1 Project Description

The first section of this report will detail the problem that this work aims to address, the definition of the goals of the project, and the scope of the work. This will re-use and further expand upon the initial project brief.

1.1 Problem Description

The modding community of a considerable amount of video games is large [?], with many developers providing extensive APIs which user-generated scripts can interact with to change elements of the game as they see fit. These may be as simple as an in-game soundboard, to as complex as an entirely new GUI that replaces the original UI of the game. Since games are played by all manner of people, the modding community of any given game can be incredibly diverse, teenagers may develop simple mods as their first foray into programming, or a seasoned programmer could make use of their skills to design a complex mod to enhance their game experience. Many games now support mods [?], from Roblox [?] to World of Warcraft [?].

Mods are written in extension languages, i.e. a language that allows for the creation of programs that are invoked by the host application (game) — no language is more prevalent in this role than Lua, a dynamically typed scripting language designed primarily with this goal in mind. However, it was not designed with game modifications specifically in mind, just as a general-purpose extension language. In fact, its original purpose was for academic projects in the field of geology [?]. Other languages can fulfil this role in games, such as C and Java, however these languages were also never designed with the goal of being an extension language in mind; Lua is by far the most dominant choice in the industry [?].

Therefore, the question remains, can a new domain-specific language (also referred to as DSL) be created that is more suited to the task of game modification development, and as such makes the development of both simple and complex game mods easier for these modding communities?

1.2 Goal

The goal of the project is to design and create a DSL for game modifications that improves upon already existent solutions like Lua. The details of this language, including its syntax, type system and program semantics, will be documented in full, and any design decisions taken will be explained and justified in the report(s). Finally, several mods will be developed in order to show off the capability and suitability of the created solution.

In order to facilitate this work, a literature review will take place to research the topics of DSLs, game modifications and Lua itself. As well as this, code analysis will be performed on existing game modification programs in an attempt to find commonalities between code and issues with the language itself, which can then be considered and remedied in the DSL.

1.3 Scope

Due to the dominance of Lua, the DSL will initially be designed as a wrapper for that language. As Lua's interpreter itself is written in ANSI C [?] (because it designed to easily embed itself into applications via a C API) it may also be possible to easily convert this proposed DSL into C source code using Lua as an intermediary, which would allow the language to support games that use C for modding. Support for any other scripting languages used for mods in games will be considered on a case-by-case basis dependant on the time remaining once a rigorous DSL that works for Lua-supported games has been designed and implemented.

2 Literature Review

The second section of this report will comprise of a literature review on the relevant material related to the topics of domain-specific languages, game modifications and Lua.

2.1 Domain-Specific Languages

A domain-specific language (DSL) is a programming language created with the purpose of performing tasks in a specific domain [?, ?, ?]. According to Tomassetti, these languages will embed ideas specific to the domain into the very language itself, with the general idea that domain experts rather than programmers can then use the language to more quickly produce the required result, as opposed to general-purpose languages (GPLs) like Java and Python [?]. DSLs can be made to be very specific, such as a language designed to build command-and-control simulators for army fire support [?] to a language that is used by scientists in the field of computational fluid dynamics [?]. Of course, these are just two examples and the applications of DSLs are endless. However, the term *domain-specific language* is also very general, and can also be applied to widestream languages that have specific purposes (and are hence still domain-specific) like SQL, XML and HTML [?].

Various different types of domain-specific language exist, namely textual languages, grahpical languages and projectional editors [?, ?].

2.1.1 Textual Languages

A textual language is much more akin to a normal programming language, and is the classical choice. The language can be built by means of a lexer and parser generator, such as Yacc or ANTLR (a non-deterministic parser), and then interpreted, translated or compiled by a hand-written interpreter/translator/compiler [?]. It can also be built by a language workbench such as Eclipse Xtext, which not only generates a parser, but also a Eclipse-based IDE for the language [?, ?]. Voelter argues that using a language framework makes "a qualitative difference" [?], whereas while Fowler "have a remarkable potential", he thinks that "this potential, however profound, is still somewhat in the future" [?].

Due to the connected nature of the proposed DSL to the game's API, it was worth considering the topic of code generation in relation to DSLs. Code generation is simply the stage of compilation where some intermediate representation of the code is used to generate different, but semantically the same code, in a language that can be evaluated by the machine [?]. In the traditional sense, this would be machine code in the compiler of the programming language, but in the case of DSLs, we can essentially *translate* the code into a different programming language. Figure 2.1 is taken from Fowler's book, and visualises this concept.

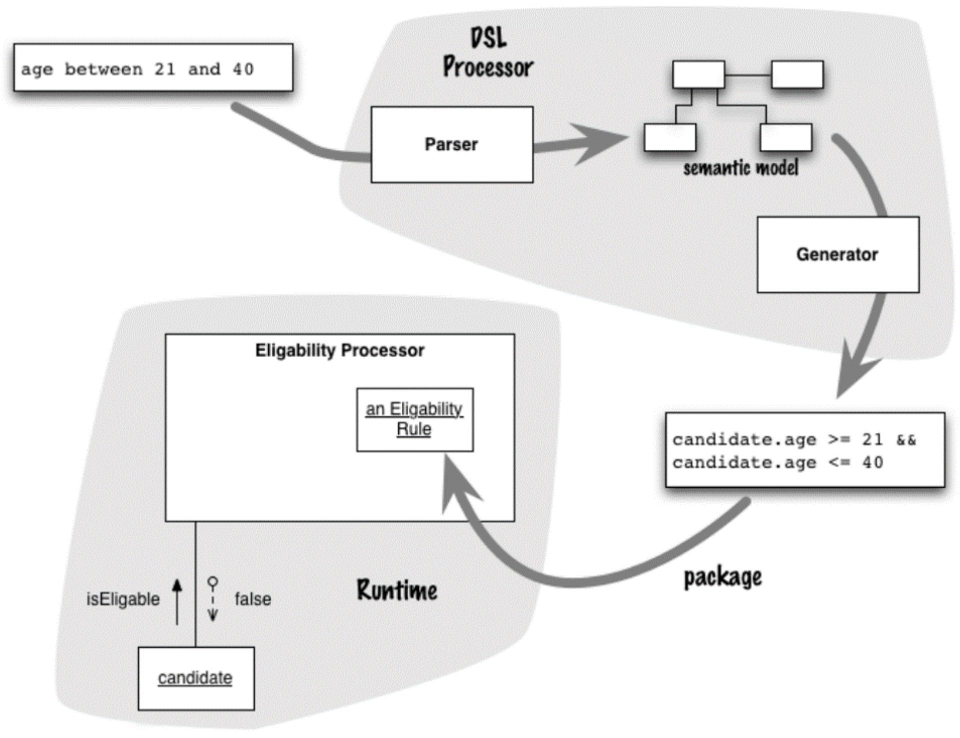

Figure 2.1: A compiler parses the text and produces some intermediate code which is then packaged into another process for execution. Source: [?]

It is also worth discussing a relatively novel type of parser, the **Parsing Expression Grammar**, or **PEG** for short. PEG was first formally proposed by Bryan Ford in 2004 as an alternative to the long-established Context-Free Grammar, that derived from Noam Chomsky's work in the 1950s in the field of linguistics [?]. PEG is an evolution of a parser type introduced in 1970 by Alexander Birman, initially known as the TMG Schema (TS) [?], but later renamed by Aho and Ullman to Top-Down Parser Language (TDPL) [?]. The key difference with PEG in comparison to the traditional CFG is that it aims to use a *prioritised* choice of operator of '/', as opposed to the normal unordered choice operator that CFGs employ of '|'. This means that PEGs are unambiguous unlike CFGs, but as Ford says, "have far more syntactic expressiveness than the LL(k) language class typically associated with top-down parsers", meaning they are more powerful than their TDPL ancestor [?]. Ford initially stated that a proof that CFGs and PEGs define incomparable language classes has been elusive in his original paper in 2004 [?], but even now this remains an open problem, as Loff *et al* discuss in their paper on the computational power of PEGs [?].

Despite the newfound popularity of PEGs, the academic community as a whole is still in disagreement as to what the best solution is to the parsing problem. Parr and Fisher argue that despite the fact that "parsing is not a solved problem" that the LL(*) parsing strategy that version 3.3 of ANTLR uses is as expressive as PEGs whilst also providing "good support for debugging and error handling", and thus propose this is the current "sweet spot in the parsing spectrum" [?]. It is worth pointing out that Parr is the creator of ANTLR, so this should be taken into consideration when deciding on a parser generator. Meanwhile, the father of Python, Guido van Rossum, has recently rewritten Python's parser using PEG, giving multiple reasons justifying the switch such as "it provides more flexibility for future evolution of the language"

[?]. Rossum has produced multiple articles of the subject, which could be a useful guide should PEG be used for the proposed DSL [?].

2.1.2 Graphical Languages

An alternative to the traditional textual approach is to model the language by means of a graphical language, hence the alternative name of **Domain-Specific Modelling** (DSM). This concept has been around for sometime and is often seen as more closely related to UML than it's text-based cousin. As described by Kelly and Tolvanen, the idea of using DSM is to provide a higher level of abstraction since no code has to be written, and to perform the code generation that converts the model produced into something the machine can understand [?]. As Tomassetti points out, the key is that domain experts who are not familiar with programming concepts will find the language more approachable, and easier to understand [?].

There are several different tools that can be used to produce a DSM: MetaEdit+ is an advanced commercial solution that has been around since 1995 [?, ?], Graphical Modelling Framework (GMF) is based of the Eclipse Modelling Framework (EMF), which was first introduced in 2006 [?, ?], and a further evolution of this tool in the form of Eclipse Sirius [?]. Tomassetti postulates that GMF is the most flexible of these tools, but also the most difficult to use [?].

Whilst this method would be the most suitable in certain situations, in this case, the domain experts are already familiar with programming. By producing a DSL of this type, we are merely limiting what the users of it can produce unnecessarily, something that it is crucial not to do due to the varied nature of mods.

2.1.3 Projectional Editors

The final type of DSL is still in its infancy, but also the most exciting — the projectional editor. As described by Fowler, in a traditional DSL, a editable representation is parsed into an abstract representation (usually an abstract syntax tree), which the compiler then uses to generate an executable representation that the computer can understand. In the case of a DSL created with a projectional editor, the abstract representation **is** the editable representation of the system, i.e. an editor exists which projects the abstract syntax tree (AST) into something human readable, be it graphical or textual [?].

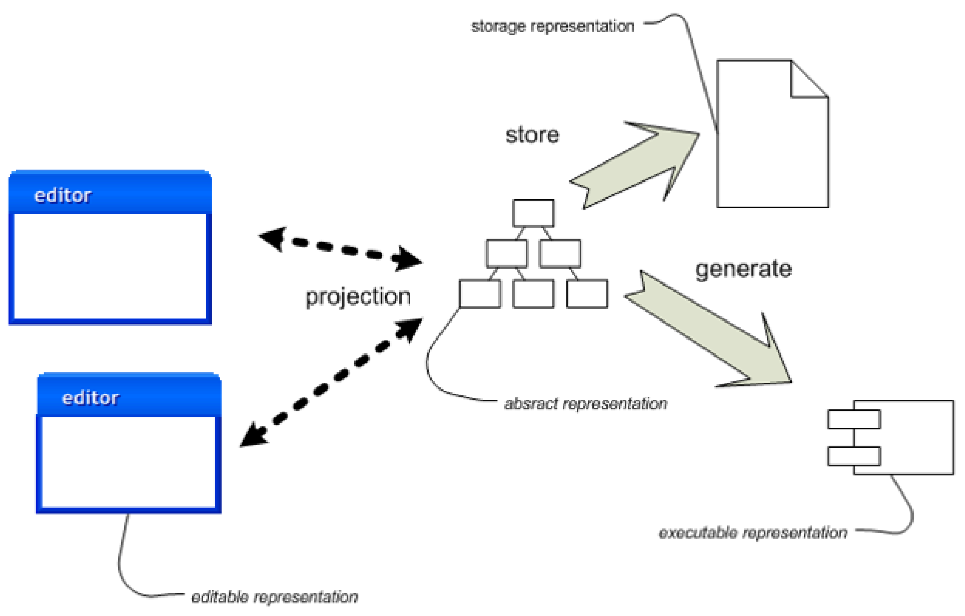

Figure 2.2: A diagram which illustrates the projectional editor model. Source: [?]

The benefit of such an idea is that the abstract representation, what the domain expert is likely to understand, can be modified directly in a similar way to graphical languages. However, in this case, the abstract representation can be modified via multiple different projections since it isn't directly tied to a specific modelling structure. In other words, we can define the same thing in various distinctive ways. [?, ?, ?]. Of course, the major shortcoming is that we're limited to using this projectional editor tool in order to use the language. To illustrate this, figure 2.3, created by Barash, shows (an edited and labelled version of) the if statement being defined in JetBrains MPS in three different syntaxes — Java, Visual Basic and Lisp [?].

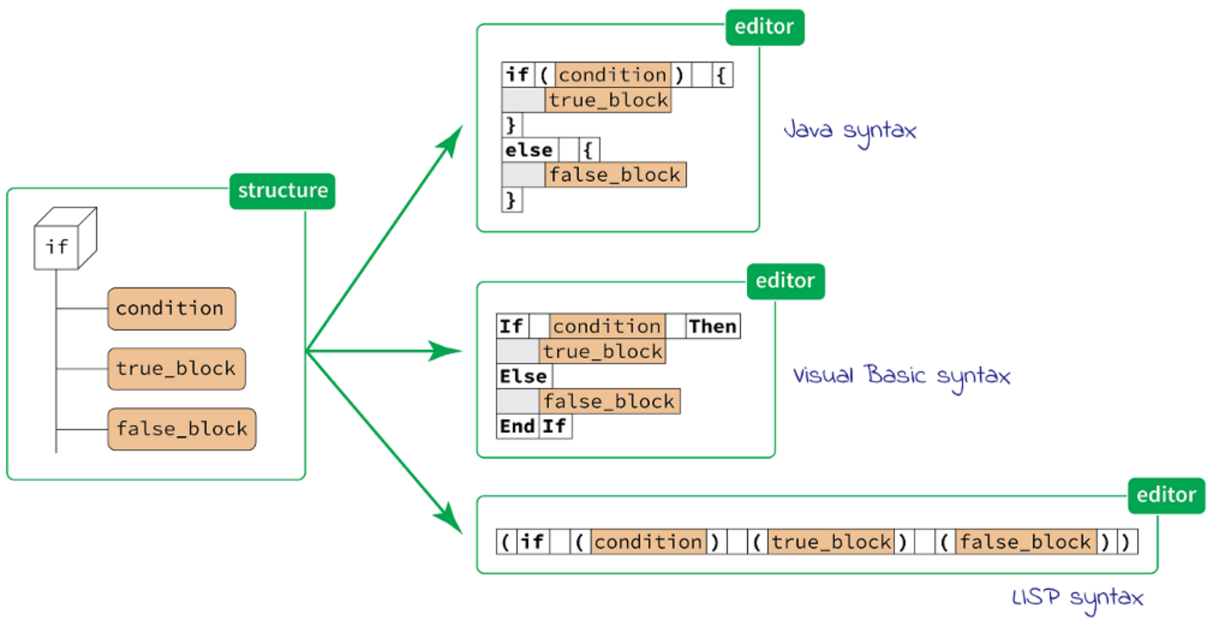

Figure 2.3: An illustrated example of three projections of the `if` structure in JetBrains MPS. Source: [?]

Although other projectional editors exist, such as the Intentional Platform and Whole Platform [?], JetBrains MPS appears to be far and away the most mature projectional editor language workbench [?]. Due to the object-oriented nature of language definition in MPS, it is also very easy to extend [?]. In fact, Voelter argues that with projectional editors like MPS, it now may even be possible to create a GPL and get rid of the entire concept of DSL versus GPL in language engineering [?].

2.1.4 Internal vs External

It is worth briefly discussing the difference between an internal and an external DSL. An external DSL is a language completely separate from the main language of the application it is evaluated by [?].

An internal DSL, on the other hand, can be thought of being more like a *dialect* of another language that it uses. The classic example of an internal DSL is a Lisp dialect, where programming is usually expressed by creating and using a DSL/dialect. The line between an internal DSL and an API is quite blurred, leading to a middle ground concept of a **Fluent API**, as described by Voelter [?].

A Fluent API, as coined by Martin Fowler and Eric Evans, essentially chains method calls in order to create something that is more human-readable as to what each stage is doing; the defined methods effectively make up the commands of the language, which is why it is often considered to be on the same lines of an internal DSL [?].

```java
var integerList = List.of(1, -61, 14, -22, 18, -87, 6, 64, -82, 26, -98, 97,
↪  45, 23, 2, -68);

prettyPrint("The initial list contains: ", integerList);

var firstFiveNegatives = SimpleFluentIterable
    .fromCopyOf(integerList)
    .filter(negatives())
    .first(3)
    .asList();

prettyPrint("The first three negative values are: ", firstFiveNegatives);
```

Code 2.1: Example of a basic Fluent API in Java. Source: [?]

The choice between using an external or internal DSL is one that should be made situationally based on the requirements of the solution. However, some members of the community outright contest that internal DSLs are fundamentally DSLs at all — Tomassetti makes the argument that since internal DSLs do not have tool support, they are not *real* DSLs [?].

2.1.5 Macros

During the research process, a recent paper by Ballantyne, King and Felleisen was found that explored the idea of implementing macros in DSLs [?]. For the uninitiated, a macro is a type of metaprogramming which specifies how an input should be mapped to a replacement output. This usually involves multiple distinct instructions that are then easily called by the programmer by one command [?]. They differ from functions since they are preprocessed — the macro instruction is directly substituted at compile time [?]. Macros are widely used in video games, so adding in the functionality to define them in a DSL designed for game modifications seems only apt.

The paper that Ballantyne *et al* focuses on the programming language of Racket, a language which derived from Scheme, which itself is a dialect of the already mentioned Lisp [?]. Although I feel as though the macro framework that it aims to provide is not appropriate for this project, the description of implementation of the macro expander, which uses a PEG, may be beneficial for implementing macros in the proposed game modification DSL.

2.2 Game Modifications

A game modification (colloquially referred to as a mod) is a user-defined change to the game's systems. In Figure 2.4, we can see the drastic difference that a mod can make in the form of ElvUI [?], which changes the UI of its host game — World of Warcraft.

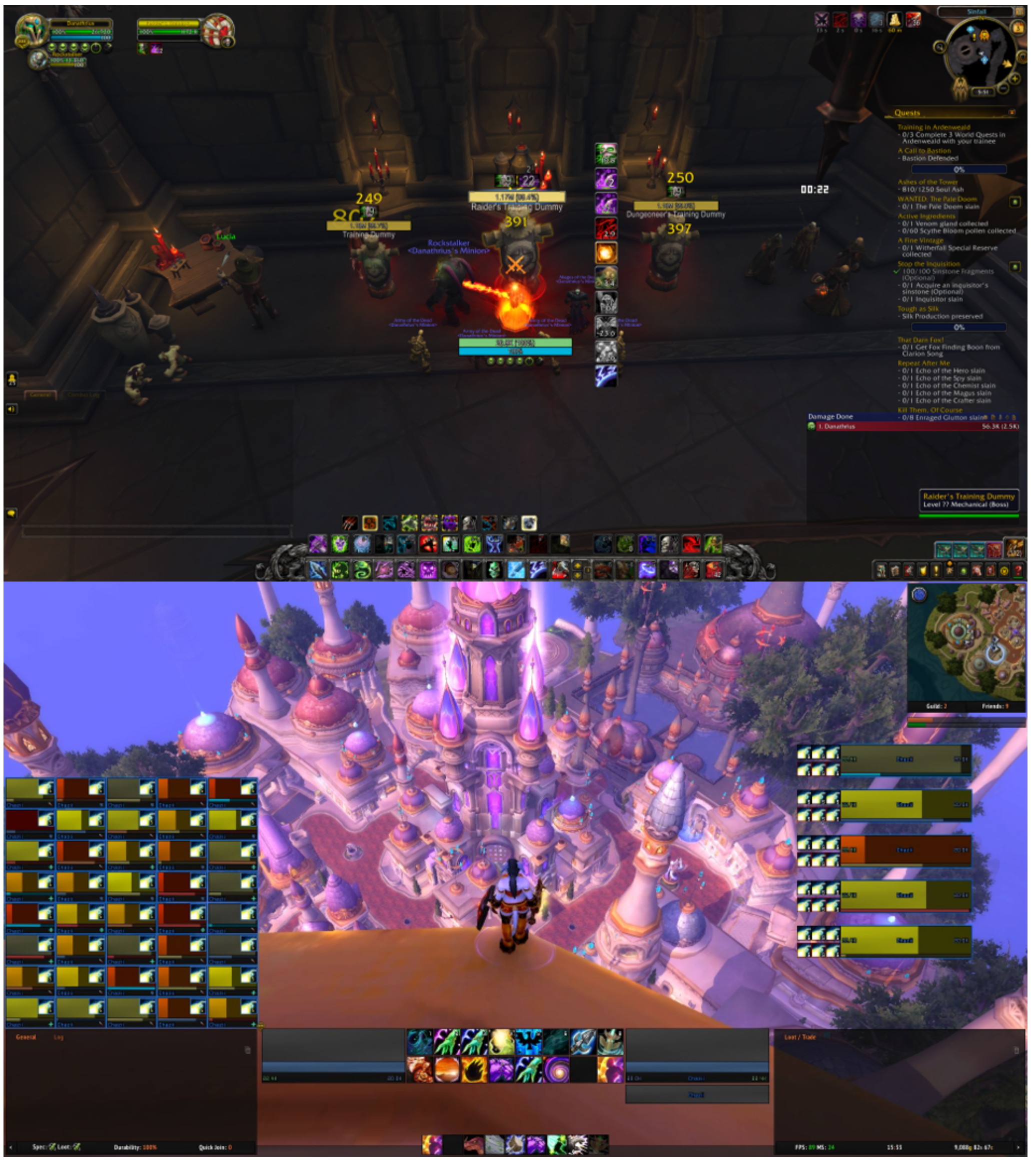
Figure 2.4: A comparison between an enhanced version of the default World of Warcraft UI (top) and the mod ElvUI (bottom). Source: Adapted from [?]

Mods can be anything from a sophisticated modular application that completely changes the game, to a small script which makes a minor change. These modifications may be officially supported by the game's developer, or may in fact be unsubstantiated changes [?]. Despite this, Lee *et al* show through their research that games with official mod support have a higher endorsement ratio, and are likely to increase the longevity of the game ("*it takes a median of 345.5 days before the initial version of a mod is released*") [?]. This incentivises companies into developing support for mods through tools such as in-game script editor.

Due to the sheer variety and number of mods that exist, it is sensible to categorise them into groups in order to better analyse their uses and differences. Scacchi suggests categorising mods into four distinct categories: User Interface Customisations, Game Conversions, Machinima and Hacking Closed Game Systems [?]. During the analysis stage of existing mod code, each sample was sorted into similar categories, although User Interface Customisations was further expanded to the additional categories of Timers and Meters. Furthermore, the Machinima type of modification was not considered during this process, mainly due to the fact that mods of this type were not encountered during this process.

2.2.1 DSLs for Game Modifications

This is not the first attempt to create a DSL for Game Modifications. Both Bethesda (Elder Scrolls and Fallout) and Epic Games (Gears of War and Fortnite) have created their own scripting languages in the form of Papyrus [?] and UnrealScript [?], which are designed to work with their respective engines, Creation Engine [?] and Unreal Engine [?]. Papyrus is entirely event driven, uses the concept of states to appropriately respond to the event occurring in the game at any given time [?]. UnrealScript similarly has the concept of states, as well as timers, but is more similar to a traditional scripting language like Java or C++ [?]. These languages have had varying success though — Papyrus is still used by Bethesda as the scripting language of choice for all of their games [?], but Epic have since scrapped UnrealScript for Unreal Engine 4 [?] in favour of C++, with Tim Sweeney (founder of Epic Games) being quoted as saying: "You know, everything you're proposing to add to UnrealScript is already in C++. Why don't we just kill UnrealScript and move to pure C++? " [?]. This shows that creating a scripting language designed with gaming in mind, and maintaining it, is no easy task.

It is also worth briefly mentioning an internal DSL called MoonScript [?], a language that seems to have been created with a similar aspiration to this project's goal in mind. This language was built by way of the LPeg parser tool [?], so it is definitely worth investigating further if a similar approach can be taken for the game modification DSL.

2.3 Lua

Due to its current popularity as the most used programming language for game modifications [?], it seems only sensible to investigate why Lua is so widely used by researching the language itself.

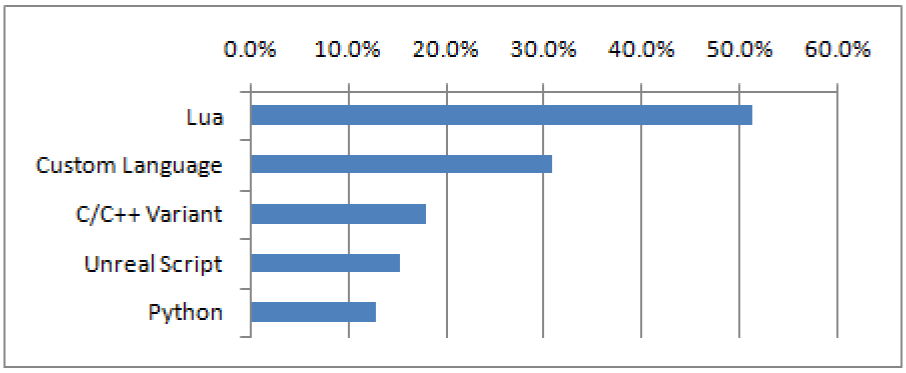

Figure 2.5: Bar chart showing the survey results of the most common scripting language used in game engine development. Source: [?]

Lua was originally created by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes in 1993 at the Pontifical Catholic University of Rio de Janerio, Brazil [?]. It was originally created for academia, designed to facilitate research and work with industrial partners, namely to create a configurable report generator for lithology profiles [?]. As detailed previously, it was primarily designed to be an extension language [?] — designed with simplicity, small size, portability and embeddability in mind [?]. Despite being incredibly lightweight (25,000 lines of C code [?]), Lua is surprisingly versatile, supporting multiple programming paradigms, such as object-oriented, functional and concurrent programming [?].

Lua has three core meta mechanisms that allow it to be so flexible: dynamic associative arrays, reflexive facilities and fallbacks [?]. Lua has no notion of a main program due to its place as an extension language [?], something that is very appropriate for game modification development. It also has a global scope [?], hence the need for the excessive use of the `local` keyword. Functions in Lua are first-class and all functions are anonymous [?]. Functions can also return multiple values which can be assigned separately [?].

An associative array, or a table, is an array that can be indexed by a value of any type, i.e. not just integer values [?]. This means that arrays in Lua are effectively all dictionary-like structures.

```
list1 = {"red", "blue", "green", "white", "black"}
list2 = {"UK"="London", "France"="Paris", "Germany"="Berlin", "Italy"="Rome"}

print(list1[1]) --> "blue"
print(list2["France"]) --> "Paris"
print(list2.France) --> "Paris" (equivalent to the line above)
```

Code 2.2: Demonstration of tables in Lua.

Reflexive facilities refers to the idea that tables are the intended way of maintaining multiple environments, and not just the global environment [?]. This allows for support for object-oriented programming, the ':' syntactic sugar allows for the object itself to be passed to its member function.

```
car = {"make"="Nissan", "model"="Skyline", "year"="1999", "colour"="Blue"}

-- This is the equivalent to...
function change_colour(car, new_colour)
    car.colour = new_colour
end
car.change_colour = change_colour

-- This.
function car:change_colour(new_colour)
    self.colour = new_colour
end
```

Code 2.3: Object-oriented programming in Lua.

Fallbacks are a method of dealing with error handling in Lua — they are an advanced feature designed to deal with errors in code that cannot afford to throw an exception, and are usually specified by the main application developers/kept to the default Lua settings [?]. There is a lot to talk about in relation to fallbacks, but I do not feel they should be considered for the design our the proposed DSL past realising that they exist and need to be factored into the interaction with Lua.

2.3.1 Lua Translator

Upon looking into how it would be possible to generate Lua code or a Lua AST, it was discovered that this will be far more difficult than first thought. This is because unlike most programming languages, Lua has no intermediary state — it translates directly from Lua code straight into C code (specifically a C struct) [?].

This will quite clearly be a problem when designing the language, and further investigation must be done as to how to approach this issue.

3 Background Research

The next section of the report will describe the background research that was conducted in order to facilitate this work. This will include a discussion on a code analysis exercise that was completed in order to first ascertain if there was a need for a DSL, and then to help inform the design of it. Following this, the design philosophy of a questionnaire that will be sent out to various modding communities to ensure the language is fit for purpose will be described and justified.

3.1 Code Analysis

In order to determine if there was even a need for game modification DSL, and then if so, to inform the design philosophy and features of the said DSL, analysis was performed on a substantial number of modifications across a range of video games. The primary goal of this analysis was to identify programming idioms and features that should be present in a game modification DSL; namely due to its current high-usage in the industry, all of the mods that were selected for this analysis were written in Lua. By performing this rudimentary analysis, it is hoped that the produced language will be fit for purpose, and not fit into any pitfalls that would result in it being a worse alternative to Lua itself.

Across eight games (World of Warcraft, Elder Scrolls Online, Sid Meier's Civilization VI, Sid Meier's Civilization V, Garry's Mod, Factorio, Binding of Isaac and Roblox), modifications were sorted into five types: **Game Features**, **UI**, **Timers**, **Meters** and **Hacks**. Mods were analysed independently, but great attention was given to mods of the same type from two different games in order to attempt to find commonality that can be implemented as a feature of the new language.

The following observations were made:

- Apart from mods of the **Hacks** type, most codebases were large and modular.
  - Variables in Lua have a global scope unless the keyword of `local` is defined, resulting in an abundance of that keyword throughout this code.
  - In some games (e.g. World of Warcraft and Elder Scrolls Online), not using the `local` keyword can result in mods clashing with one another when they declare functions/variables of the same name.
- The `if` statement was often used to check if a function successfully returned the expected value. This leads to a lot of unnecessary conditionals and hard-to-read indented code.
- Due to the excessive usage of both `if` conditions and `for` loops, the `end` keyword is also repeated a lot, which seems rather redundant.
- The code is all very event-driven — all of the games appear to provide their own event handlers with the exception of World of Warcraft.
- Settings and profiles are common structures throughout mods of all but the **Hacks** type. These implementations differ greatly based on the nature of the mod, but mods of the same type have similar structures.

- Perhaps rather obviously, all code is deeply rooted in the API of the game it is modding.
- Very often, when a `for` loop is employed, the index value of `ipairs` is discarded with the '_' character.
- There seems to be more in common between code of the same game than between code of the same type but developed for different games. It is quite possible that there are other undiscovered commonalities that are more subtle.

Whilst this exercise was certainly useful in the sense that it led to the realisation that there was a need for a specific DSL, it also highlighted the need to speak to domain experts, i.e. the developers of these mods, in order to gain a better understanding of the shortcomings of using Lua in this context. This therefore informed the decision to send out a survey to various modding communities to acquire more information and ascertain these findings.

3.2 Study Design

To briefly conclude this section, it is worth justifying the study design of the Lua questionnaire that will be sent out to various modding communities in order to obtain feedback from domain experts as to how to create a more tailored game modification programming language. The Harvard University questionnaire design tip sheet was used as a reference guide when writing this questionnaire [?]. The full questionnaire can be found in appendix **??**.

The first two questions ask the participant both how experienced they are as a programmer in general, and how experienced they are with Lua. These questions use a scale of 1 to 7, where the 1 end of the spectrum denotes a Beginner and the 7 end denotes an Expert. An odd number was selected here since a middle category uses provides better data [?]. These two questions were intentionally asked in unison using the same metrics to try and gauge the skill level of the participant without asking for any personal information. It was decided that despite the obvious shortcomings of this approach, i.e. that everyone has a different way of judging their skill level, that this would be better than simply asking them how long they have been programming/programming in Lua for, since the amount of experience does not always correlate with aptitude.

The next question asks the participant for the games that they develop Lua mods for — this was asked in order to attempt to group participants together and more easily pick out commonalities that exist in responses between developers for mods of the same game, but also perhaps more importantly, to find commonalities that are universal across games.

With these initial closed questions out of the way, participants were now asked targeted open questions which will hopefully reap the kind of information that is desired. Firstly, they are simply asked to comment on if Lua is a scripting language for each game they mod for, which is placed first in order to try and get them thinking. Next, they are asked generally what they like and dislike about developing mods in Lua — it is hoped that obvious shortcomings of the language will become apparent by the answers to this question. This is followed by a question that inquires if there any features from other languages that are missed when coding in Lua, which should point out any missing features that may be useful in a game modification development setting. The next question merely asks them generally if there is any extra support they can think of that would help with mod development, the hope being that this will mop up any information that has not been caught by the earlier more targeted questions. Finally, the participant is asked if they have any other comments to ask — this is the only non-mandatory question, since hopefully the other questions cover everything that come to their mind whilst answering the questionnaire.

4 Language Specification

This section of the report will define the requirements of the solution, as well as provide an initial high-level design of the language that will developed in order to meet these requirements. Justification will also be provided as to the reasoning behind this approach above other alternatives. Finally, the evaluation strategy for the project will be briefly described.

4.1 Requirements

Based on the research that has been done thus far, it is clear there is indeed a need for a full DSL to approach this problem. The following requirements are therefore proposed for the system:

REQ1 The language must be able to communicate with the API of each game.

REQ2 The language must have locally scoped variables by default.

REQ3 The language must be translatable into Lua.

REQ4 The language must have error handling for undefined values.

REQ5 Programs in the language should run within 90% of the runtime of the semantically equivalent Lua programs.

REQ6 The language should provide its own event handlers.

REQ7 The language should provide a Java-like or Pythonic for-loop mechanism.

REQ8 The language could provide constructs for defining and using macros.

REQ9 The language could be converted into working C code.

REQ10 The language could provide its own timers.

4.2 Proposed Design

The proposed design is as follows:

- A syntax very similar to Lua that addresses all of the issues found during the code analysis, and any that are provided from the participants of the survey. The syntax **must** be similar to Lua to ensure that documentation still makes sense when calling upon objects and methods from the API of the game. It will also be somewhat Pythonic because the author has the opinion that Python is a high-level scripting language with a nice syntax.

- A **PEG** parser generator tool. A PEG parser was selected over a traditional CFG like ANTLR mostly because of the influence of Python recently switching to a PEG parser, and it is felt that Van Rossum's series of articles on the topic [?] may be beneficial during the implementation phase of this project. The exact tool used will be down to experimentation, but the first tool that will be tried is TatSu [?] due to its extensive documentation and use of Python [?].

It is important to note what it is currently omitted from this design. As of yet, no proposition has been made as to how the produced AST from the parser will be converted into the appropriate Lua code. This is because of the problem described in Section 2.3.1 of this document — further research must be done prior to the start of the implementation phase in order to determine a method to resolve this issue.

4.3 Evaluation Strategy

In order to validate the overall success of the project, it must be proven that each of the requirements elicited in Section 4.1 has been met. To achieve this, the evaluation strategy described hereafter is proposed. The following criteria **must** be met:

1. At least two example modification programs must be created with the language that communicates with the API of the selected games [**REQ1**], tests the default locally scoped status of variables [**REQ2**] and demonstrates error handling for undefined values [**REQ4**].
2. Both of these example modification programs must be translatable into Lua [**REQ3**].

The following criteria **should** be met:

1. Benchmark tests should be run against a program written in the language and a semantically equivalent program in Lua, where the former runs within 90% of the runtime duration of the latter [**REQ5**].
2. A program should be created in the language that uses the in-built event handlers [**REQ6**] and uses a Java-like or Pythonic for-loop structure [**REQ7**].

The following criteria **could** be met:

1. A program could be created that defines and uses a macro [**REQ8**] and uses in-built timers [**REQ10**].
2. One of the previously described programs could be converted into working C code [**REQ9**].

5 Project Plan

The final section of this report will detail the project plan — firstly, an account of all work done to date, followed by a plan of the remaining work, and finally a risk assessment.

5.1 Completed Tasks

The following tasks have been completed so far:

- A **literature review** was carried out (Section 2) in order to gain an understanding of the areas of game modifications, domain-specific languages and Lua. This research not only provided an understanding of the research topics, but also helped inform the design decisions that subsequently were made.
- A **code analysis** exercise was conducted (Section 3.1) where Lua modifications from various games were analysed in order to ascertain if there was a need for a DSL. Rather quickly, it became evident that there is scope for an alternative solution, so the focus of this work instead became on finding common idioms and problematic features that the new language design could factor in/resolve.
- Following this code analysis, it was determined that it would be beneficial to reach out to various modding communities to gauge their opinion on Lua, and how they would improve it. A **questionnaire** was therefore designed in such a way (as described in Section 3.2) that relevant information would be received from the participants. The full questionnaire can be seen in Appendix **??**. In order to send this questionnaire out however, ethics approval (ERGO2) must first be received in order to be able to distribute it. The appropriate applicaiton for this has been submitted and the questionnaire is ready to be released pending approval.
- **Requirements elicitation** was then performed using the SMART (Specific, Measurable, Achievable, Relevant and Time-Bound) criteria and MoSCoW prioritisation system (Section 4.1). These requirements were then used to generate a set of evaluation criteria to determine the success of the project at its end (Section 4.3).
- An **initial design** was then created based on the findings of the literature review and background research.

5.2 Future Work

Further to the completed work, the subsequent tasks will now be carried out:

- Once ERGO2 approval is received, the **survey** in Appendix **??** will be deployed across various modding communities. The results will then be analysed, with this information then informing a revised version of the design.

- When this analysis has been completed, a **formal design** of the language will be defined, composing of the choice of parser generator, grammar, and method of translating the produced AST into Lua code.
- After the design has been specified, the next stage is the **implementation** of the DSL by means of creating the parser and the translator components.
- During the implementation stage, a **type system** will be formally defined for the language.
- Simultaneously, the language's **operational semantics** will also be formally defined.
- Finally, the produced language will then be **tested/evaluated** against the criteria specified in Section 4.1. This will involve creating several demonstration game modifications in the new language that adhere to the criteria set out in the evaluation strategy, as well as performing benchmark testing.

For a description of when each task was completed, refer to the Gantt chart that can be found at Appendix **??**.

5.3 Risk Assessment

A risk assessment was conducted at the start of the project in order to provide counter-measures to mitigate any predicted risks that may occur over the course of the project. This can be seen in Appendix **??**.