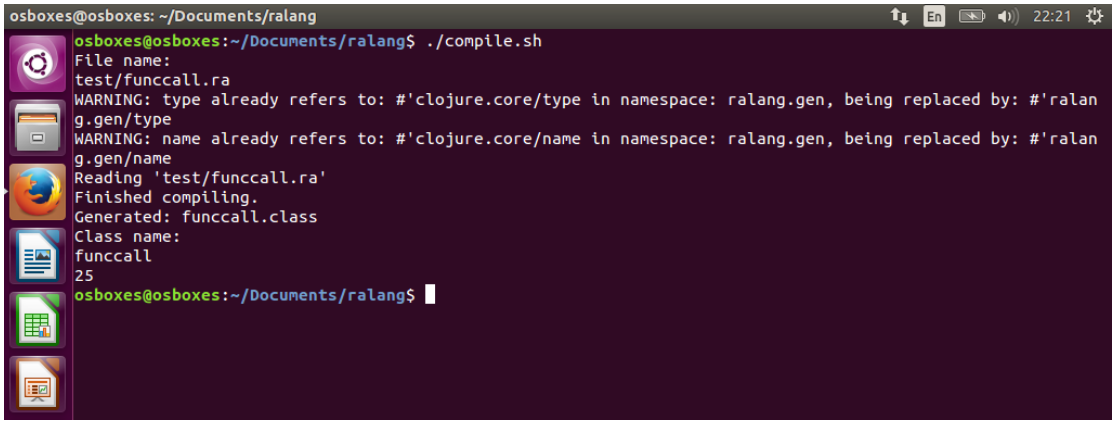


Figure B.5: Compiling and running funccall.ra on Ubuntu

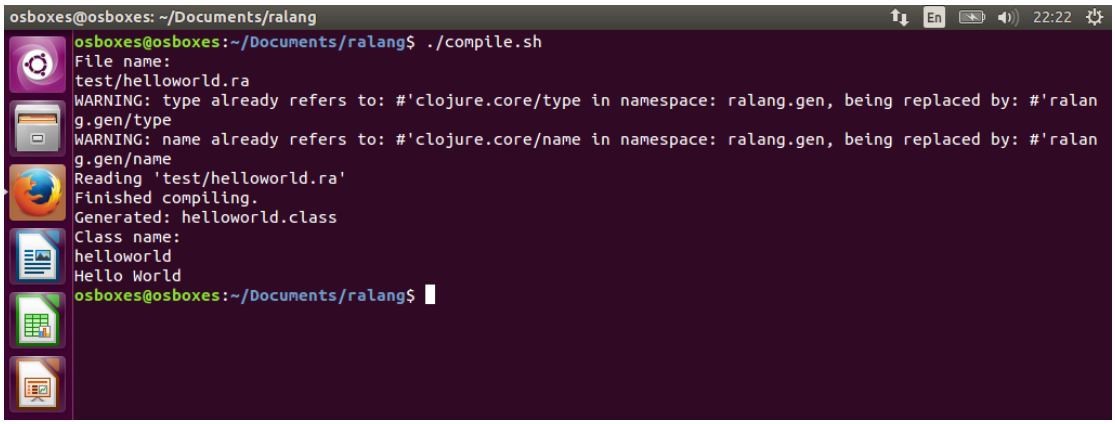


Figure B.6: Compiling and running helloworld.ra on Ubuntu

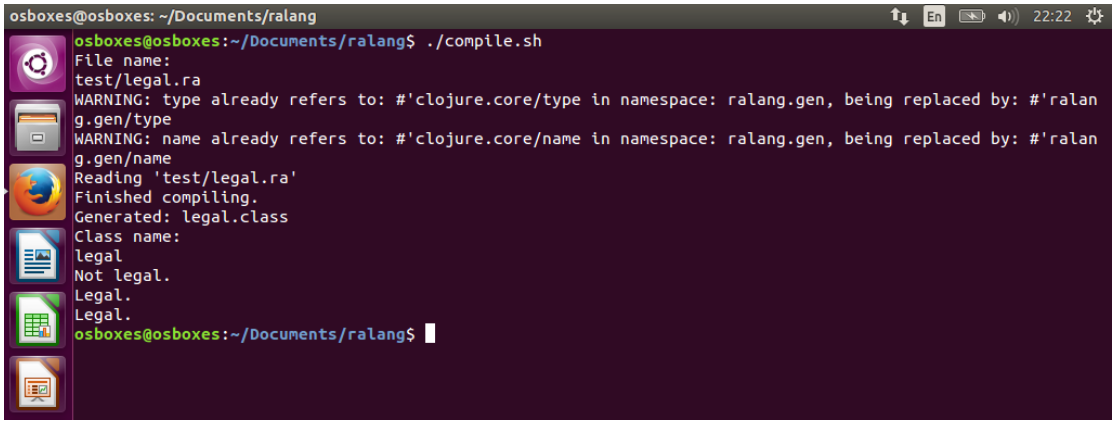


Figure B.7: Compiling and running legal.ra on Ubuntu

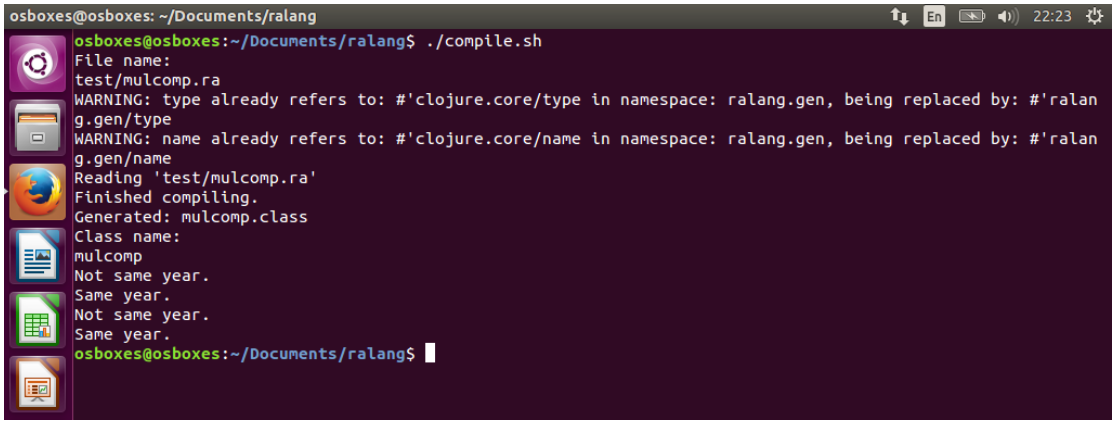


Figure B.8: Compiling and running mulcomp.ra on Ubuntu

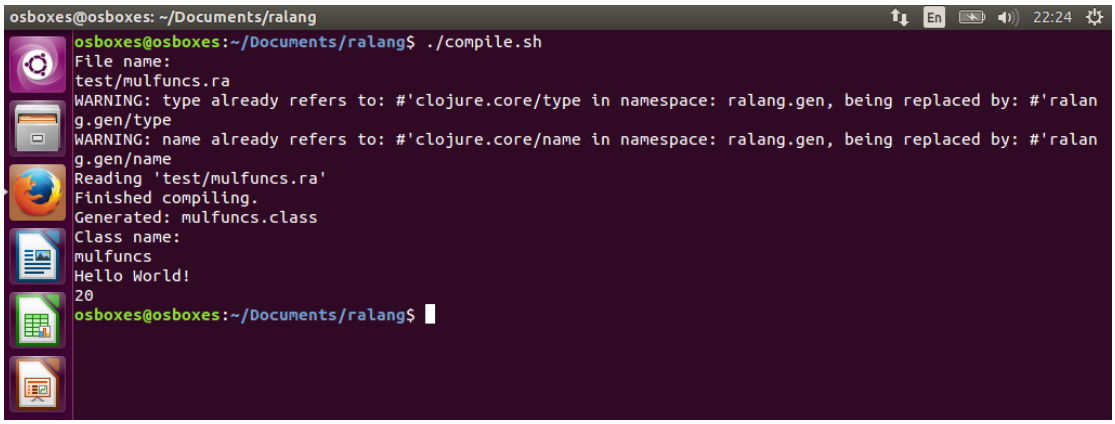


Figure B.9: Compiling and running mulfuncs.ra on Ubuntu

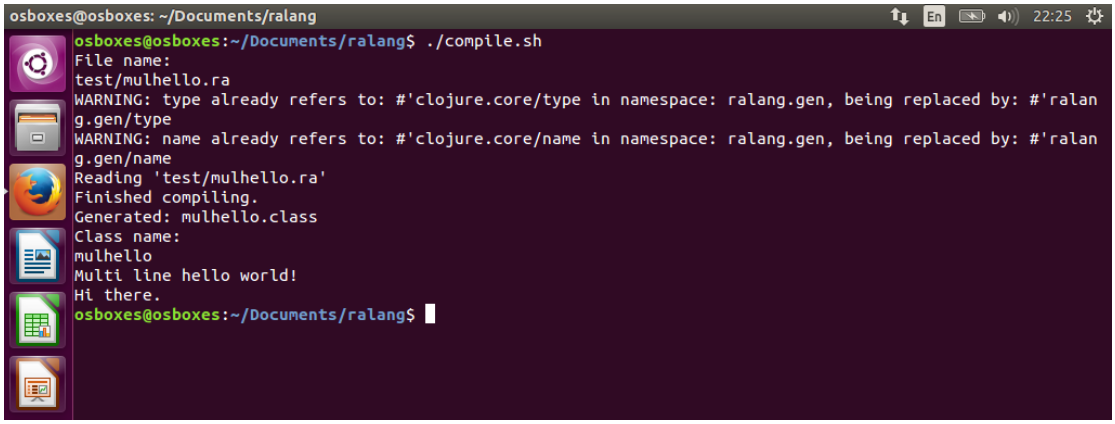


Figure B.10: Compiling and running mulhello.ra on Ubuntu

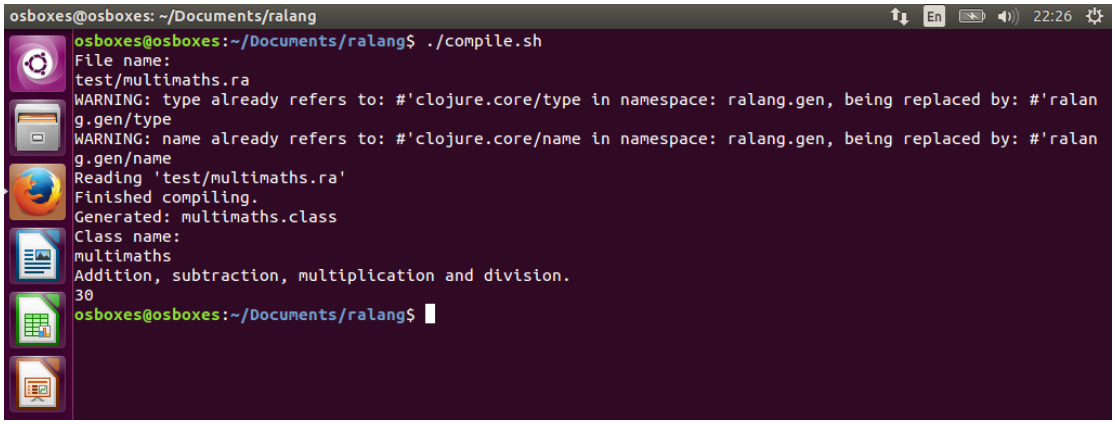


Figure B.11: Compiling and running multimaths.ra on Ubuntu

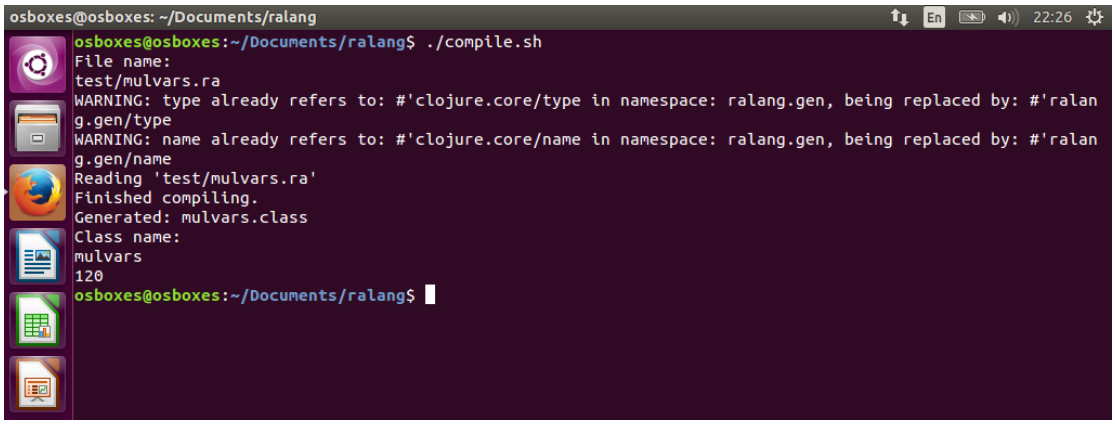


Figure B.12: Compiling and running mulvars.ra on Ubuntu

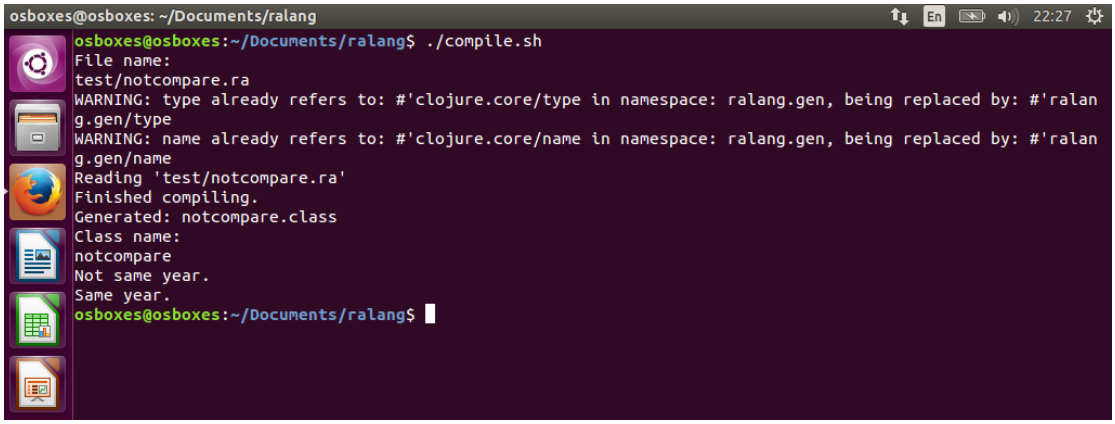


Figure B.13: Compiling and running notcompare.ra on Ubuntu

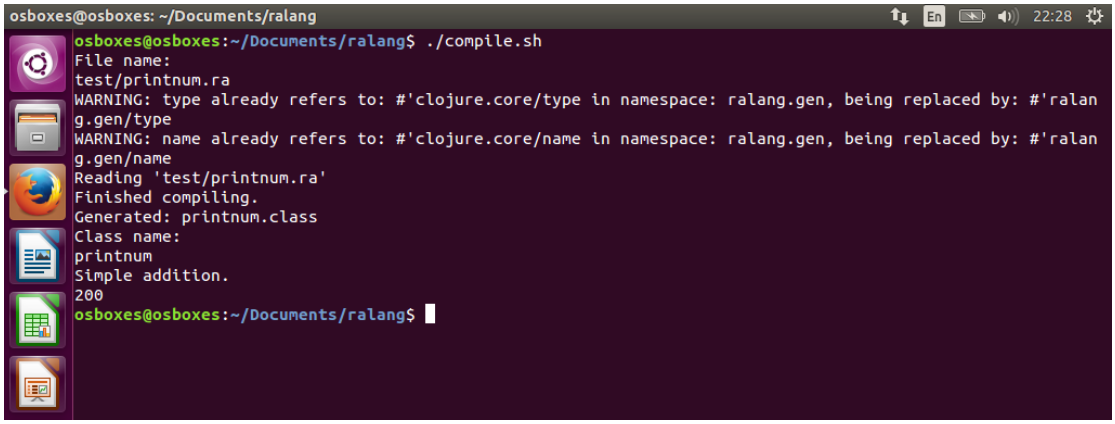


Figure B.14: Compiling and running printnum.ra on Ubuntu

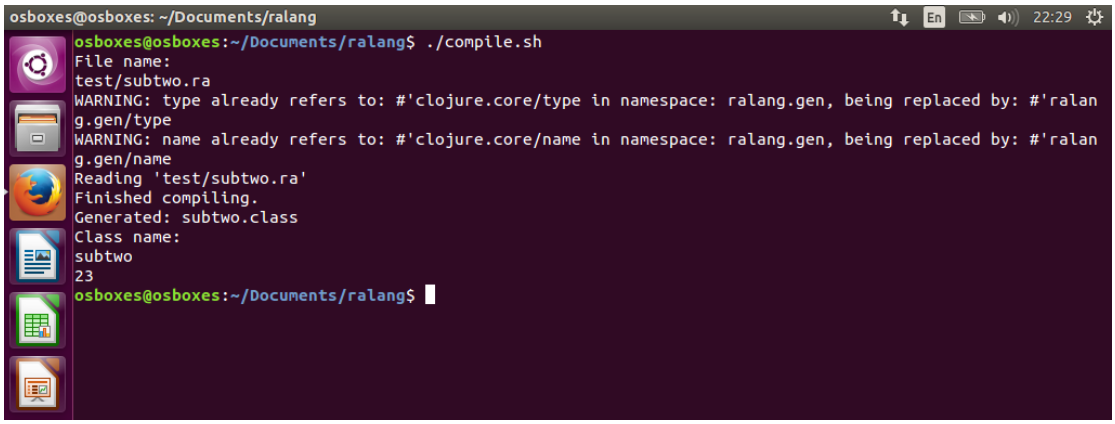


Figure B.15: Compiling and running subtwo.ra on Ubuntu

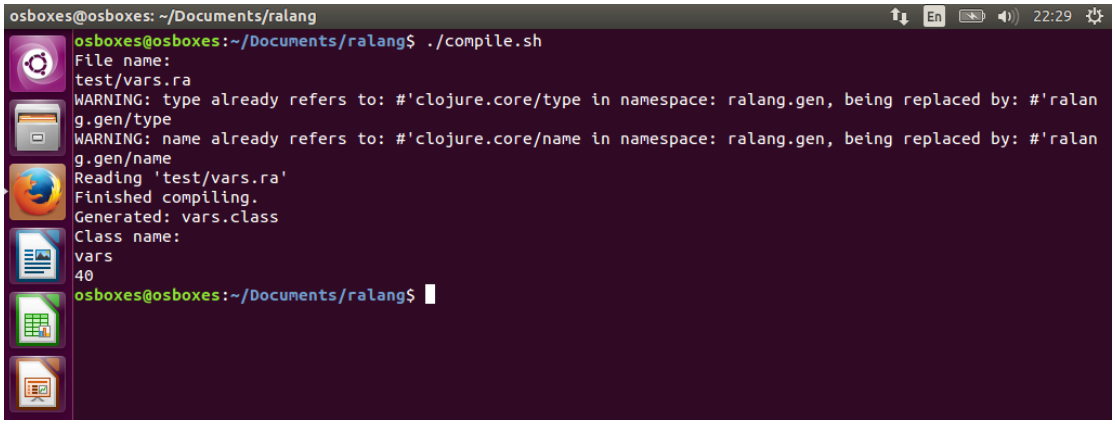


Figure B.16: Compiling and running vars.ra on Ubuntu

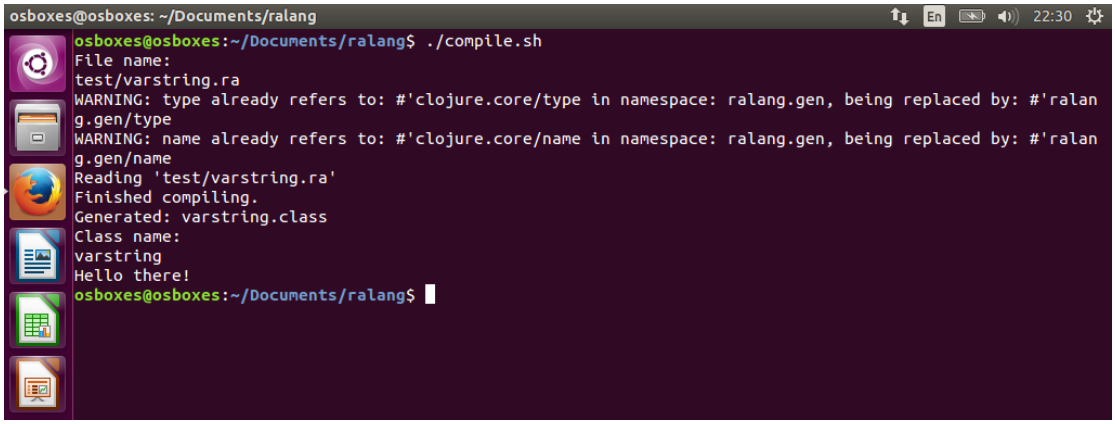


Figure B.17: Compiling and running varstring.ra on Ubuntu

B.4 Windows - Windows 10 32-bit

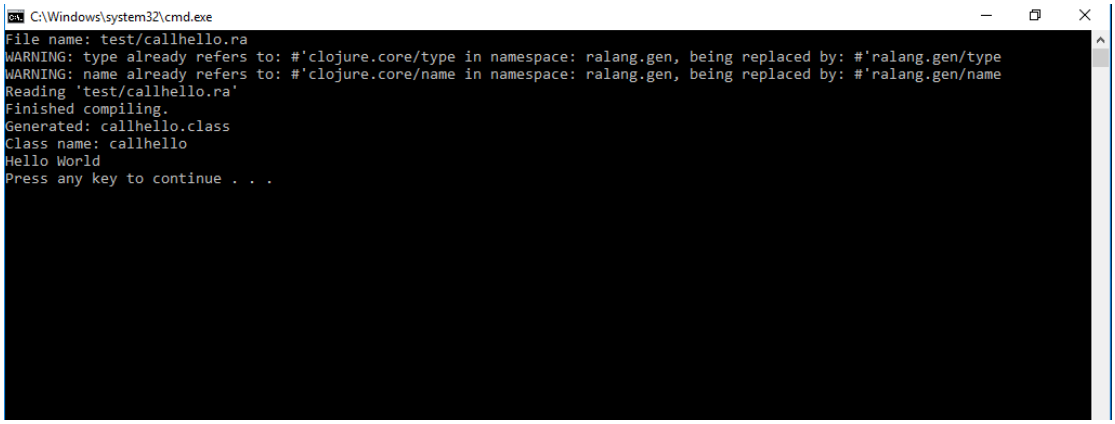


Figure B.18: Compiling and running callhello.ra on Windows

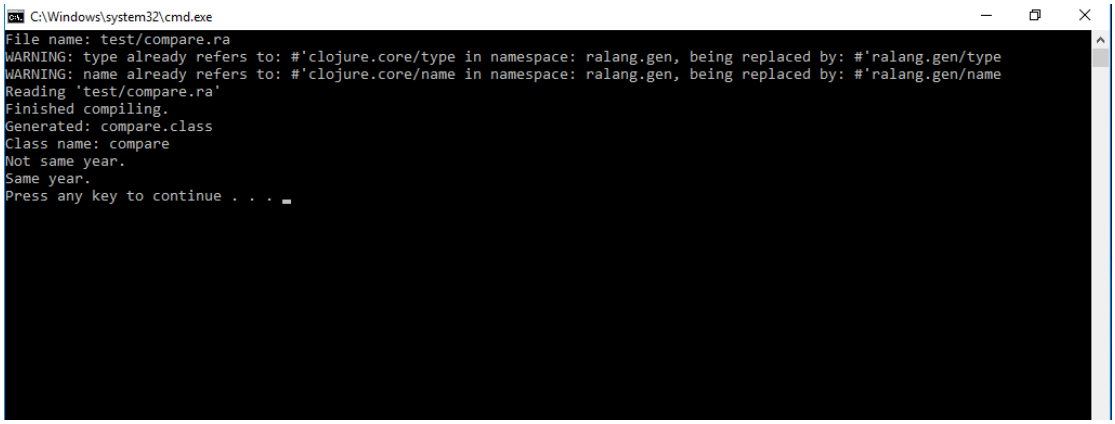


Figure B.19: Compiling and running compare.ra on Windows

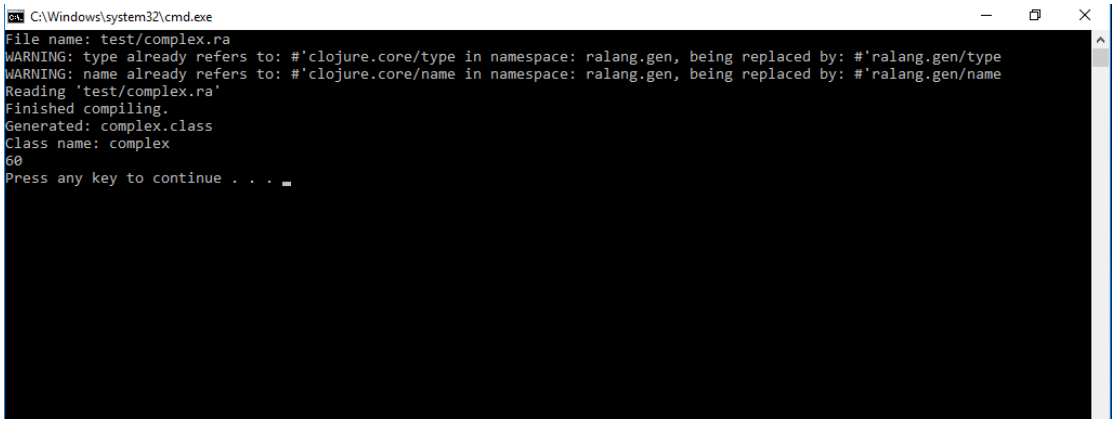


Figure B.20: Compiling and running complex.ra on Windows

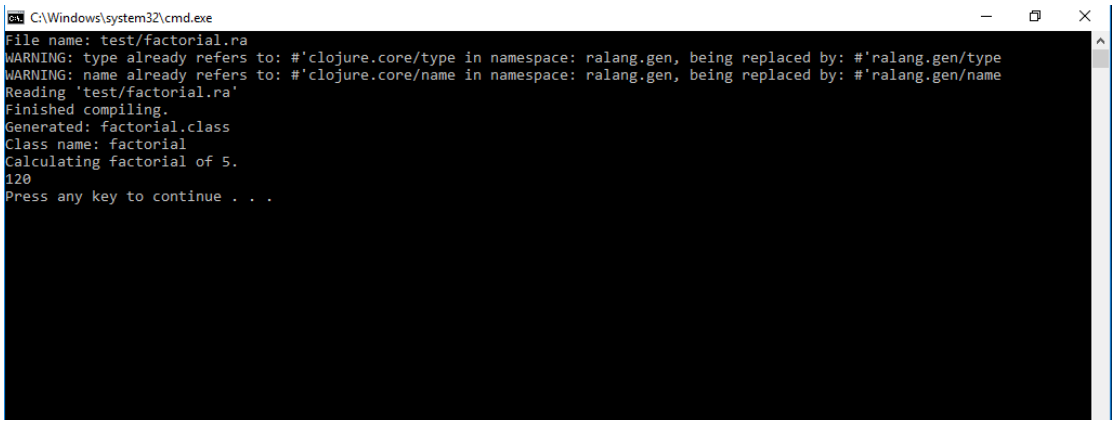


Figure B.21: Compiling and running factorial.ra on Windows

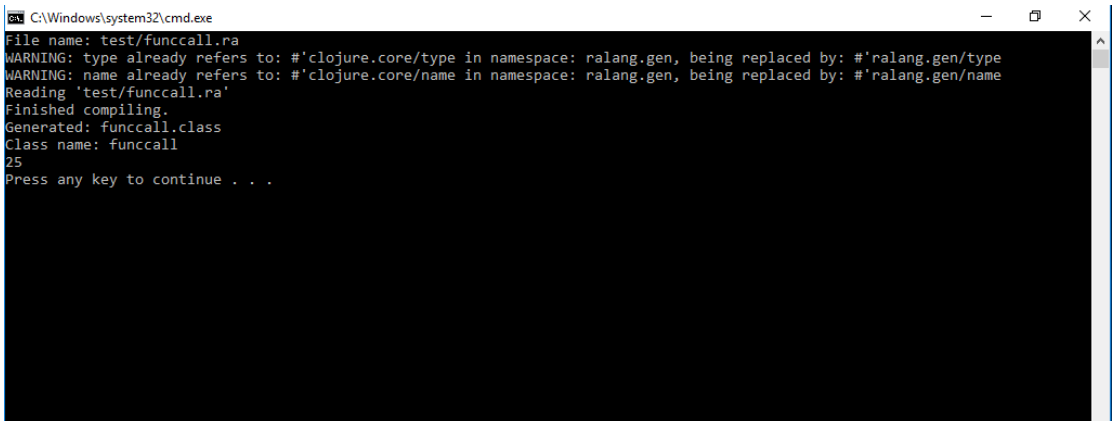


Figure B.22: Compiling and running funccall.ra on Windows

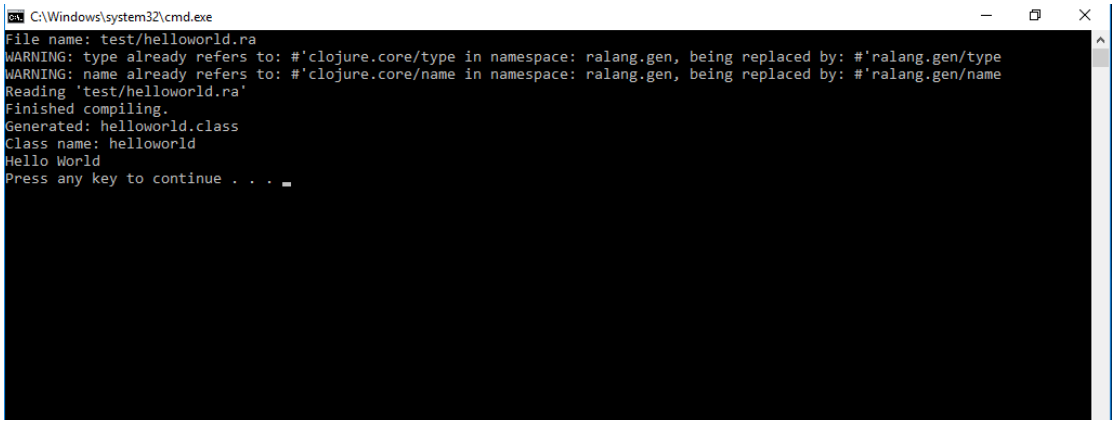


Figure B.23: Compiling and running helloworld.ra on Windows

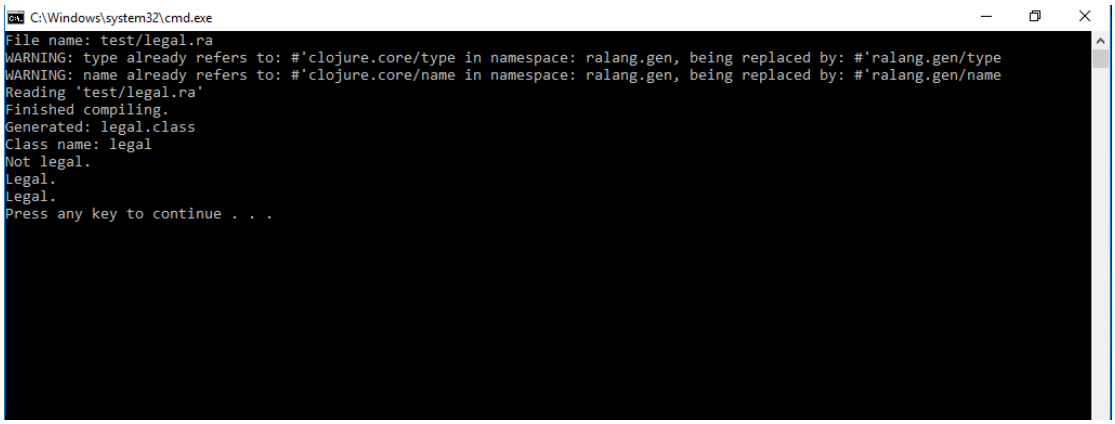


Figure B.24: Compiling and running legal.ra on Windows

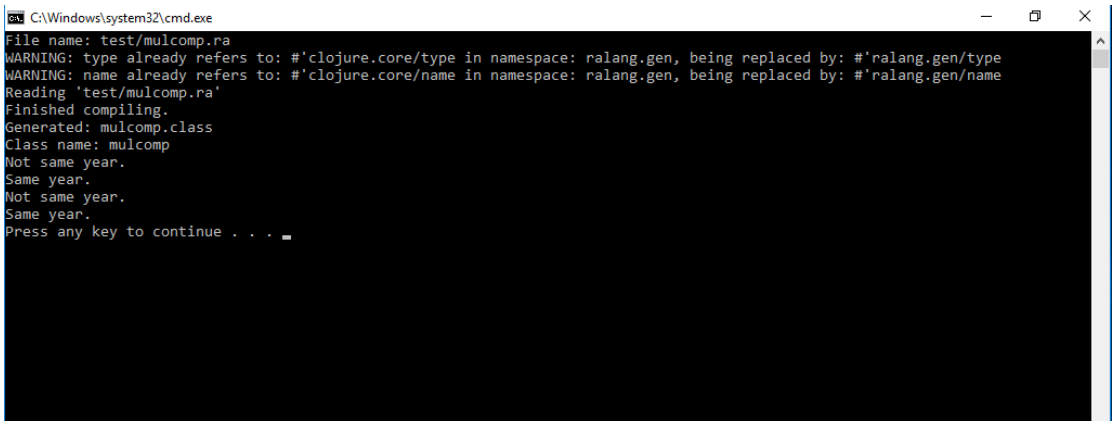


Figure B.25: Compiling and running mulcomp.ra on Windows

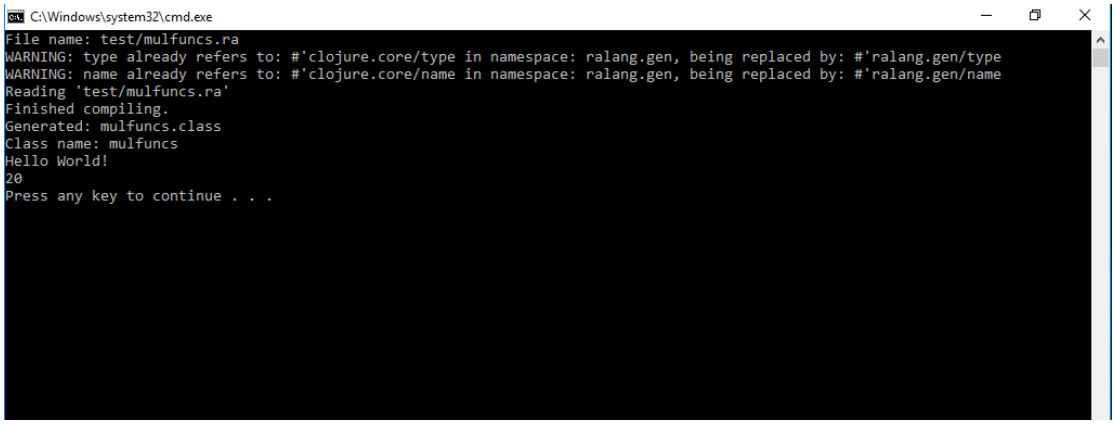


Figure B.26: Compiling and running mulfuncs.ra on Windows

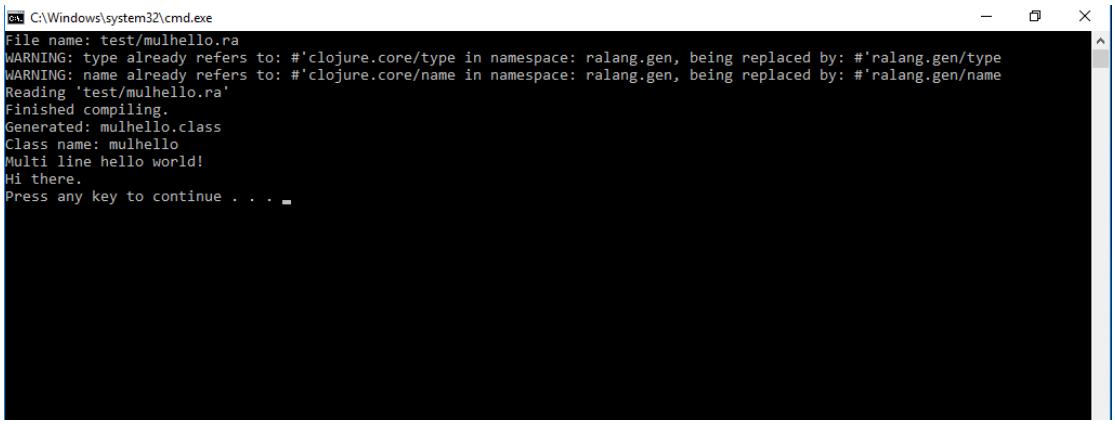


Figure B.27: Compiling and running mulhello.ra on Windows

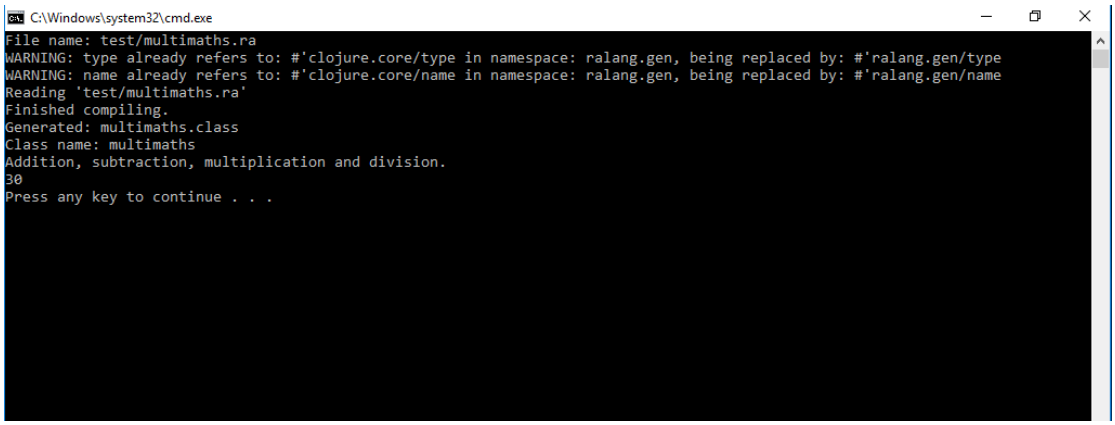


Figure B.28: Compiling and running multimaths.ra on Windows

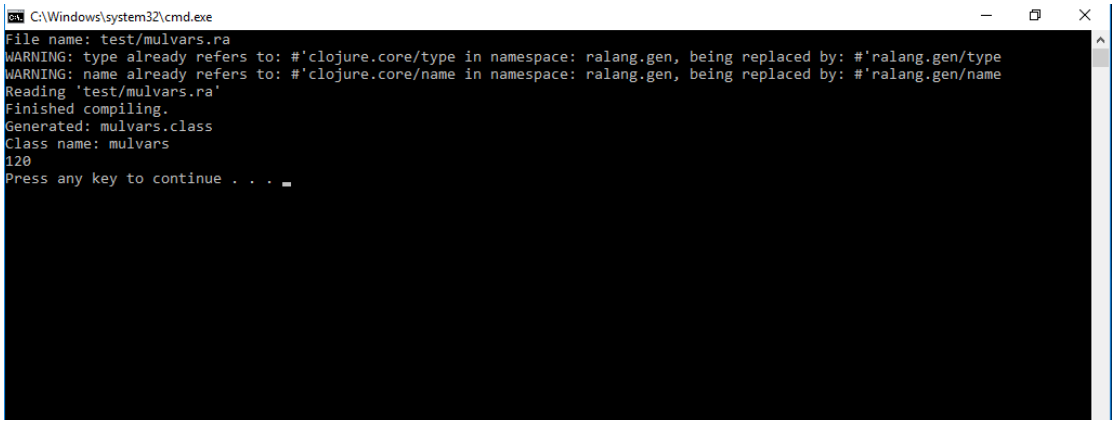


Figure B.29: Compiling and running mulvars.ra on Windows

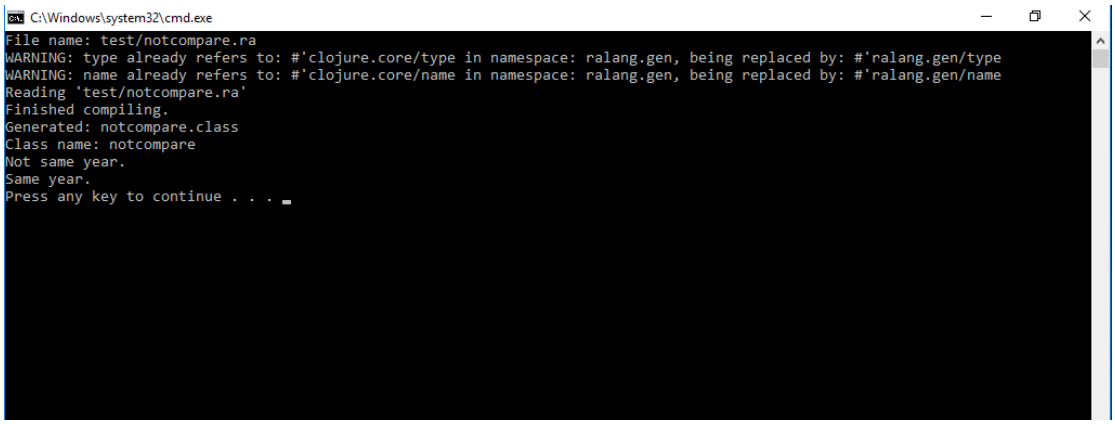


Figure B.30: Compiling and running notcompare.ra on Windows

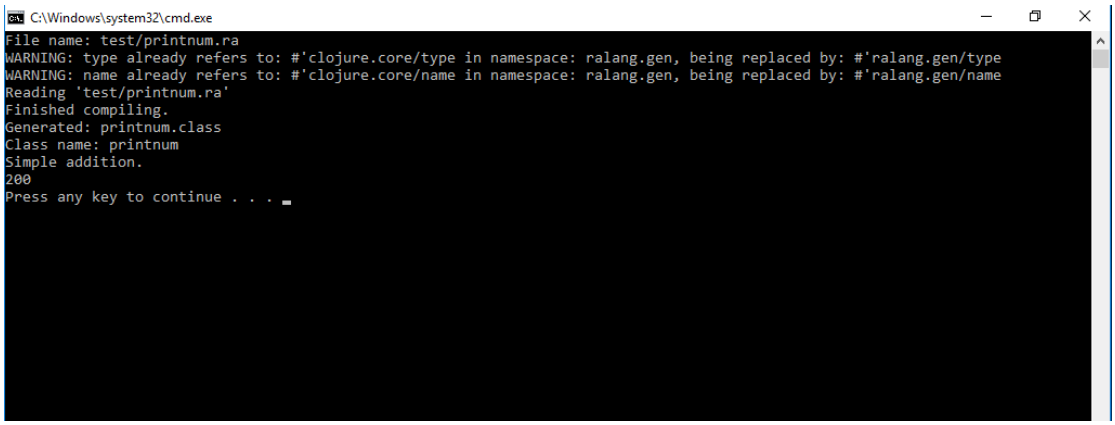


Figure B.31: Compiling and running printnum.ra on Windows

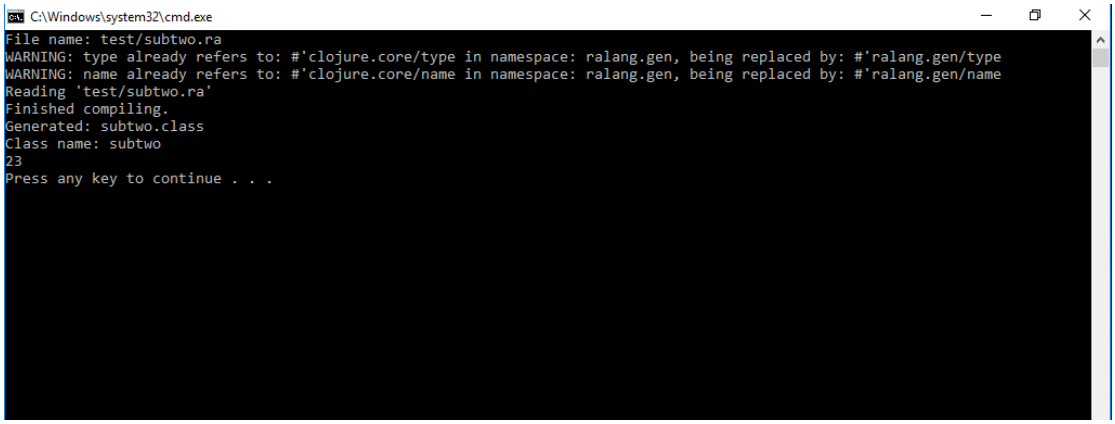


Figure B.32: Compiling and running subtwo.ra on Windows

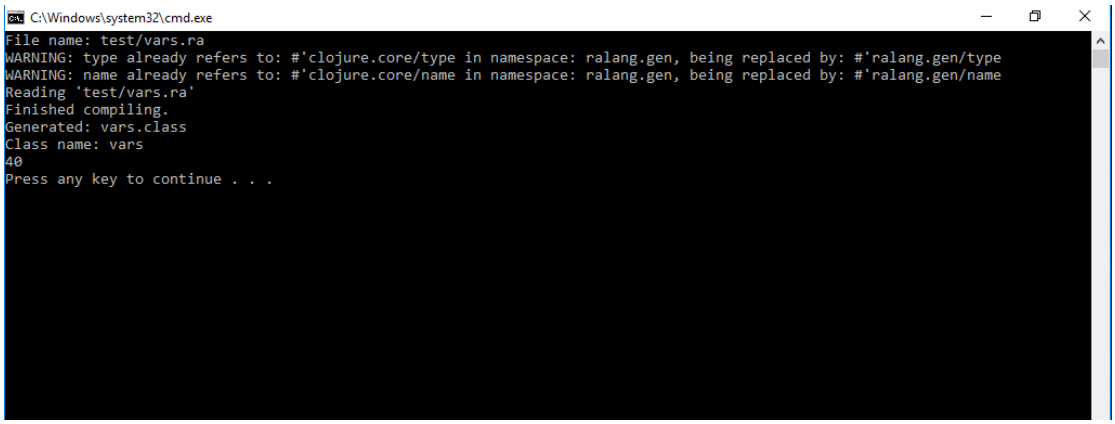


Figure B.33: Compiling and running vars.ra on Windows

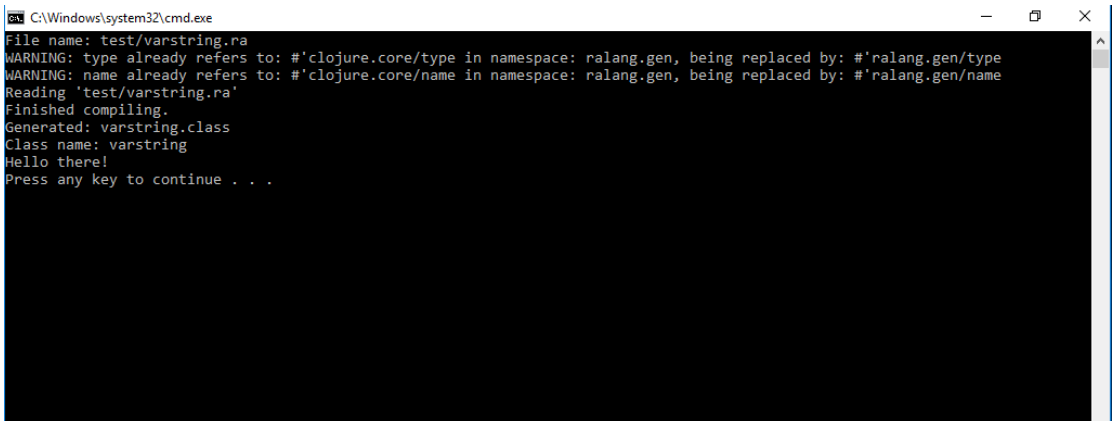


Figure B.34: Compiling and running varstring.ra on Windows

Appendix C Java integration

factorial.ra

```
1  module maths.factorial
2  # Name of the module to be imported by Java
3
4  function factorial(Int n) -> Int:
5      # Calculates the factorial of n
6      if n < 2:
7          return 1
8      else:
9          return n*factorial(n-1)
```

test.java

```
1  import maths.factorial;
2  // Imports factorial generated by Ralang
3
4  public class test {
5      public static void main(String[] args) {
6          System.out.println(factorial.factorial(1));
7          System.out.println(factorial.factorial(3));
8          System.out.println(factorial.factorial(5));
9          System.out.println(factorial.factorial(7));
10         System.out.println(factorial.factorial(9));
11     }
12 }
```

Figure C.1: Importing Ralang module into Java application

C:\WINDOWS\system32\cmd.exe

```
C:\Users\Daniel\Documents\ralang>javac test.java

C:\Users\Daniel\Documents\ralang>java test
1
6
120
5040
362880

C:\Users\Daniel\Documents\ralang>
```

Figure C.2: Running Java application after importing Ralang class

Appendix D High-level design

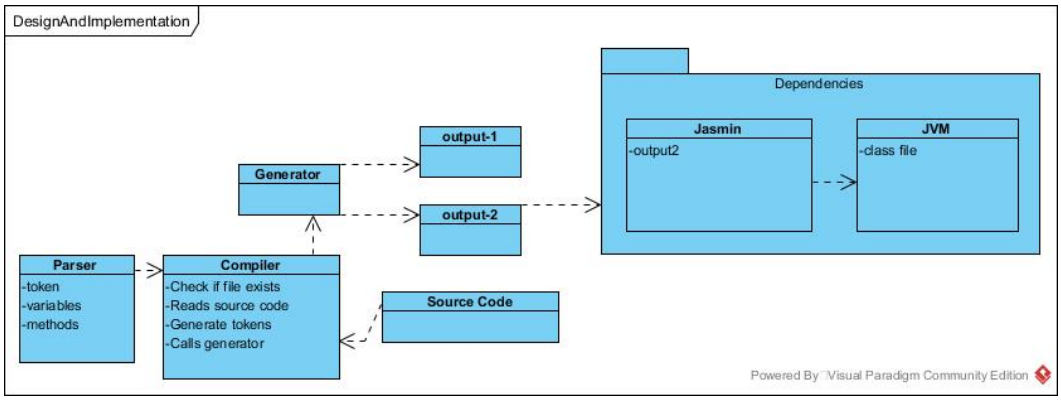


Figure D.1: High-level design of Ralang compiler

Appendix E Proposal - Introduction

Computer programming is across a range of different subjects such as Computer Science, Biology, Mathematics, Physics, etc. For that reason the first programming language is very important and critical choice to make. When the programming language is too hard students will become overwhelmed with the amount of work they have to do and might give up on the subject whilst if it's too easy then the students will be under qualified for a job after they graduate.

E.1 Aim

The aim of this project is to design and implement a compiler for a programming language to assist students currently enrolled in a computer science degree. This language is purely education and is not meant to replace any existing language currently in use, but to compliment them by providing the same functionality in a neater syntax so that students can familiarise themselves with computer programming instead of running into problems caused by syntactically incorrect programs as described by Krpan and Bilobrk (2011).

E.2 Objectives

1. The syntax of the language should be relatively easy for students taking a programming module in higher education. This can be evaluated by carrying out a simple test along with a questionnaire. Using statistics to compare the results to previous tests, the outcome of the tests can be obtained relatively quick as soon as the questionnaires have been filled in and the tests completed.

2. Given the amount of time provided to complete this project it's important to only focus on improving the correctness of the program rather the program's efficiency. Aho et al. (2006) says the code optimisation process must also be correct to preserve the meaning of the compiled program, this means focusing more on the code generation will achieve better results overall. It will then be possible to execute pre-defined test programs and data to evaluate whether the compiler provides the correct results.

3. The compiler is able to compile source code both for Linux and Windows operating system. Various number of tests will be used to check if the program performance has a significant changes in both these Operating Systems, or if they perform at the same level. To do this, a virtual machine will be created and given the same amount of memory, then tests can be performed. Setting up virtual machines and writing the test cases may take some time, nevertheless this will allow the creation of statistics on the data obtained from the results to compare the performance of the programming language in different operating systems.

E.3 Research methodologies

Only peer-reviewed journals and papers, and books on the compiler writing are going to be used through-out this research. Peer-reviewed journals and papers will be filtered out by skimming through abstracts and introductions, as time is a constraint for this project. The most relevant 10 to 15 journals and papers will be selected for the literature review. Well known books will be referenced in the reference section at the end. Books will only be used if there are no relevant papers on the subject.

Appendix F Proposal - Literature review

F.1 Introduction

Computer programming is a fundamental requirement at any Bachelor of Science degree such as physics, biology and computer science. Nevertheless, learning a programming language can be a difficult and taunting challenge for people who have not programmed before. Vihavainen, Airaksinen, and Watson (2014) says despite the enormous efforts to improve CS failure rates, these numbers are still very high across the board. Institutions also have to ensure students enrolled in computer programming courses are able to understand the content of the course and at the same time the students must be prepared to use the skills gained to go into industry confident and able to perform the required jobs. (Krpan and Bilobrk, 2011)

This literature review attempts to answer two main questions: What do the majority students enrolled on a computer programming course find difficult and how can this problem be tackled. How the problem can be tackled also needs to take into consideration programming languages currently being used in the industry, and whether learning a certain pedagogical language, would enable the students to transit to another industry programming language such as Java, C# or C++ for example.

F.2 Introductory programming languages

There are various different approaches to the problem of what programming languages are better to learn as a introductory language. Daly (2011) compares using Alice/Java course against a pure Java course. Alice is a object-oriented 3D programming environment used to introduce students to computer programming. The results carried out of this research found that students learning Java along with Alice were a lot more confident than those learning pure Java. Even though Daly (2011) concludes that students that learn Alice alongside Java seem to be more enthusiastic about taking another programming course than those that learn pure Java, it failed to address the issue of how well the students actually performed in the end in terms of being able to write a simple computer program.

Krpan and Bilobrk (2011) takes a similar approach to the problem above but this time around students are learning C, QBasic and Python. In this experiment the students attempt 3 different exercises: calculating factorial of n, reversing a string and counting words in a text file. The students seemed more confident about programming in Python, and rated C as the most difficult programming language to use, nevertheless achieved an overall better results in the C language.

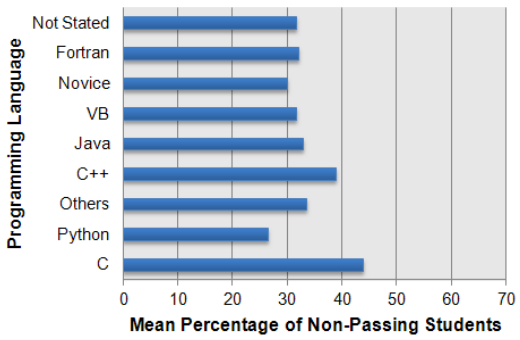


Figure F.1: Non-passing students grouped by programming language. (Watson and Li, 2014)

Watson and Li (2014) obtained a more comprehensive set of results by combining data from 145 universities and 16 college from various different countries and looking at the top programming languages used at these institutions. Figure 1, compares Fortran, Novice, VB, Java, C++, Python, C and other languages. Python programming language in this case achieves the best mean percentage, whilst C programming language seems to get more non-passing students, nevertheless as mentioned in Watson and Li (2014), the mean percentage of non-passing students does not seem to vary extraordinarily as the results range from 25% to 45% and remains consistent throughout.

In Watson and Li (2014), shows that there are trends mean percentage of non-passing students is actually decreasing over the years. This could be due to the fact that easier to learn programming languages are being introduced, and therefore it is easier for students to pass. This does not necessarily mean that students will be able to solve problems by themselves after they finish the computer programming course, and this could be an issue in the future.

Another problem is whether the students will be able to tackle issues by themselves when they finish their course. Jayal et al. (2011) says that there is a positive increase in the number of students taking Python as their first programming language. It reason that students taking Python can focus more on the problems themselves rather than syntactic issues. This means that students taking Python as their first language are more likely to be able to tackle issues by themselves.

Python is getting a lot of popularity in computer programming course as an introductory language. (Yadin, 2011) Python programs allow students to focus on the procedural programming allowing them to think of problems and how to solve them using algorithms. Students tend to find object-oriented programming and this could be the reason why Java seems to lower the confidence levels in students when taught as a first programming language like shown in Daly (2011). In fact, many institutions were able to lower the drop out rate by introduction Python as an introductory programming language. (Nikula et al., 2007)

Similar issues apply when teaching functional programming to first years. Chakravarty and Keller (2004) says that functional programming is hardly a good idea for a student who has never came across programming before.

F.3 Conclusion

At various different institutions around the world different approaches are being taken to lower the number of drop out and to increase the number of students who will be motivated enough to pursue another computer programming course. A lot of these institution seem to be preferring a Python approach to this problem, and drop outs seem to be decreasing. (Jayal et al., 2011)

Nevertheless, the trend doesn't seem to be mutual across every institution as the results tends to vary and indicate otherwise as observed by Krpan and Bilobrk (2011) where the average students preferred Python but achieved an overall better grade in C. A slight different case where students across 144 different universities obtained about the same performance in different languages, suggesting the language might not impact how well students solve problems using the computer. (Watson and Li, 2014)

It's also important to note that students learning procedural programming rather than objected-oriented or functional programming, are able to perform better when it comes to writing algorithms and solving problems. (Kölling, 1999)

It is also important to note that most of the statistics provided in the above papers were very limited and might not represent fully correct data, as only samples were collected. The largest sample of data found was done to 161 institutions from all around the world.

Appendix G Proposal - Design

G.1 Requirement specification

Ralang source code should be easy to read and easy to write. By comparing Java source code with Ralang source code in Figure 2. it's clear that Ralang requires less lines of code and less characters to print "Hello, World" to the screen, meaning that student will have to write much less to obtain the same results.

Objective 3 for this project is that the compiler for Ralang must be able to run both on Windows and Linux operating system. In order to make that happen the source code of the program is compiled to Java bytecode to run on the Java Virtual Machine. There are several languages that will run on the JVM such as Java, Scala, Clojure and Jython. A functional programming would be more suitable for this project because of compiler work a lot with trees and also pattern matching, in which case Clojure may be very suitable.

To understand whether the programming language meets the requirements, a simple test along with a questionnaire will be carried out to first and second year students, who may have prior programming knowledge and they will be required to follow a written or video tutorial, then write a simple application and finally answer some questionnaire questions about how they feel about the language.

Aho et al. (2006) lays out the seven different phases to writing a compiler: Lexical, syntax and semantic analyser, intermediate code generator, machine-independent code optimiser, code generator and machine-dependent code optimizer. To ensure the project can be successfully completed within a certain time frame, code optimisation will be skipped. In that case more time can be spent to ensure the correctness of the compiler.

From the literature review, it was concluded that the Python programming language is used to at various institutions to introduce students to computer programming, nevertheless many students claimed Python was far easier than languages like C but still the overall average performed better in C. (Krpan and Bilobrk, 2011)

It was understood the majority of people preferred Python because of its neat and clear syntax but due to being a dynamic programming language students tend to have many problem being able to write and run their own programs. Ralang will implement static typing, where the compiler will do various checks and report errors where possible but also implement a neat and clear syntax like Python.

G.2 Development methodologies

There are several development methodologies available to use for this project such as RAD or scrum software development.

Scrum software development assumes that the problem can not be completely laid out, and changes may be made along the way. This methodology allows quick prototyping and would allow quick deliverables, so it would work well for projects that would require to be changed frequently.

Similarly, RAD provides a fast development process and it's easy to adjust to the requirements of the project. It will allow easy prototyping and making changes as necessary to obtain best results possible, whether that would be the correctness of the program, or changes to the syntax of language or even how to deal with compiling errors.

RAD seems the most suitable methodology as the development of the project won't be altering constantly, there might be slight changes along the way until the final product is complete, nevertheless the changes will always be minor until the end product is finished.

Appendix H Research Questionnaire

Research Questionnaire

Name and E-mail are *optional* it may be necessary for us to get in touch with you in the future for further research on this subject. We really appreciate your cooperation.

Name (optional): _____

E-mail (optional): _____

Educational institution/Work place: _____

Course/Year: _____

Please number the following languages by *preference,* (lowest is most preferred). Also, please keep in mind you don't need to fill all of them in.

__ ASP.NET
__ C
__ C++
__ C#
__ Clojure
__ F#
__ Go
__ Haskell
__ Java
__ Javascript
__ Lisp
__ Matlab
__ Objective-C
__ Perl
__ PHP
__ Python
__ Ruby
__ Scala
__ Swift
__ Visual Basic

Please study the following program below. Try to understand the control flow of Ralang, and identify their similarities and differences between the two examples.

```
module factorial

function main([String] args) -> Void:
    print "Calculating factorial of 5."
    print factorial(5)
    # 120

function factorial(Int n) -> Int:
    # Calculates the factorial of n
    if n < 2:
        return 1
    else:
        return n*factorial(n-1)
```

Figure 2. Program written in Ralang

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace factorial
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Calculating factorial of 5.");
            Console.WriteLine(factorial(5));
            // 120
        }

        static int factorial(int n)
        {
            // Calculate the factorial of n
            if (n < 2)
            {
                return 1;
            } else {
                return n * factorial(n - 1);
            }
        }
```

Figure 1. Program written in C#

Figure H.1: Research Questionnaire about Ralang's syntax - Page 1

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow					
2. There are a lot of similarities between both languages					
3. Ralang could be used as an educational programming language					
4. C# syntax is very neat and easy to understand					
5. Flow of control seems to be very different in both languages					
6. Ralang's syntax and control flow are not made very clear					
7. I would be interested in learning more about Ralang.					

Please use the following section to let us know how you feel about Ralang project:

Figure H.2: Research Questionnaire about Ralang's syntax - Page 2

Appendix I Questionnaires

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow	✓				
2. There are a lot of similarities between both languages				✓	
3. Ralang could be used as an educational programming language			✓		
4. C# syntax is very neat and easy to understand					✓
5. Flow of control seems to be very different in both languages					✓
6. Ralang's syntax and control flow are not made very clear			✓		
7. I would be interested in learning more about Ralang.	✓				

Please use the following section to let us know how you feel about Ralang project:

~~I'm unsure initially~~ From first view Ralang seems to be neat and tidy. I like the structure and flow but feel it mimicks C# very closely even including an array of strings in the main() function even though it may not necessarily be used.

Figure I.1: Research Questionnaire - Answer sheet number 1

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow		✓			
2. There are a lot of similarities between both languages				✓	
3. Ralang could be used as an educational programming language	✓				
4. C# syntax is very neat and easy to understand		✓			
5. Flow of control seems to be very different in both languages	✓				
6. Ralang's syntax and control flow are not made very clear					✓
7. I would be interested in learning more about Ralang.	✓				

Please use the following section to let us know how you feel about Ralang project:

With more work could develop into an extremely capable language

Figure I.2: Research Questionnaire - Answer sheet number 2

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow		✓			
2. There are a lot of similarities between both languages				✓	
3. Ralang could be used as an educational programming language		✓			
4. C# syntax is very neat and easy to understand				✓	
5. Flow of control seems to be very different in both languages					✓
6. Ralang's syntax and control flow are not made very clear					✓
7. I would be interested in learning more about Ralang.		✓			

Please use the following section to let us know how you feel about Ralang project:

It is very easy to follow with single syntax

Figure I.3: Research Questionnaire - Answer sheet number 3

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow	✓				
2. There are a lot of similarities between both languages		✓			
3. Ralang could be used as an educational programming language		✓			
4. C# syntax is very neat and easy to understand		✓			
5. Flow of control seems to be very different in both languages		✓			
6. Ralang's syntax and control flow are not made very clear				✓	
7. I would be interested in learning more about Ralang.		✓			

Please use the following section to let us know how you feel about Ralang project:

I want to see more.

Figure I.4: Research Questionnaire - Answer sheet number 4

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow	✓				
2. There are a lot of similarities between both languages		✓			
3. Ralang could be used as an educational programming language	✓				
4. C# syntax is very neat and easy to understand	✓				
5. Flow of control seems to be very different in both languages		✓			
6. Ralang's syntax and control flow are not made very clear				✓	
7. I would be interested in learning more about Ralang.	✓				

Please use the following section to let us know how you feel about Ralang project:

I think it is quite an achievement and I would like to get to know more about Ralang.

Figure I.5: Research Questionnaire - Answer sheet number 5

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow		✓			
2. There are a lot of similarities between both languages	✓				
3. Ralang could be used as an educational programming language	✓				
4. C# syntax is very neat and easy to understand		✓			
5. Flow of control seems to be very different in both languages		✓			
6. Ralang's syntax and control flow are not made very clear			✓		
7. I would be interested in learning more about Ralang.	✓				

Please use the following section to let us know how you feel about Ralang project:

interesting

Figure I.6: Research Questionnaire - Answer sheet number 6

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow			✓		
2. There are a lot of similarities between both languages				✓	
3. Ralang could be used as an educational programming language			✓		
4. C# syntax is very neat and easy to understand		✓			
5. Flow of control seems to be very different in both languages		✓			
6. Ralang's syntax and control flow are not made very clear		✓			
7. I would be interested in learning more about Ralang.				✓	

Please use the following section to let us know how you feel about Ralang project:

Figure I.7: Research Questionnaire - Answer sheet number 7

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow		✓			
2. There are a lot of similarities between both languages			✓		
3. Ralang could be used as an educational programming language		✓			
4. C# syntax is very neat and easy to understand		✓			
5. Flow of control seems to be very different in both languages	✓				
6. Ralang's syntax and control flow are not made very clear	✓				
7. I would be interested in learning more about Ralang.	✓				

Please use the following section to let us know how you feel about Ralang project:

Figure I.8: Research Questionnaire - Answer sheet number 8

Please use the table below to give us some feedback:

	Strongly Agree	Agree	No opinion	Disagree	Strongly Disagree
1. Ralang appears to be readable and easy to follow		✓			
2. There are a lot of similarities between both languages				✓	
3. Ralang could be used as an educational programming language			✓		
4. C# syntax is very neat and easy to understand	✓				
5. Flow of control seems to be very different in both languages				✓	
6. Ralang's syntax and control flow are not made very clear				✓	
7. I would be interested in learning more about Ralang.		✓			

Please use the following section to let us know how you feel about Ralang project:

Figure I.9: Research Questionnaire - Answer sheet number 9

Appendix J Research Questionnaire - Results

Table J.1 is used to evaluate the results collected from the research questionnaire. Each of these values will be multiplied by the number of results collected in table J.2.

	Strongly Agree	Agree	No opinion	Disagree	Strongly disagree
Ralang appears to be readable and easy to follow	4	3	2	1	0
Ralang could be used as an educational programming language	4	3	2	1	0
Ralang's syntax and control flow are not made very clear	0	1	2	3	4
I would be interested in learning more about Ralang	4	3	2	1	0

Table J.1: Contains the values for each cell

	Strongly Agree	Agree	No opinion	Disagree	Strongly disagree
Ralang appears to be readable and easy to follow	3	5	1	0	0
Ralang could be used as an educational programming language	3	3	3	0	0
Ralang's syntax and control flow are not made very clear	1	1	2	3	2
I would be interested in learning more about Ralang	5	3	0	1	0

Table J.2: Research questionnaire results for N=9

Appendix K Presentation

K.1 Slides

Figure K.1: Presentation slides, demonstration and viva slides - Introduction

Figure K.2: Presentation slides, demonstration and viva slides - Overview

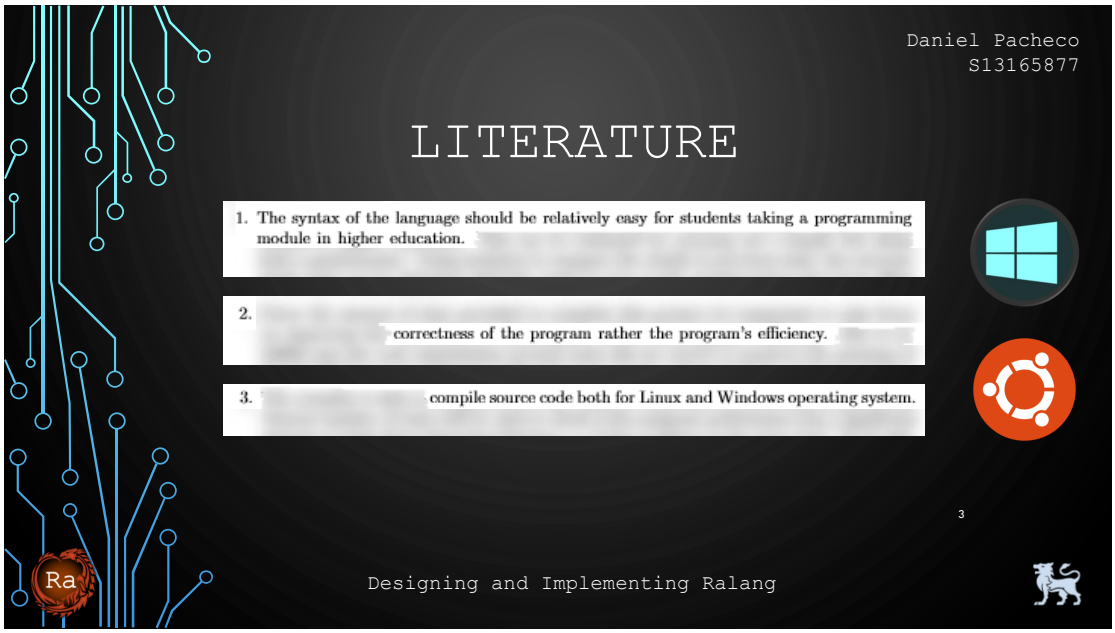


Figure K.3: Presentation slides, demonstration and viva slides - Literature

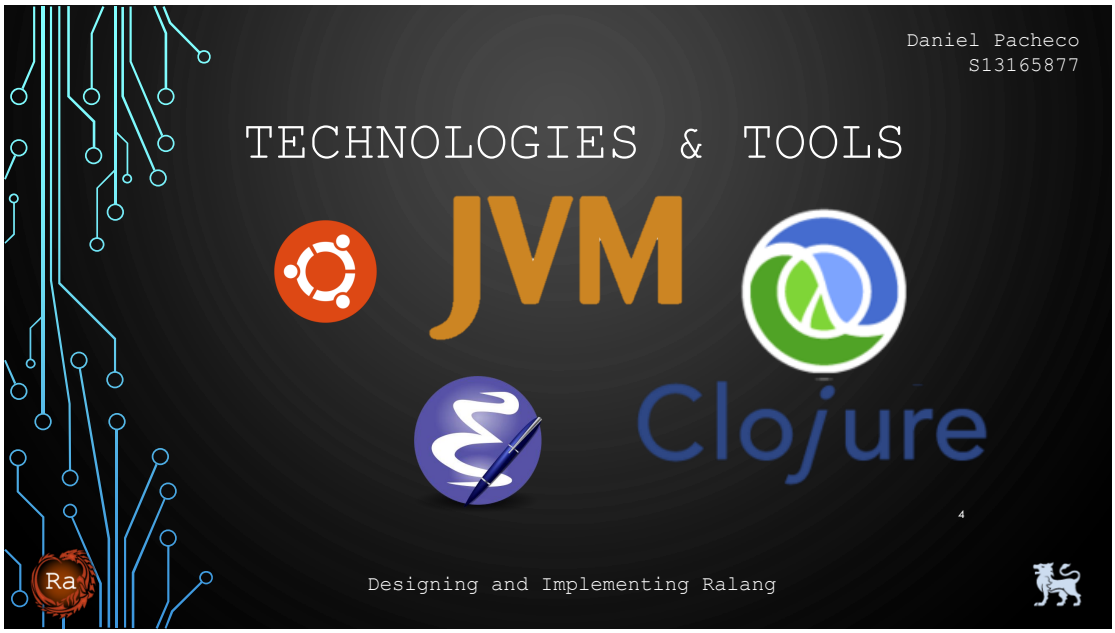


Figure K.4: Presentation slides, demonstration and viva slides - Technologies and tools

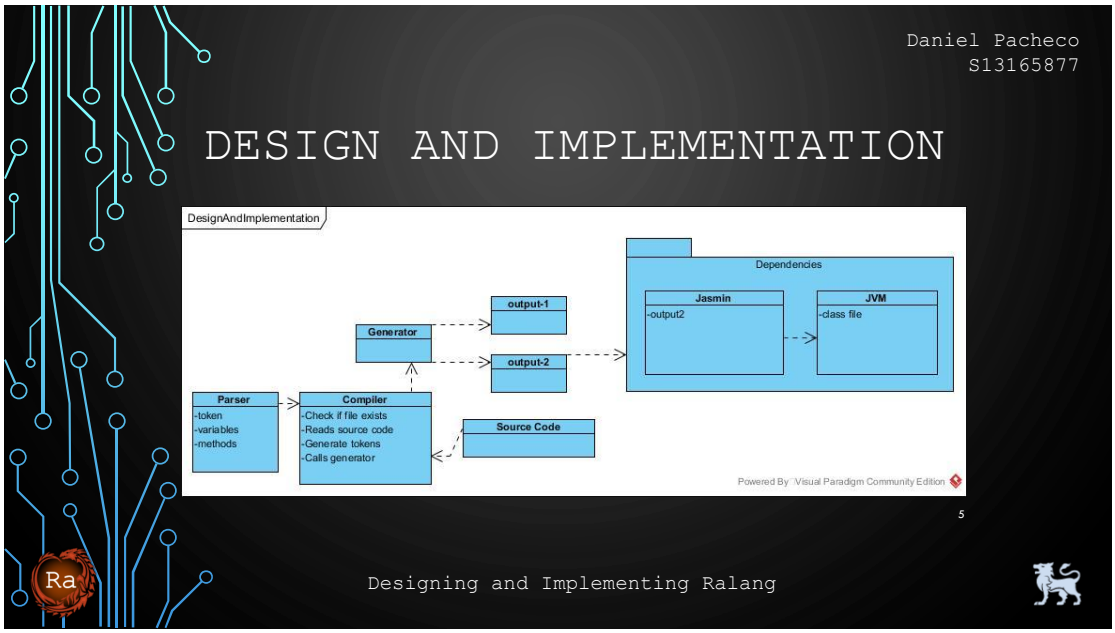


Figure K.5: Presentation slides, demonstration and viva slides - Design and implementation

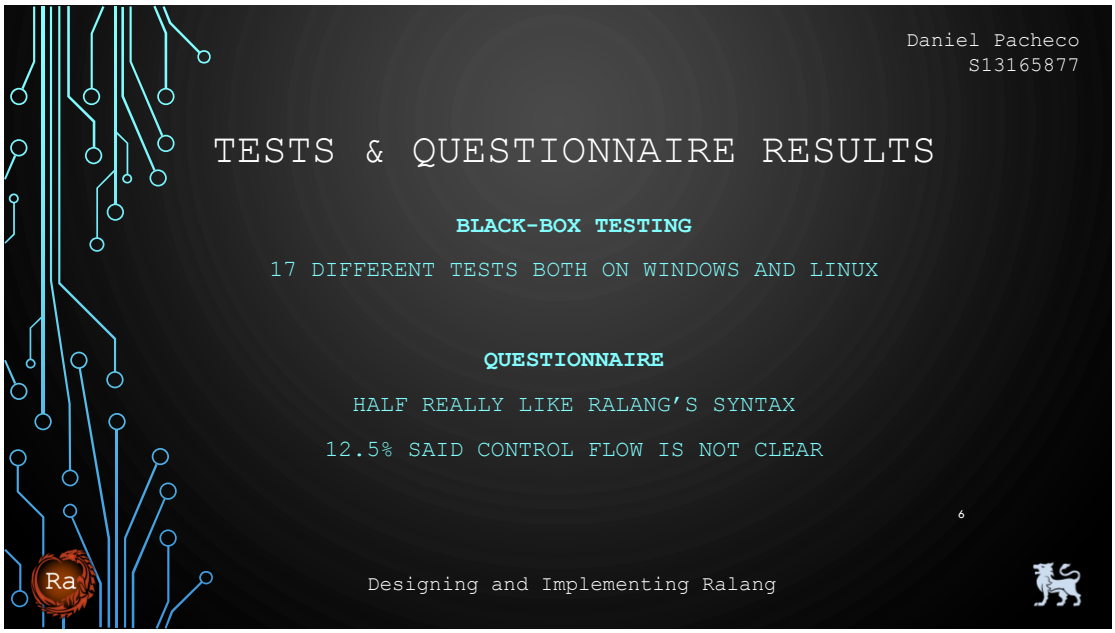


Figure K.6: Presentation slides, demonstration and viva slides - Tests and questionnaire results

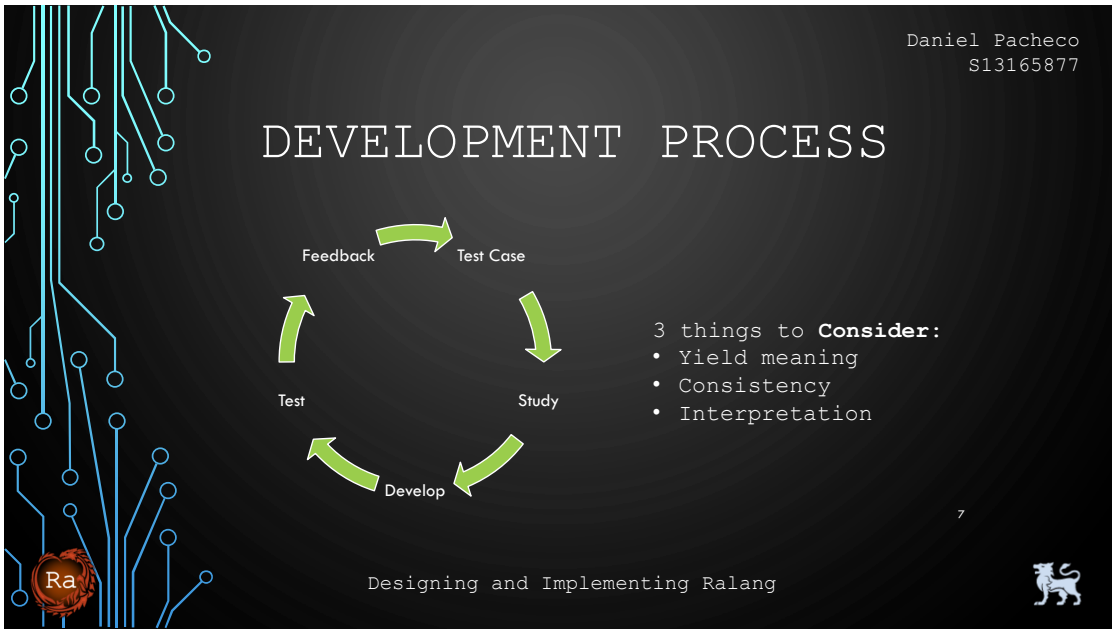


Figure K.7: Presentation slides, demonstration and viva slides - Development process

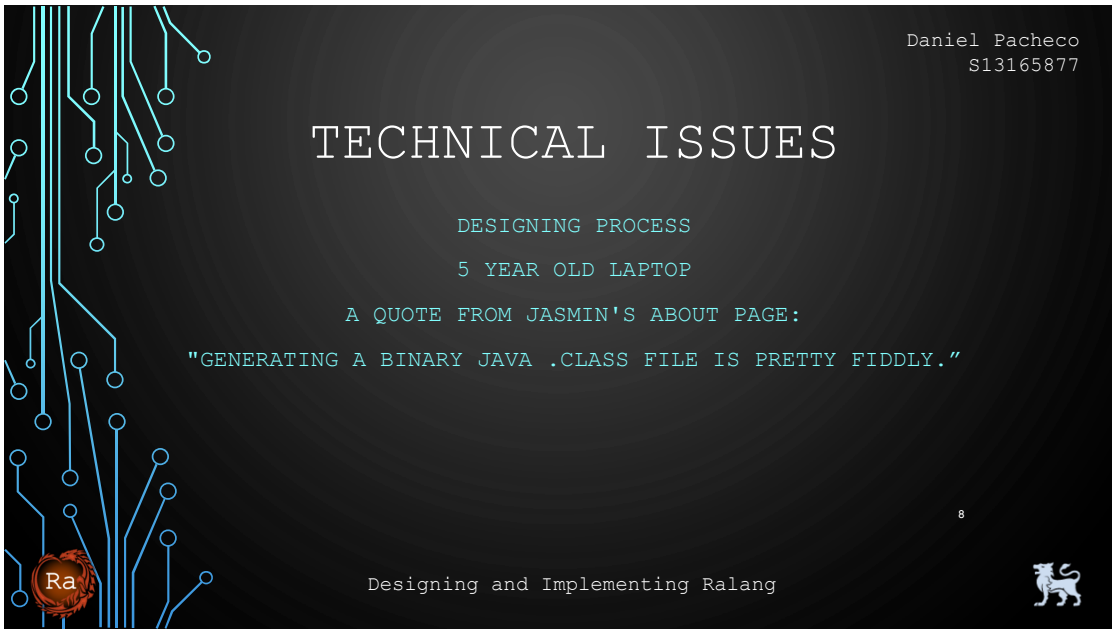


Figure K.8: Presentation slides, demonstration and viva slides - Technical issues

Figure K.9: Presentation slides, demonstration and viva slides - Further work

Figure K.10: Presentation slides, demonstration and viva slides - Resources

K.2 Handout

Pro tip: Ralang relies heavily on the Java Virtual Machines, so ensure you have version 7 or above to be able to run your programs successfully. You will also need Jasmin installed on your machine to generate class JVM files.

Latest Ralang developments:

- Windows and Linux compatible
- Source file checker
- Working parser
- Print statements
- Mathematical expression
- Boolean expression
- Boolean operation
- If and Else branches
- Method declaration
- Recursion
- Variables
- Multiple functions arguments
- Function's returning values
- Multiple functions

Planned developments:

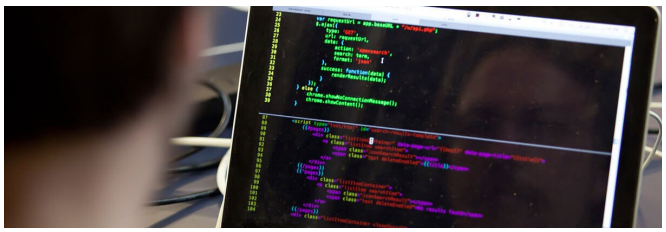
- Uniting testing
- User-defined data types
- Code optimisation
- Tail call recursion optimisation
- Conditions (similar to switch cases)
- Arrays, Sets and Maps
- String manipulation

If you would like to find out more about Ralang, read through the source code or help to improve it, please get in touch:

RALANG

RALANG IS A PROGRAMMING LANGUAGE FOR BEGINNERS

How can I develop my own programs using Ralang?



The process to develop using Ralang is as straight forward as any other programming language.

You begin by choosing your favourite Integrated Development Environment. You may need to look through example source code or the documentation to be able to understand the standards of the language and how to write Ralang code.

Scripts are provided with the language to make it easier for you to compile your programs both on Windows, Mac OS and Linux, whichever one you prefer.

You can then run these scripts which will ask for the source file, and the same script will take care of running the application without any further inputs from the developer.

Whilst the script is running there are a few things happening in the background:

1. Ralang compiler will check if the file you specified exists or not, and will output messages accordingly.
2. The program will get converted into Jasmin code.
3. Jasmin will then read the output code from the compiler, and generate a JVM class file.
4. JVM will then run the program automatically for you.

Daniel Pacheco
ID: s13165877

Figure K.11: Leaflet explaining how to get started with Ralang