NTNU – Trondheim
Norwegian University of
Science and Technology

Implementing the next generation of the Methodical Accelerator Design language using LuaJIT

Martin Valen

Master of Science in Cybernetics and Robotics
Submission date: March 2014
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Assignment

The student shall create an application to be used in the Methodical Accelerator Design-project at CERN. The application shall contain:

- A parser that can read, create and run Lua code. It shall be able to run in both batch- and interactive mode. Syntax- and run time errors should be mapped to the correct line in the original file. The parser shall be easily extensible to parse other languages.
- A unit testing framework to be used in the application and in modules other users create.

If there is time, the student shall also:

- Show the use of the framework by extending the parser to be able to parse MAD, a new language extending Lua. It shall contain C-style include statements and the possibility of creating short anonymous functions that can be called without parentheses if they do not take arguments.
- Show the possibilities of the application by extending it to be able to parse and translate MAD-X files.

Portability to other operating systems (Windows, Linux, Mac-OS) shall be facilitated.

Acknowledgements

Going into a project of this sort with minimal knowledge of the actual workings of a compiler proved to be quite an interesting challenge. I doubt that I've learnt more during any other half year of my studies than I've done during this last one. Being allowed to contribute to the worlds coolest experiment is a major motivation boost, which really helps during those countless times when nothing works, my program hates me and I hate my program.

I would like to give a huge thanks to my supervisor at CERN, Laurent Deniau, who taught me a lot, both about interpreters and compilers, but also about clean and structured code and programming in general. I would also thank him for choosing me for the project, as the half year in Geneva was incredibly interesting, both from an academic and a social point of view.

I would also like to thank my NTNU supervisor, Amund Skavhaug, who has given me great input on how to properly write a thesis, both when writing this thesis and when doing my specialisation project earlier.

Thanks are also in order to my flatmates and other friends from the time at CERN, for taking my mind of my project with quizzes, ski trips, lunches, kings meals and all other things I did down there. In addition to thanking my friends and family for shaping me into the awesome person I am today, I would like to give a special "thank you" to Øystein and Øystein who gave me a couch to sleep on as well as a constant stream of distractions while I finished the thesis.

Trondheim 31. March, 2014.

Martin Valen

Summary

To further the boundaries of human knowledge, physicists at CERN use large particle accelerators to smash elementary particles together at high energies. To design these accelerators, the physicists need a language specially designed for the task, to allow them to spend their time doing physics instead of programming. The Methodical Accelerator Design (MAD) project at CERN is developing this kind of languages. The author of this thesis was chosen to cooperate in this project to aid in the implementation of the new iteration of the MAD language. A new version is needed, because the old version (MAD-X) has several problems that will make it unsuitable for the next generation of accelerators. The project this report is based on is an application able to run code written in Lua, the new language MAD, and MAD-X. The implemented MAD application is written in the Lua language and run using the excellent Just-In-Time compiler LuaJIT.

The application contains a parser constructor factory and a generator factory to facilitate addition of different languages. Using the factories, the user can chose between three different input languages and two different output languages. The parsers are implemented using LPEG, a module creating top-down parsers. With the help of LPEGs regex module, the grammars of the implemented languages are written in a syntax closely related to Extended Backus Naur Form. The parsers then create an Abstract Syntax Tree that is used by the generators to create Lua or MAD code. The Lua code can then be run using LuaJIT, while the MAD code can be used to debug the application.

To make the application fully able to interpret all Lua files and act as one would expect interpreters to act, a module for correctly mapping errors to their correct lines is implemented. The implementation utilises a map that contains all the generated codes lines and their corresponding lines in the input code.

The application is extended with support for a new language, MAD, which is an extension of Lua. MAD contains support for C-style include statements and a short function syntax for functions that can be lazily evaluated. MAD is also extended to be backwards compatible with MAD-X, by translating MAD-X code into Lua code.

A stand-alone framework for unit testing is implemented, allowing all modules in the MAD application to be tested quickly and easily. The framework is inspired by the pre-existing BTD and Lunit-frameworks.

The future works of the application are discussed and solutions to potential

problems are suggested. The interpreter is tested against other Lua interpreters and is shown to be able to compete on an equal basis. The application is also shown to be easily extendible, helped by the unit testing framework that easily allow the module designers to test their functions and APIs. The implemented application, which can be found at [8], can therefore be of help to the MAD project and thus also to accelerator physicists in general.

Samandrag

For å fremme kunnskapen til menneskeheita, bruker fysikarane på CERN store partikkelakseleratorar for å knuse elementærpartiklar saman ved høge energiar. For å designe desse akseleratorane, treng fysikarane eit programmeringsspråk som er spesiallaga for jobben, sånn at dei kan bruka tida si på fysikk i staden for på programmering. Methodical Accelerator Design (MAD)-prosjektet ved CERN utviklar denne typen språk. Forfattaren av denne rapporten blei valt ut til å delta i dette prosjektet for å hjelpe til med implementeringa av den nye iterasjonen av språket MAD. Ein treng ein ny versjon, då den førre versjonen (MAD-X) har fleire problem som gjer den upassande til den neste akseleratorgenerasjonen. Prosjektet denne rapporten er basert på omhandlar ein applikasjon som kan køyre kode skrevet i Lua, det nye språket MAD, og MAD-X. Den implementerte applikasjonen er skrevet i Lua og køyrest ved hjelp av den utmerka Just-In-Time-kompilatoren LuaJIT.

Applikasjonen inneheld ein parser constructor-fabrikk og ein generatorfabrikk for å fasilitere innlegging av forskjellige språk. Ved hjelp av desse fabrikkane kan brukaren velge mellom tre forskjellige inputspråk og to forskjellige outputspråk. Parsarane er implementert ved hjelp av LPEG, ein modul som lagar ovanfrå og ned-parsere. Vha. regexmodulen til LPEG kan grammatikkane skrivast i ein syntaks som er særs lik Utvida Backus Naur Form (EBNF). Parsarane lager så eit Abstrakt Syntaks Tre som blir brukt av generatorane til å lage Lua- og MAD-kode. Luakoden kan bli køyrd vha. LuaJIT, mens MAD-koden kan bli brukt til å debugge applikasjonen.

For å gje applikasjonen mogelegheita til å tolke alle Luafiler og opptre som ein vil anta at tolkarar skal opptre, er ein modul for å lenke feilmeldingar til deira tilhøyrande linjer implementert. Implementasjonen bruker eit kart som inneheld alle utgongslinjene i den genererte koden og alle deira korresponderande linjer i inngongskoden.

Applikasjonen er utvida med støtte for eit nytt språk, MAD, som er ein utviding av Lua. MAD støtter include, brukt som i C, og ein kort funksjonssyntaks for funksjonar som kan bli evaluert seinare. MAD er og bakoverkompatibel med MAD-X, ved at det kan oversettje MAD-X-kode til Lua-kode.

Eit frittståande rammeverk for einhetstesting er implementert, noko som let alle modular i MAD-applikasjonen bli testa på ein kjapp og konsis måte. Rammeverket er inspirert av dei allereie eksisterande BTD- og Lunit-rammeverka.

Det framtidige arbeidet til applikasjonen er diskutert og løysingar til mogelege problem er føreslått. Tolkaren blir testa mot andre Lua-tolkarar og viser seg å kunne konkurrere på like fot. Applikasjonen viser seg å vera enkel å utvide, godt hjulpe av einheitstestingsrammeverket som gjer det særs enkelt for moduldesignere å teste funksjonane og APIa sine. Den implementerte applikasjonen, som kan finnast på [8], kan vera av stor bruk for MADprosjektet og dermed og for alle slags akseleratorfysikarar.

Contents

List of Figures

Listings

Chapter 1

Introduction

Since its founding in 1954, researchers at The European Organization for Nuclear Research, CERN, have been working together to unravel the mysteries of the cosmos. The main focus has always been on the building blocks of the universe, resulting in its researchers being the first ever to create anti-atoms and, more recently, resulting in the discovery of the Higgs Boson ([4] and [1]) which resulted in a Nobel Price for Englert and Higgs. Notable side effects include the birth of the World Wide Web and 4 Nobel prices to CERN. To be able to perform these experiments, larger and larger accelerators are needed. The largest so far, the Large Hadron Collider (LHC), clocks in at 27 km in circumference. CERN has several upgrades for the LHC in the pipeline, the most important being the High-Luminosity LHC, which will increase the number of collisions by a factor of ten from 2020. Other, smaller upgrades are being applied now, as the LHC is in its first long shutdown period (LS1), during which many improvements will be made to increase the collision speed to 14TeV at center of mass.

But the LHC is only the last of a long line of circular accelerators that have been used at CERN, most of which are now included in the pre-acceleration steps of the LHC, as one can see in Figure 1.1. The LHC still has its limitations and to reach even higher luminosity and energies, newer and better accelerators have to be built. The two main ones being researched now, are the Compact Linear Collider (CLIC) and the Future Circular Collider (FCC). CLIC is going to be a 50km linear collider, achieving the same energies as the LHC, but colliding electrons and their antiparticles, giving researchers a new angle of input. The FCC will be a circular accelerator of 80-100km circumference, using the LHC as a part of its pre-acceleration.

To design an accelerator and to simulate particles moving through it, one will need a physics module, to decide how the particles will interact with the magnets and be acted on by the exterior forces, and a simulation framework to do the simulations. Both of these can be written in FORTRAN or C and as long as they both have clear API, it is possible to write the accelerators by hand in C/FORTRAN and get good results. When the designed accelerators grow in size, it becomes harder and harder to optimize/build them by hand, and one might need to add

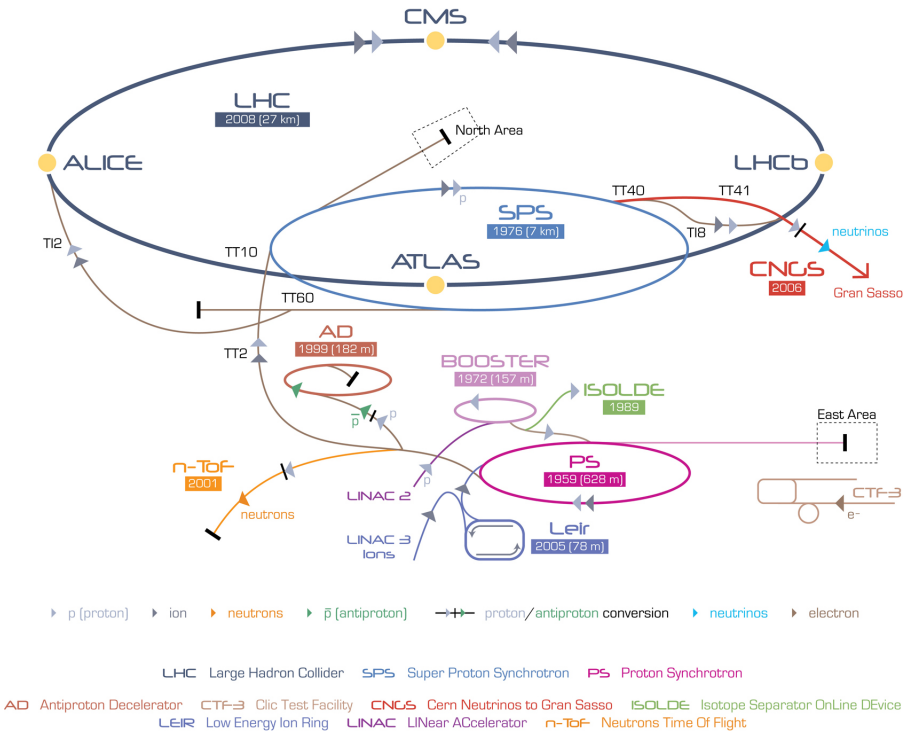


Figure 1.1: CERNs current accelerator complex. [cern.ch]

some more modules to add e.g. optimization. Even later, when building the accelerators, it might be practical to have a database containing the exact positions of all the magnets, all the different strengths, and in general all the information about them. The framework that has been made will need to communicate with all these modules in addition to all other modules one might need. The more elements added, the more does the designing of accelerators move from being a physics- to a programming-related task. By developing a language with the modules as a base, where one can easily design a simple accelerator to test and optimize, the physicists can spend more of their time solving physics related problems and less solving programming related problems.

Through the years, CERN has developed several languages to solve this problem and let the physicists care about physics, not programming. These are gathered under the umbrella term Methodical Accelerator Design (MAD) and most notably contain the deprecated MAD8 and MAD9, as well as the current MAD-X. MAD-X, which has been around since the start of the millennium, has become a de facto standard for accelerator physicists from all around the world and is the language the LHC was designed in. But because of its abundance of 'known bugs are features', memory leaks, non-modular construction and in general poor design, it has been

decided that a new MAD will need to be made. New MAD[1] will have a new syntax based on LuaJIT and should have a better structure making it easier to maintain and extend. It should also be able to read MAD-X files and translate them to MAD-ξ.

Lua[25] is a scripting language designed, implemented, and maintained by the Pontifical Catholic University of Rio de Janeiro, Brazil. It is used in several industrial applications and is the leading language in game development, including well known games as World of Warcraft and Angry Birds. In addition to it being fast, reasons for it being largely used include that it is easily embeddable because of its small size and high portability, as well as having the liberal MIT-license. However, a language such as MAD will need large amounts of computations for its physics parts and will therefore need even more speed than Lua can achieve. Mike Palls LuaJIT[39] is another implementation of Lua, but implementing it as a Just-In-Time compiler. This allows for a significant speed boost, outperforming Lua by a factor of 3-100, depending on how the code is set up[2]. Just-In-Time compilation is a technique that gives a program both the benefits of an interpreted and a compiled language. It works as an interpreted language from a black box point of view, that is by not compiling the code before running it, but in its internals it compiles the code dynamically just before it is run, allowing for the speed of compiled code. It can actually be faster than pre-compiled code in some cases, as a JIT-compiler may dynamically compile the code with run-time information, allowing better optimization than a pre-compiler would. LuaJIT is, according to its own homepage, widely considered to be one of the faster dynamic language implementations and may therefore bring MAD the speed it needs.

This report is written with two main goals in mind. It is to be used as a side-kick to the documentation of the implemented MAD application, to aid others trying to implement new features or fix bugs that might be in the code. It will therefore not only contain the actual implemented features, but also what was attempted to be implemented and why it was not used. It is also written for anyone attempting to start a project similar to this one, showing different approaches to the implementation and pitfalls one might encounter. The report is written mainly for readers with a background closely related to the authors[3], but it shall also be possible to read the report from a physics background. It will therefore contain introductions to subjects that might be elementary to the average computer scientist, especially concerning language theory and compiler structure. The author will attempt to make as few assumptions as possible about the reader, but it is assumed that the reader has some background from C or similar languages. No knowledge of Lua or MAD-X is required, but it is assumed that the reader knows enough about programming languages to understand smaller Lua code snippets

[1] The proper name has yet to be determined, potential names being MAD, New MAD, MAD-NG (Next Generation), MAD-11, MAD-E, MAD-XI, and MAD-ξ. Many of these names will be used in this report and in the attached code, most often just MAD. I.e. if it's not specified to mean MAD-8/9/X, it means new MAD.

[2] Several Lua functions are not JIT-compiled, making the programs significantly slower. `http://stackoverflow.com/questions/7167566/luajit-2-optimization-guide` contains a good optimization guide for LuaJIT.

[3] Cybernetics and embedded systems

without too much explanation.

To achieve these goals, the report will begin with an introduction to languages and grammars for languages. To provide some motivation for this project, MAD-X is introduced. Basic syntax and usage is shown, before the problems and suggested solutions for fixing them are introduced. Following some of the suggestions, a background in compilers is given, showing some previous work concerning Abstract Syntax Trees, lexers and parsers, before finally focusing on Parsing Expression Grammars and how to create grammars following this form. After the background, it will be shown how the created application was implemented. The implementation chapter will begin by introducing projects either extending Lua or facilitating the extension of Lua and how they manage it, before moving on to develop the structure of the applications code. The parser specific elements Abstract Syntax Tree and Lua grammar will be described, then it will be shown how the new framework for unit testing used in the application was developed. The report will then go on to show how source code generation, interactive mode and correct error reporting was managed. As those elements conclude the main work on the application, it is shown how the framework can be easily used to extend Luas grammar and create a new language, MAD, containing some simple extensions. The report then goes on to show how one can implement support for languages completely different from Lua, by adding support for MAD-X. Once the MAD-X interpreter has been fully described, the implementation chapter is wrapped up by showing how the final kinks of the application are handled, as well as giving an introduction to the work that can be done to improve the application. The report then shows how the application fares during testing, testing its speed compared to other applications interpreting Lua code. Finally, the entire works viability is discussed in the discussion-chapter, before the work is concluded in the conclusion-chapter. The appendices show how to install MAD and LuaJIT, explanations of the different command line arguments allowed by the application, as well as showing the designed Abstract Syntax Tree definition, the definition of the SPS-lattice and the entire grammar of the Lua language.

Chapter 2

Background

To understand the motivation for this project a brief introduction to the MAD languages' history and current status will be given, as well as a brief introduction to its syntax and use. To simplify the discussion around the languages, this section will begin with a short explanation of languages, grammars and the Extended Backus Naur Form. This is followed by the MAD introduction, before the chapter is conluded by an introduction to compilers and some examples of how these can be implemented.

2.1 Languages and grammars

To be able to communicate, everyone needs to be able to use some sort of communication, whether oral, written or bodily, that one can ve certaub the other part will understand the same way you intended it to be understood. To do this, languages are perfect. In a perfect world, a language is such a system that when someone has conveyed their meaning, everyone who knows the same language will understand that exact meaning. Unfortunately, this is rarely the case, as languages evolve over time and regions, which may give rise to many unfortunate situations[1]. Fortunately, computer languages haven't had time to evolve as much as human languages and their region is more or less the entire world, because of the internet, and this section will therefore deal with the perfect world-kind of languages if nothing else is noted. To be able to properly describe a language, both for computers and for humans, one will need a definition of what is allowed and what is not, otherwise one would be allowed to take a random stream of letters, put in some spaces and call it a sentence. This is what grammars are for. A grammar is something one can turn to if one needs to check if a string of letters is a proper construct in a given language. How to properly describe a grammar isn't something that is defined, as the best way of making a grammar will depend on the language the grammar

[1] Like when you call a baby cute, while the parent understands it as if you called the baby a weirdo (snål). Or someone calling someone a sweet girl, while the girl understands that she's being called a slut (tøs).

should define. An example of a grammar for a language where one is only allowed to write 'hello world' could be something as simple as the string `'hello world'`. A more advanced grammar could be as seen in listing 2.1.

Listing 2.1: A simple grammar

```
while get word do
  if word is not in list_of_legal_words do
    return false
  end
end
return true
```

This grammar will allow a stream of words, but only if the words are contained in a predefined list of words, thereby allowing e.g. the word 'hello' but not the word 'fweFW'[2]. By extending this grammar with different classes of words (`noun`, `verb`, ...) one would almost be able to describe the entire english language. But to add the different classes one will need an easier way of describing the language, as it will get too convoluted otherwise. Going into all the different ways of describing the grammars will be too long for this report, hence only Extended Backus Naur Form will be introduced and used.

2.1.1 Extended Backus Naur Form

EBNF is a way of describing context free grammars first introduced by [46] and later standardised by the International Organization for Standardization in ISO/IEC 14977. Its symbols are defined in table 2.1.

definition	=
concatenation	,
termination	;
alternation	|
option	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-

Table 2.1: Standardised symbols for EBNF.

An example of its usage is shown below, describing a simple definition of a number.

[2]This grammar already does the assumption that a word is a string of characters from the latin alphabet, as going further into the different alphabets will unnecessarily lengthen this report.

```
decnozero = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
    '9' ;
zero      = '0' ;
decimal   = zero | decnozero ;
number    = ['-'], decnozero, {decimal} | zero ;
```

This defines the `decnozero` which can be all decimal characters excluding zero, and `zero` which is zero. A single `decimal` is then defined to be `zero` or `decnozero`. A number is therefore a decimal without zeroes, followed by zero or more decimals, it may also be preceded by an optional minus sign. If it does not fit this description, it may also be a single zero. One may also note here, the use of `terminal` and `non-terminal`. A terminal is a rule that does not exist on the left hand side of the grammar, as e.g. a single string. A terminal is a rule that exist on the left hand side of the grammar, and may therefore be exchanged in all points where it occurs.

2.1.2 Context/Context free grammars

Using EBNF, one could write the grammar taking a stream of legal words as the following.

```
sentence    = {legal_words} ;
legal_words = legal_1 | (* omitted for brevity *) | legal_n ;
```

This should be enough to write a grammar for most languages, but it will be hard to get some meaning out of it, as most languages are context dependent. The sentence 'I'm over it' by itself can mean both that you are standing above something and that you are not bothered by something any more, depending on the context in which it is said. This is something that is really hard to code into a grammar, as the grammar in itself doesn't understand what it is parsing, it only checks if it is an allowed construct. To actually use a grammar for something useful, one will need to connect a meaning to the different rules. In a context free language, a set string of characters will have the same meaning no matter where it is used, making it easier to attach meaning to the different rules.

2.1.3 Lexical elements

So far, it has been assumed that a word is a sequence of alphabetical letters, not containing spaces, which is something most humans easily assume, while a computer will have to be told. To achieve this, the stream one receives is split into lexical tokens. This can be done several ways, and most grammars will not contain the lexical definitions, but they are still very important. A grammar will use lexical elements as `number` and `word`, meaning that these will have to be defined somehow as well.

```
number = decimal   , {decimal} ;
number = decnozero , {decimal} ;
number = decimal   , {decimal}, '.', {decimal} ;
```

These are all valid lexical definitions of a decimal number, but which definition is chosen will greatly influence how the language works.

2.2 Methodical Accelerator Design

MAD-X is a language written by physicist for physicists to be used to design particle accelerators. It is the successor of MAD8 and MAD9, which are both still used in the older accelerators at CERN. It is widely used all over the world as the de facto standard for accelerator design and its user base is as big as a user base for something as peripheral as an accelerator design language can be.

2.2.1 Syntax

In this section, the syntax and use of MAD-X will be described. As it is meant as a quick pointer to how MAD-X is used, to make the later parts understandable for the reader, all parts of the syntax will not be shown and several sections will be overly simplified. If the reader wishes to know more about MAD-X, the MAD-X primer[16] gives a good introduction. MAD-Xs syntax is very simple at first glance[3], as one can see in listing 2.2.

Listing 2.2: EBNF grammar of MAD-X

```
chunk     = {stmt, ';'} ;
stmt      = assign | defassign | labelstmt | cmdstmt |
    linestmt | macrodef | macrocall | ifstmt | repeatstmt |
    whilestmt ;
labelstmt = name, ':', name, [','], attrlist ;
cmdstmt   = name, [','], attrlist ;
attrlist  = {attr} ;
attribute = defassign | assign | expr ;
defassign = name, ':=', expr ;
assign    = name, '=', expr ;
```

MAD-X is a context dependent language, but all different contexts will not be explained in this section, as this explanation exists to give a short overview, not to teach physicists how to use the language.

Most importantly, MAD-X is not lexically scoped, meaning that all variables and constants created can be reached from any part of the program, including from different files[4]. Another main part of the language is that it is case insensitive.

A `labelstmt` will store the return of its `cmdstmt` in a variable named the value of name_1. The command is a constructor, making name_1 become a child of

[3]The definition of if, while, repeat, and expr have been left out for brevity. Someone coming from C/C++ will understand them at first glance.

[4]There's no telling how many different ways this can break a program. If one wishes to use a variable name in a module, but someone else have used the same variable name in their module, this will create huge problems for the poor physicist trying to use both modules, believing that they are both without side effects.

name_2. name_1 can then later be called as the command for other label statements, spawning new children. Name_ 1 becomes a clone of name_ 2, with the additional attributes from attrlist.

A `cmdstmt` is quite similar to a function call in C. It will execute name_1 with its attributelist as arguments.

An `attrlist` is a table containing the different attributes. An `attr` is either a key/value-pair, or a single value to be stored in a numbered storage (like a C-array).

`assign` works as one would assume, assigning the value of its expression to the name.

`defassign` will create a new deferred expression. A deferred expression will work as a variable from the users side, but every time it is used, the value of its expression will be re-computed. This is useful e.g. for variables containing random numbers.

Listing 2.3: A sequence defined in MAD-X syntax

```
LQP = 1;
LDP = 0.5;
QPp: QUADRUPOLE, l = LQP, k1 = 0.2;
QPm: QUADRUPOLE, l = LQP, k1 = -0.2;
DPp: DIPOLE, l = LDP, k1 = 0.1;
DPm: DIPOLE, l = LDP, k1 = -0.1;
BAD.Accelerator: SEQUENCE, LENGTH = 3;
QPp1: QPp, at = 0;
DPp1: DPp, at = 1;
QPm1: QPm, at = 1.5;
DPm1: DPm, at = 2.5;
ENDSEQUENCE;
```

The example in listing 2.3 shows the definition of a sequence of magnets, i.e. a particle accelerator[5]. This example creates four classes of magnets; two quadrupoles and two dipoles, both with opposite bends. Then it creates a sequence of magnets named `BAD.Accelerator` of length 3. Since `sequence` is a special command in MAD-X, one that changes the context, the magnets created between a `sequence`-command and an `endsequence`-command will be created as global variables in addition to being added to the sequence that is being created.

Another way of creating sequences is the MAD8/9-way, using the command `line`.

```
linestmt = name ':' 'line' '=' linector ;
linector = '(' ( name | linector ) {',' ( name | linector )
    } ')' ;
```

As long as everything under name is either a magnet or another sequence, this will create a sequence as well. The positive thing with this approach is that one can do arithmetics on the sequence, by multiplying it to get several repetitions of the same

[5]The author hereby allow anyone to use this accelerator, as long as he is not held responsible when it fails to work.

sequence after each other, or by using the unary minus-operator to invert it. This allows for more elegant creations, e.g. allowing the design of the entire 7km Super Proton Synchrotron (SPS) in 9 lines, see Appendix C: MAD-X Example Sequence. It is deprecated and not used any more, but for MAD-X to be compatible with its older versions, it has to support it.

A powerful, oft used, feature is the special command `macro`.

```
macrodef  = name ['(' {string , ','} ')'] ':' macro '='
    '{'{.}'}';
macrocall = 'exec' ',' name_macro '(' {string , ','} ')';
```

`Macrodef` will save the string and the parameters to the variable named the value of name, while `macrocall` will call the macro named name_macro with the arguments given. When a macro is called, the MAD-X-parser will read the entire macrostring and exchange every occurrence of parameter_i with argument_i, then parse it as MAD-X-code. This is a powerful feature, but it is also dangerous. If one e.g. calls the parameter i, one will also exchange the i in a while, which will, most likely, break the code.

2.2.2 Problems

[7] contains a list of problems with MAD-Xs current implementation, which will be briefly shown here.

Data management

Because of its design, there will be a lot of memory leaks during its usage. This is partly because it was originally designed to run one-shot jobs, where one could let the OS handle the garbage collecting, but this has changed after introducing macros, leading to several severe cases of leaking. To improve the speed of data lookup, the core of the application sets a lot of global variables, i.e. leading to all modules having side effects. The problem with side effects, that is made even worse by all variables being global, leads to loads of well known user tricks of the type "If your program doesn't work when you write 'use, sequence1;', then write 'use sequence2; use, sequence1;'". These tricks should of course be unnecessary and will pollute the code.

Interpreter design

The interpreter is designed as a strict string interpreter and does not contain neither a lexer, parser, nor an Abstract Syntax Tree, which is common in interpreters (see figure 2.1), as will be shown in the next section about compilers. As a direct consequence of this, other fields of the application, e.g. the physics, have been conditioned to work with it. An example of a problem that arises from this is that a logical expression can contain many less than/greater than etc., but only one and (&&) or or (||). Another is that the constructs if, while, and repeat are only able to contain one level of if/while/repeat within their blocks. The grammar is also

ambiguous, which allows for silent failure in many different cases. If one adds an extra comma to certain commands which are supposed to remake global tables, the command will silently fail and the old table will be kept, on account on it being global.

2.2.3 Conclusion

[7] lists features the current MAD-X is lacking that will need to be implemented in the new MAD. It also comes with a suggestion as to how this can be achieved in a new application, which follows the structure of figure 2.1.

2.3 Compilers

In the very dawn of computer science, programs were most likely written in assembly code, but as the needs of the programs grew larger, one needed to have higher level languages to be able to keep the code maintainable. This lead to the need of compilers. [2] defines a compiler as something that can read a program in one language and translate it to an equivalent program in another language. The other language can be binary to create executables or another language to be run or read later. A second language processor defined by [2] is the interpreter, which reads a program in a language and executes it, instead of creating another program.

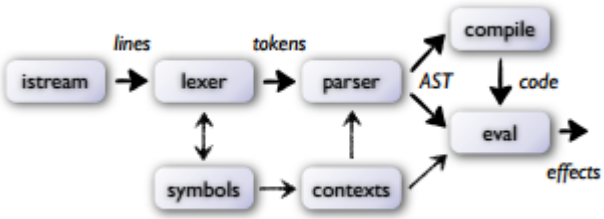
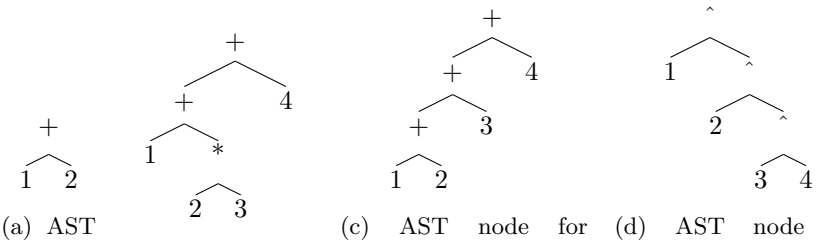


Figure 2.1: Structure of a lexer/parser based compiler. The lexer takes a stream of characters and translates them to a stream of tokens. While doing this, it will need to read from the applications symbol table which again might need to know something of the context. The stream of tokens will then be fed to the parser, which will read the tokens and apply high-level rules to them to create an AST. This AST can then be transformed/optimized by the transformer, or simply evaluated.

The most common way of writing a compiler or interpreter is to use a lexer → parser → transformer approach, as shown in figure 2.1. The lexer will provide a stream of lexical tokens to the parser which will translate the token stream to an Abstract Syntax Tree. This AST can then be optimized and transformed by a transformer, or it can be evaluated.

2.3.1 Abstract Syntax Tree



(a) AST node for 1+2

(b) AST node for 1+2*3+4

(c) AST node for 1+2+3+4, balanced for left-associativity.

(d) AST node for 1^2^3^4, balanced for right-associativity.

Figure 2.2: AST nodes for different expressions

An AST is a generalized way of expressing a program. As the name suggests, it has a tree structure with a set of allowed nodes. Examplewise, a node for the expression 1+2 can be expressed as figure 2.2a. It is important for an AST to be general, to be able to properly denote the source code, and simple, to be easy to analyse and transform.

An AST should properly describe the precedence of the different operators. The precedence means that the expression $1+2*3+4$ should be parsed as $1+(2*3)+4$, as the * has a higher precedence than +. The resulting AST node should therefore be the node in figure 2.2b, making sure that the * is in a lower node than the +.

Another aspect of expressions that need to be described by an AST is the associativity. Associativity means how an expression like $1+2+3+4$ is parsed. Is it parsed as the right-associative 1+(2+(3+4)) or the left-associative ((1+2)+3)+4? The AST needs to be properly balanced after whether the operator is right- or left-associative. + is a standard left-associative operator, which means that $1+2+3+4$ should evaluate to figure 2.2c, while the exponentiation-operator is typically right-associative, meaning that 1^2^3^4 should evaluate to figure 2.2d.

2.3.2 Lexers

Lexers are important for the simplicity of the parser and since it is pure string comparison, it can be really fast. A commonly used lexer generator is `flex`[13] made by Vern Paxson in 1987, but it is also common to write ones own lexer, to be in full control of the program. A typical token returned by the lexer is the type of the token (PLUS, MINUS, WHILE, LEQ, NAME, NUMBER) and an optional value if needed, as it will be in the case of e.g. NAME and NUMBER. One practical element of having a lexer is that one can remove unnecessary elements like spaces and comments, as well as taking care of the different versions of words (if/iF/If/IF) one can get in case-insensitive languages. Lexers also give the possibility of writing simpler parsers that therefore run faster. Lexers and parsers can, and usually will, be set up to run in parallel, with the parser parsing the stream of tokens that the lexer generates while it generates it.

2.3.3 Parsers

The parser takes the lexers stream of tokens and creates some sort of structured representation of it, e.g. an AST. The parser will also do syntactical analysis to root out syntax errors.

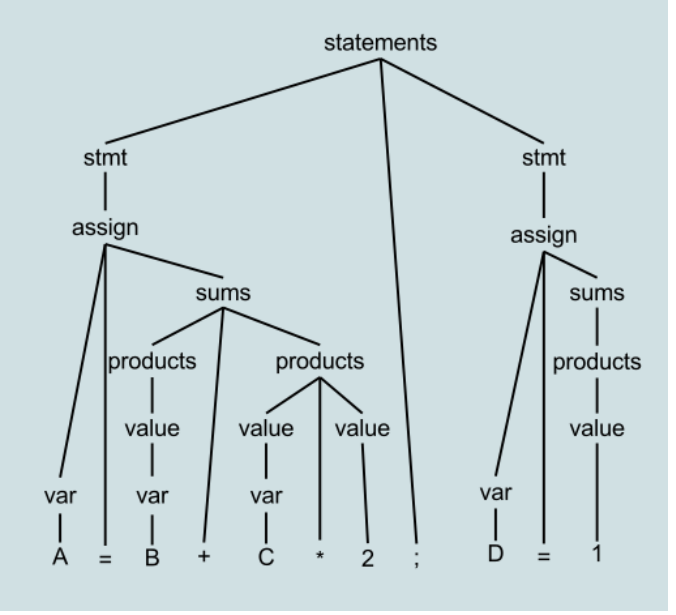


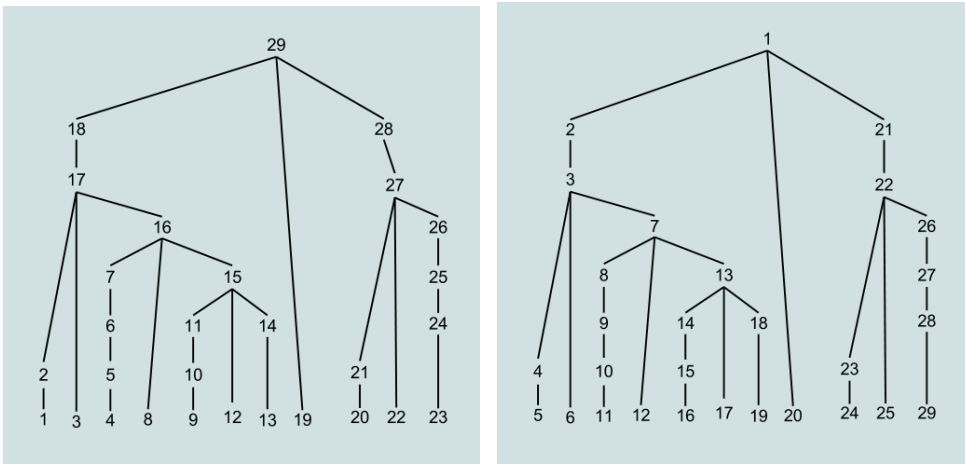
Figure 2.3: Typical parse tree for A = B + C*2; D = 1. [Source: Wikipedia]

There are two main ways of parsing, bottom-up or top-down. The difference is how they traverse the parse-tree[6]. An example of a a parse tree for the two statements `A = B + C*2; D = 1` can be seen in figure 2.3. A bottom-up parser will evaluate all the lower level constructs before evaluating any higher level rules, as seen in figure 2.4a. The top-down parser in figure 2.4b works completely opposite, evaluating the leaves once it knows that they are needed. [40] describes several different algorithms both for implementing top-down and bottom-up parsers, and a brief introduction of some of these will be given below.

Top-Down Parsing

The simplest version of a top-down parser is a recursive descent parser. To construct a recursive descent parser, one needs to think of each rule as a function where the name of the function is the non-terminal on the left hand side of the rule. Then, each instance of a non-terminal on the right hand side of the rules are a call to its corresponding function. Each instance of a terminal on the right

[6] A parse tree doesn't have to be created, it is merely a convenient way of visualizing the parse steps.



(a) Steps for Bottom-up parsing (b) Steps for Top-down parsing

Figure 2.4: Numbered parse trees for different approaches to parsing. Both trees parse the expression `A = B + C*2; D = 1` [Source: Wikipedia]

hand side of a rule is a call to match with that terminals input symbol. A recursive descent parser has no 'intelligence' behind how it chooses which rule to try to evaluate first. Given a rule:

```
X = 'a' | 'a',A | 'b',B ;
```

A recursive descent parser will choose the left-most rule first, evaluating as true at once when it has found a match. That will be a problem in this case, as the terminal 'a' will return true at once, never letting the rule 'a',A be checked, even if it might be the correct one. Therefore, one must always put the longest rule first, changing the grammar to:

```
X = 'a',A | 'a' | 'b',B ;
```

Using this rule, a recursive descent parser would check if the first letter is 'a' and if it is, it would continue to evaluate rule A, continuing until A is proven wrong. If A was wrong, it would consume 1 'a' from the stream and evaluate itself to true. This can lead to big spikes in performance, as a badly written grammar can make a parser backtrack a lot, loosing valuable time. A way of circumventing this problem is by adding lookahead. Lookahead means that the parser will look k steps ahead of the rule, to see if it should be evaluated or not. [40] defines `LL(k)`-parsers to be scanning the input from the `L`eft to the right, conducting a `L`eftmost derivation and having `k` steps of lookahead. It is commonly accepted that LL(1)-parsers represent the most acceptable compromise between expressive power and ease of implementation.

Bottom-Up Parsing

A common way of implementing a bottom-up parser is by using shift-reduction, which is normally implemented using a stack with the sentence to be parsed as the first item pushed to the stack. One will then push the current input symbol on the stack, until one has parsed the right hand side of a rule. The entire right hand side will then be popped and the left hand side of said rule will be pushed, reducing the stack. The parse succeeds once the stack contains one item which is the top level rule and fails if this has not happened once the entire sentence is parsed. This approach has problems as well though, as one might need to decide which rule to reduce to. This can also be solved by lookahead and [40] describes a few different classes of algorithms for lookahead, most notably the `LR(k)`-parsers. LR(k) stands for scanning from `L`eft to right, conducting a `R`ightmost or `R`everse derivation, and using `k` symbols of lookahead.

2.3.4 Left recursion

It is common that grammars shipped with languages to be left recursive, either immediate or indirect, which is a problem for most top-down parser implementations. A grammar is immediate left-recursive if it contains a rule as the one in listing 2.4.

Listing 2.4: Rule with immediate left recursion

```
A = A , a_1 | A , a_2 | ... | A , a_n | b_1 | b_2 | ... |
    b_m ;
```

Where `b_j` does not start with `A`. In this case, the parser may apply `A = A a_1` without ever getting to a terminal state. A grammar is indirectly left recursive if it contains a set of rules:

Listing 2.5: Set of rules with indirect left recursion

```
A_0 = A_1 , a_1 | ... ;
A_1 = A_2 , a_2 | ... ;
...\\
A_n = A_0 , a_{n+1} | ... ;
```

The parser could apply `A_0 = A_1 a_1 = ... = A_0 a_{n+1} ... a_2 a_1` and end up with the same problem as in immediate left recursion.

Several articles have been written about the problem of left-recursion in top-down parsers, both looking into how to write the parsers to allow left-recursion and about how to rewrite the grammar to remove it. [34] suggests a conservative extension to Parsing Expression Grammars, allowing them to parse left-recursive grammars. [40] shows a way of re-writing rules and sets of rules to remove left-recursion, while still being able to express the same language. This approach will be shown below.

Removing immediate left recursion

If given a rule for `A` which contains immediate left recursion as in listing 2.4, one can remove the left recursion by adding an extra rule `A'`, giving us the set of rules in listing 2.6.

Listing 2.6: Example result after removing immediate left recursion

```
A  = b_1  , A' | ... | b_m  , A' ;
A' = a_1  , A' | ... | a_n  , A' ;
```

This solution is without left recursion, as all `b_i`s are guaranteed not to start with `A` and `A'` will always need to consume an `a_i` before continuing with itself.

Removing indirect left recursion

Algorithm 1 Algorithm for removing indirect left recursion

Input: set of non-terminals X
  {Step 1}
  Order non-terminals from X in arbitrary order $A_1...A_n$
  for $i = 1$ **to** $i = n$ **do**
    {Step 2a}
    for $j = 1$ **to** $j = i - 1$ **do**
      if A_i contains choice on the form $A_j a$ where $A_j = b_1|...|b_m$ **then**
        Replace $A_j a$ with $b_1 a|...|b_m a$ in A_is production rules
      end if
    end for
    {Step 2b}
    Remove immediate left recursion in A_i
  end for
  return Set of non-terminals $A_1...A_n$ without left recursion

If given a set of rules `A_0-A_n` which contain indirect left recursion, as in listing 2.5, one can remove the left recursion by using algorithm 1. This algorithm guarantees that the rules are free from left recursion, but may get harder to read than the initial version. It is therefore suggested to try a few different ways of arbitrarily order the rules to find the best one.

2.3.5 Parsing Expression Grammars

Parsing Expression Grammar is an unambiguous way of describing formal languages, introduced by [15]. It was created as a response to the fact that most parsers were created using regular expressions, which are initially formalised, but loose the formal anchor when they are extended to allow for more expressive syntax. PEGs gives the same expressiveness as many of the extended regular expressions, but keeps a formal basis. [15] shows that a linear time parser can be built for any

PEG, using a technique called `packrat parsing` introduced in [14]. Packrat parsing is a parsing technique that memoizes the rules it has evaluated, making sure that a certain part of the input is only evaluated once, even if the parser backtracks past it several times. This introduces a linear memory cost to the parser, as the memory cost will be a constant times the size of the input instead of a constant times the nesting depth, which most recursive descent parser implementations will use.

LPEG

Another way of implementing a PEG is to use virtual parsing machine, as suggested by [33] and [18], and implemented in LPEG[22]. LPEG (Lua PEG) translates each rule into a program that can be built at run-time and be put together with other programs to create larger rules. It also has a simple implementation, hence it can easily be JIT-compiled.

One can write grammars directly in LPEGs syntax, which is described on its homepage. There are also several examples in the LPEG-recipes page[23]. If using regular expressions is too hard to let go off, LPEG is distributed with a regular expression-module, implemented in LPEG, where one can write grammars in a regex-like syntax and compile it to LPEG afterwards. This efficiently keeps a high readability without loosing any of the positive sides of LPEG. Its syntax is described on its homepage[22, @The re module] and only a short description will be given here, to allow the reader to understand the example code given in later chapters.

Table 2.2 shows the main syntactical elements of re. The most interesting parts are the captures, as the rest are quite self-explanatory ways of describing a grammar. The captures are what is used to do something with the grammar, e.g. to build ASTs. The simple capture will return the string it captures as its first return value and all embedded captures as further returns, while a table capture will return a table where all the embedded captures are stored in its fields. The $->$-operator will use its embedded captures as input to *name* which will normally be a table, where it returns the content of table[embedded captures], or a function where it returns the return values of the function with the embedded captures as arguments. $=>$ is a special case of $->$. Where $->$ does not evaluate before the entire input file have been parsed and been deemed to fulfil the grammar and all the lower rules' captures have been evaluated (as with all the other operators as well), $=>$ forces an immediate evaluation at once when it is reached. It will then immediately evaluate all its embedded rules and give the entire input file, the current position and all its embedded captures as arguments to the function. Depending on your return values, LPEG may then continue parsing from a position of the functions choosing, or stop because of some error.

(p)	grouping
'string'	literal string
"string"	literal string
[class]	character class
.	any character
%name	pattern defs[name] or a pre-defined pattern
name	non terminal
<name>	non terminal
{}	position capture
{ p }	simple capture
{: p :}	anonymous group capture
{:name: p :}	named group capture
{~ p ~}	substitution capture
{| p |}	table capture
=name	back reference
p ?	optional match
p *	zero or more repetitions
p +	one or more repetitions
p^num	exactly n repetitions
p^+num	at least n repetitions
p^-num	at most n repetitions
p − > 'string'	string capture
p − > "string"	string capture
p − > num	numbered capture
p − > name	function/query/string capture
p => name	match-time capture
& p	and predicate
! p	not predicate
p1 p2	concatenation
p1 / p2	ordered choice
(name < − p)+	grammar

Table 2.2: The syntax of LPEGs re-module

Chapter 3

Implementation

This chapter will show how the final application was implemented and which choices were made to solve the different problems that surfaced during the implementation. The chapter begins with an introduction into how similar projects have been structured and implemented, before a brief introduction to the structure of the application will be given. The implementation of the different parts related to parsing Lua will be introduced, including the AST, grammar and parsing, before it will be shown how the new unit testing framework `utest` is implemented. This chapter then goes on to show how the source code generator, interactive mode and correct error reporting have been handled, before it will be shown how easily one can extend the application by implementing interpreters for the new MAD and the old MAD-X.

When the author entered the MAD-project, some design choices had already been made. Most importantly that the new MAD-implementation should be written in Lua and run using LuaJIT, thereby restricting the program to following Lua 5.1 syntax, instead of the new 5.2[1].

3.1 Previous work

As the desired application has many different features[2], it is hard to find related works that embed all the different ones. In this section, a few projects that have one or more of the different features will be shown.

3.1.1 GSL Shell

GSL Shell[44] is an interactive command line interface that gives access to a collection of functions and algorithms from the GNU Scientific Library. It is based on LuaJIT, its syntax is an extension of Lua, and it runs in interactive mode. It also

[1] LuaJIT can be built to support most of Lua 5.2, but it will not support the entire 5.2.

[2] Using LuaJIT, having interactive and batch mode, easy to extend to support several different languages, packed in a single binary, runnable on several platforms, ...

comes with an executable that is used to install the files. This seems to be quite a good match for the desires of MAD, but when the new MAD project was initiated, GSL Shell was deemed not to be easily extendable. A module for language extension has been added later, and this module is based on luajit-lang-toolkit which is shown later.

3.1.2 Terra

Terra[42] is a new low-level system programming language that is designed to interoperate seamlessly with the Lua programming language. As explained by [9], Terra is implemented by doing changes in Luas C-library, making it load combined Lua/Terra-programs. In the edited library, they parse the Terra-files and build an AST for the specific Terra-functions. It then replaces the function call with a call to specialise the function in the current environment.

Terra is designed to be a low-level language to be used when one is really in need of speed and is meant to be used instead of making ones own Domain Specific Language. During implementation, it seems like the creators have meant for it to be used instead of having several different languages, meaning that the possibility of adding more supported languages have not been of the highest importance. Because of this and because the approach changes the Lua library, making it harder to update Lua and LuaJIT, this approach was not chosen to be investigated further.

3.1.3 Nyanga

Nyanga[37] is a language created by Richard Hundt as an answer to LuaJIT-users complaining about the language lacking this or that special feature. The language in itself is an object-oriented Lua-dialect, using LuaJIT to run it.

As figure 3.1 shows, Nyanga uses LPEGs re-module to write a grammar in an easy-to-read fashion, then compiles said grammar to LPEG, getting the safety of a PEG. It matches the Nyanga-file with the LPEG-grammar and creates a Nyanga-AST. This AST is sent to the transformer-module which transforms a Nyanga-AST to a Lua-AST. It then has a generator-module which can either transform the AST directly to bytecode to be run by LuaJIT, or to Lua-code to be dumped or run with LuaJIT.

Nyanga does not have an interactive mode, nor does it come in a single binary, but as these problems are something one can overcome and LPEGs re-grammars are easy to create, this approach was investigated further.

3.1.4 LuaJIT Language Toolkit

According to its announcement post ([28]) on LuaJITs mailing list, the LuaJIT Language Toolkit[29] is a Lua implementation of Lua, targeting LuaJIT by emitting optimized bytecode. Its purpose is almost exactly the same as MADs, creating a framework to easily extend Lua. The toolkit was not around when the author initiated this project, as it was first announced in february 2014, and was therefore

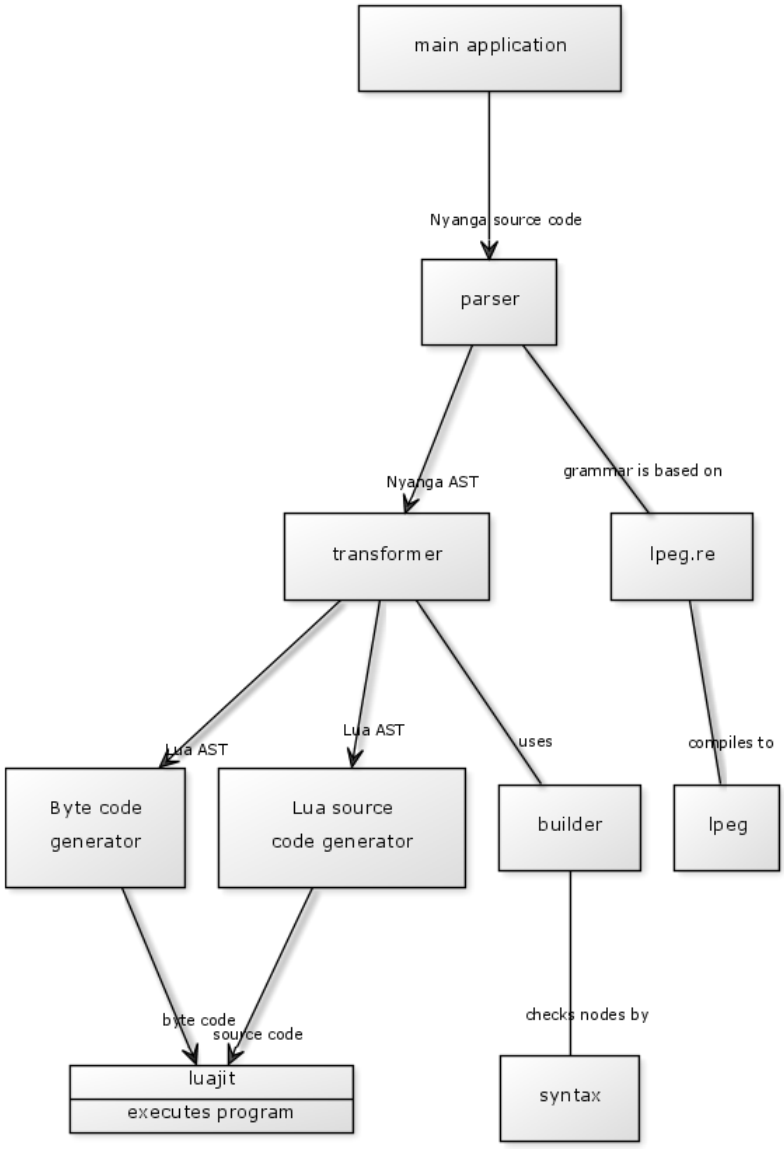


Figure 3.1: Nyangas interpreter chain. The parser takes Nyanga code and translates it to a Nyanga AST. The transformer then transforms this Nyanga AST to a Lua AST, which is either sent to a byte code generator or a source code generator, both of which are run using LuaJIT.

not taken into consideration when deciding how to structure the code. It will still be explained briefly how it is implemented.

The toolkit started with Nyanga as a base, but its author did not like LPEG and went for a lexer/parser approach instead. The lexer and parser are translated versions of LuaJITs lexer and parser, while the bytecode generator is based on Nyangas generator. To extend Lua using this toolkit, one will only need to change the lexer and parser, which should be simpler than digging into LuaJITs C-implementation.

3.2 Main structure

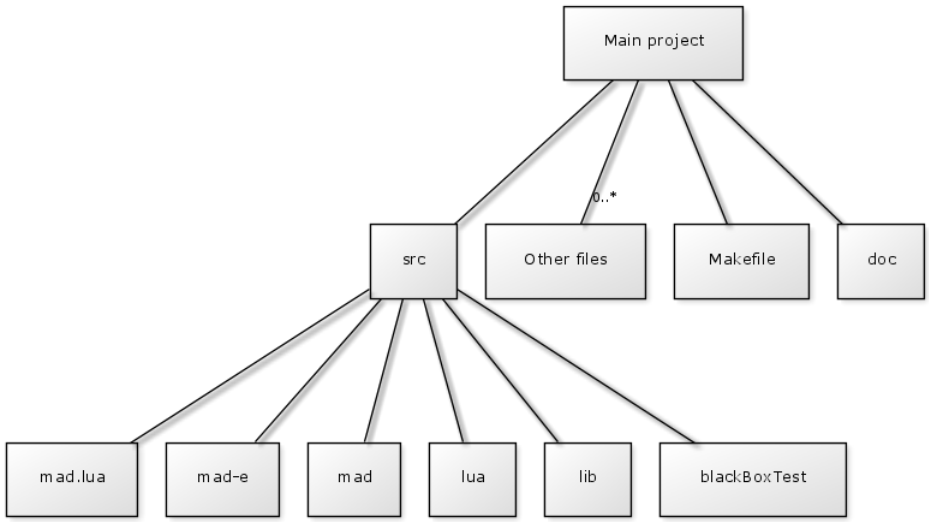


Figure 3.2: The main structure of the new MAD. doc contains documentation. src contains the source files. mad.lua is an umbrella file to be used by the users and contains the main parts of the application. lua will contain extensions to the Lua-language, while mad contains all aspects of MAD, including physics and compiler. lib contains all external libraries, as LPEG and potential others. blackBoxTest contains full files in different languages used for simple testing of the entire project. mad-e is a script used to interpret and run .mad/.lua/.madx-files.

Before starting a project of this size, a skeleton is needed, to make sure that it is properly structured and that it is easy to figure out where to add new modules.

The main structure is as shown in figure 3.2 and will be explained briefly here:

All the source code of the application will be in the `src`-folder.

`doc` contains all major documentation for the program. There will also be documentation within the files..

The other files in the top-level will be the Makefile to install the program[3] and

[3]Until one can ship it as a single binary, if possible.

other files meant for clarifications in between developers.

`mad-e` is the script that will run the application. Its structure will be:

```
init: Set application specific variables, start processes,
    etc.
options: Handle command line arguments given to the
    application.
        Ex: Start unit tests if '-utest' flag is given.
exec: Start interpreting files given, or go to interactive
    mode.
fini: Stop processes started by the application, etc.
```

This script will automatically use luajit to run, ensuring speed.

`mad.lua` will be the main file to be used in all MAD-files. By having `local mad = require'mad'` in the beginning of ones file, one will get access to all the MAD-specific modules one might need (e.g. `mad.sequence`).

The `mad`-folder will be further described below.

The `lua`-folder will contain eventual modules needed to extend the Lua-language, not necessarily having to do with the MAD-language. Examples of this is `lua.tableUtil` which contains functions for recursively printing a table[4] or `lua.process` which contains functions for starting a process to be run and communicated with in parallel with the program[5].

The `lib`-folder is where the applications external libraries will be added. To avoid filling the namespace, it was decided that lib should not be added to requires search paths[6]. To overcome this, all external libraries will have a wrapper-file to fix everything search path-related without the user needing to think about it. Then the user can require the wrapper function instead. The set up for a standardized wrapper will be:

```
--Filename: lib/xxx.lua
local pcp = package.cpath
package.cpath = ";;./lib/path/to/xxx.so;.\\lib\\path\\to\\
    xxx.dll;"
-- Change comments if xxx is a lua-file
--[[local pcp = package.path
package.path = ";;./lib/path/to/xxx.lua;.\\lib\\path\\to\\
    xxx.lua;"]]
local xxx = require"xxx"
package.cpath = pcp
return xxx
```

This will ensure that the user will not be able to require other modules from the libraries without explicitly trying to.

[4] E.g. for printing an AST.

[5] E.g. gnuplot.

[6] package.cpath and package.path

3.2.1 mad

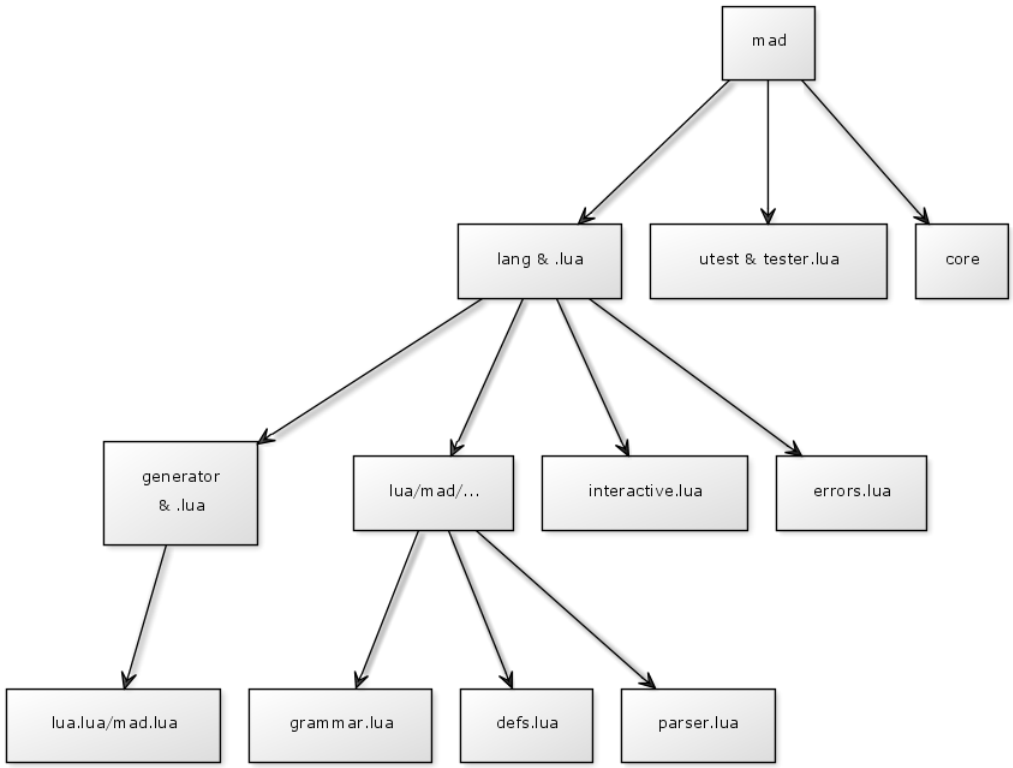


Figure 3.3: The structure of the mad subfolder. Core contains modules related to the running the application. Utest contains the unit testing framework. Lang contains modules that have to do with the different languages the application supports.

Figure 3.3 shows the main structure of the `mad` subfolder. The `mad`-folder will contain all modules having to do with the mad-application that are not an external library or an extension to the Lua-language. This folder is where most parts of the application will be and is split into several sub-folders. The important ones for this report are the ones having to do with the interpreter and the application as a whole, while the ones having to do with physics etc. will not be described further. The three main sub-folders are `lang`, `utest` and `core`.

The `utest` folder and the corresponding `tester.lua` module contain the framework for unit testing and will be introduced in a later section.

`core` contains the modules that are used strictly for running the application. `exec.lua`, `init.lua`, `fini.lua`, and `options.lua` are all in this folder and are all connected to the script running the application.

The `lang` folder contains all files having to do with the different languages

supported by the application. `lang.lua` is a parser constructor factory, where the user can get the parsers for the different languages. `interactive.lua` and `errors.lua` are modules having to do with the interactive mode and the error reporting, both of which will be explained later. `generator.lua` is a generator factory, where the user can get different generators to get different kinds of source code. The two supported generators so far are for Lua and MAD code, but modules for e.g. bytecode can be added later.

All parsers will have the same structure, with a file containing the grammar, one containing the rules for building the AST and one stringing them together into a proper parser. When lang.lua looks for a parser for e.g. the xxx-language, it will look for the file ./lang/xxx/parser.lua. How the files defs.lua, grammar.lua and parser.lua work will be described in later sections.

3.3 Abstract Syntax Tree

A proper Abstract Syntax Tree needs to contain enough nodes to properly define a program, but few enough for functions parsing the AST to not be overly complex. The ultimate AST would be general enough to be able to properly define all languages, but in practise, one will specialize it towards a special language, in this case Lua.

It was quickly decided that Nyangas Lua-AST was too complex and that a sparser one would need to be created. One example is that Nyanga uses different nodes for a global and a local assignment. As they are both doing the same thing, they should be the same node. The only difference is that one can have different 'kinds' of assignment, in Luas case local and global. Therefore an assignment-node is made, which has an optional kind='local'.

When implementing the literal node, there were some different solutions to consider about how the value of the literal should be stored. Should they be stored as in Nyanga, where the value is stored directly in the node, or should the value be stored as a string. Both sides have positive and negative sides. Storing the value will make it easier to read the type from the AST when that might be needed (e.g. for transforming), but storing the full string will greatly simplify the parsing and the generator steps. To be able to store the value as itself, one will also need to keep track of which string delimiter was used on said string, as it will need to be generated with the same delimiter or problems might arise. Instead of having an optional key in the AST-node to store the delimiter and because one will only need to look at the first character of the value to tell which type it is[7], it was decided to store all values as strings.

All the accepted nodes of the AST can be found in Appendix D: Lua AST.

[7] "'[=> string, number => number, n = nil, t/f = boolean

3.4 Lua grammar

The entire Lua EBNF-grammar can be seen in Appendix E: Lua Grammar. This section will be about how to make this grammar work in a PEG and in LPEGs re-module while keeping it readable.

3.4.1 Removing left recursion

This is a correct EBNF-grammar, but it is not possible to use it in a PEG without doing some changes because of it containing left-recursion. There have been several attempts at removing this problem, examplewise by [10], changing the grammar to the grammar shown in listing 3.1.

Listing 3.1: Donnelys Lua EBNF grammar without left recursion

```
infix_exp     = unar_exp { binop unar_exp }
unar_exp      = unop unar_exp | postfix_exp
postfix_exp   = 'nil' | literal | '...' | funcexpr |
    prefix_exp | tablector
prefix_exp    = varOrExp { nameThenArgs }
funccall      = varOrExp nameThenArgs { nameThenArgs }
varOrExp      = var | '(' exp ')'
nameThenArgs = [ ':' name ] args
var           = ( name | '(' exp ')' varsuffix ) { varsuffix
    }
varsuffix     = { nameThenArgs } ( '[' exp ']' | '.' name )
```

Others have solved it by changing the PEG, like LpegLJ[24], a LuaJIT-implementation of LPEG, which implements the suggestions made by [34]. By using LpegLJ, one could use the same grammar as provided by Lua, without having make any major changes.

It was decided to stay with LPEG, as LpegLJ is still in its infancy and that it is still slower than LPEG run with LuaJIT. If it gets stable releases that are faster than LPEG at some point in the future, it will still be easy to change to it. Because of this decision, the grammar will need to be rewritten. Donnelys suggestion might work, but to be in control of the changes, the rewriting will be done by the author.

The grammar contains left recursion in two distinct sets of rules, the ones concerning `exp` and the ones concerning `var/funcall/prefixexp`.

Removing immediate left recursion from exp

Listing 3.2: Lua expression definition with left recursion

```
exp = nil | false | true | Number | String | '...' |
    functiondef | prefixexp | tableconstructor | exp, binop,
     exp | unop, exp ;
```

The rule in listing 3.2 has immediate left recursion in its `binop`-choice, as it can choose `exp` an infinite amount of times without consuming any input. By using the

algorithm presented in listing 2.6 and making the result more readable, one gets the rules in listing 3.3

Listing 3.3: Lua expression definition without left recursion

```
exp     = unexp , { binop , unexp } ;
unexp   = unop , unexp | valexp ;
valexp = nil | false | true | number | string | '...' |
    fundef_a | prefixexp | tablector ;
```

Removing indirect left recursion from var/funcall/prefixexp

Listing 3.4: Lua variable and funcall-definition with left recursion

```
funcall    = prefixexp , args | prefixexp , ':' , name ,
    args ;
var        = name | prefixexp , '[' , exp , ']' | prefixexp
    , '.' , name ;
prefixexp = var | funcall | '(' , exp , ')' ;
```

Listing 3.4 contains indirect left recursion, hence it can not be used by LPEG and needs to be rewritten. Using algorithm 1, one can rename the rules to the following, after applying step 1 and 2a.

```
A1 = A3 a1 | A3 a2 a3 a1     = A3 b1 ;
A2 = a3 | A3 a4 | A3 a5 a3   = a3 | A3 b2 ;
A3 = A2 | A1 | a6            = a3 | A3 b2 | A3 b1 | a6 ;
```

We then need to remove the immediate left-recursion from A3, using the rules in listing 2.6, which gives:

```
A3  = a3 A3' | a6 A3' ;
A3' = b2 A3' | b1 A3' | epsilon ;
```

epsilon is the empty statement. Returning to the initial nomenclature, inserting fitting names for the `A3'`, `b1` and `b2`, and splitting rules for simplicity the end result becomes:

Listing 3.5: Variable and function call definitions without left recursion, but not working on PEG

```
funcall      = prefixexp , call ;
var          = prefixexp , index | name ;
prefixexp    = name , prefixexp_r | '(' , exp , ')' ,
    prefixexp_r ;
prefixexp_r = [ suffixexp , prefixexp_r ] ;
suffixexp    = index | call ;
call         = args | ': ,' name , args ;
index        = '[' , exp, ']' , | '.' , name ;
```

`name` has been moved to the end of `var` as LPEG evaluates the first rule first, not the longest. Listing 3.5 is a correct rule set, but it will still not work to create

a parser. The reason for this is that `suffixexp` will consume every single `call`, including the last one that `funcall` needs, thus making all function calls evaluate as variables. A solution to this is to add lookahead in `suffixexp`, efficiently only allowing it to consume any input if it is immediately followed by another. This solution will introduce a lot of backtracking though, as it will need to parse every consumed `index|call` twice.

Because of the previous solution being inadequate, a new attempt was made with a new structure, which lead to the set of rules in listing 3.6

Listing 3.6: Variable and function call definition without left recursion, but too much backtracking

```
prefixexp   = funcall | var | grpexp ;
grpexp      = '(' , exp , ')' ;
var         = varprefix , {varsuffix} ;
varprefix   = name | grpexp , varsuffix ;
varsuffix   = {call} , index ;
index       = '[' , exp , ']' | '.' , name ;
funcall     = callprefix , call , {call} ;
callprefix  = var | grpexp ;
call        = [ ':' , name ] , args ;
```

Listing 3.6 was created by first making a `var` that worked following the EBNF, then making a `funcall` using `var` that fulfilled the EBFN. Doing it this way made the final writing of `prefixexp` a simple affair. This solution manages to properly differentiate between a variable and a function call, and can then be used to create a proper AST. It does not, however, manage to do so without massive worst-case backtracking. `prefixexp`, which is the rule used in `exp`, needs to parse an entire `var` and decide that it does not end in a `call` before failing on its evaluation of `funcall` and being able to actually parse the `var`. This leads to unacceptably slow parsing, as every variable needs to be parsed twice, and this approach can therefore not be used. An example of this slow parsing can be shown in the parsing of `grpexp` in figure 3.4. The figure shows an attempt at parsing 4000 lines of an expression on the form $(((1) + 1) + 1)$ with x parentheses. As one can see from the graph, doubling the depth of the parsed expression has the same effect as doubling the length of the file.

The main problem with both rule sets is that one either can not differentiate between a variable and a function call, or one will have to allow single variables to be used as statements, thus allowing illegal Lua-syntax. If both of these problems shall be disregarded, the parsing will be unacceptably slow both in worst- and medium-case situations. By splitting the variable and function call-rules into several rules, depending on where they are to be used, one can differentiate in the cases where it is important (not allowing variable for statement) and not have to use use lookahead in the rule where both variables and function calls will be used the most (in expressions).

Listing 3.7: Variable and function call definition split into definition and expression-parts

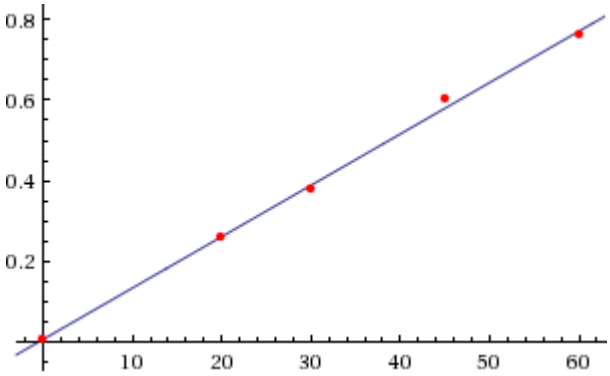


Figure 3.4: Timing of the expression $(((1)+1)+1)$ with x parentheses. y-axis is the time in seconds needed to parse 4000 lines of the expression. $x = 100$ failed with 'Too many pending calls/choices'.

```
varexp  = (name    | grpexp) , { tableidx | funcall };
vardef  = (name | grpexp , varsfx) , {varsfx};
varsfx  = {funcall} , tableidx;
funstmt = (name | grpexp) , {varsfx} , funcall , {funcall};
funcall = [ ':' , name ] funargs;
```

In listing 3.7, `varexp` is used in expressions, while `funstmt` is used as a stand alone statement and `vardef` is used in assignments. This solution removes the need for massive backtracking, thus speeding up the parsing process.

3.4.2 Operator precedence and associativity

or
and
< > >= <= = ==
..
+ -
* / %
not - #
^

Table 3.1: Luas operator precedence, higher priority the further down in the table the operator is located. .. and ^ are right associative, the rest are left associative.

The Lua grammar in the appendix does not contain the precedence nor the associativity of the operators, but to properly build the AST, this needs to be implemented. Nyanga[37, /blob/master/src/lang/tree.lua] does this in the AST-building phase, balancing the tree after getting a binary expression from the grammar, using a rule set very similar to listing 3.3. Having this in the grammar instead of hidden away in the AST-definitions will simplify reading and also speed up the

building phase, as LPEG can parse some extra rules faster than a function can go through all nodes of a table[8]. A first attempt at creating a rule set containing both the associativity and the precedence, following the precedence and associativity shown in table 3.1 and building out from listing 3.3 yielded listing 3.8.

Listing 3.8: Lua expressions with precedence and associativity, but with left-recursion

```
exp         = orexp ;
orexp       = orexp     or      andexp     | andexp ;
andexp      = andexp    and     boolexp    | boolexp ;
boolexp     = boolexp   boolop  concatexp  | concatexp ;
concatexp   = sumexp    '..'    concatexp  | sumexp ;
sumexp      = sumexp    sumop   prodexp    | prodexp ;
prodexp     = prodexp   prodop  unexp      | unexp ;
unexp       =           unop    unexp      | powexp ;
powexp      = valexp    '^'     powexp     | valexp ;
```

The problem with listing 3.8 is that when adding support for associativity, left recursion was inserted as well. Following the rules for removing immediate left recursion in listing 2.6, one reaches listing 3.9.

Listing 3.9: Orexp with left recursion and associativity removed

```
orexp   = andexp orexp_r ;
orexp_r = or andexp orexp_r ;
```

This set of rules remove the left recursion, but it also removes the associativity.

It was therefore decided to create an AST that does not make any assumptions about the associativity and allowing the Lua-code that runs the code to use its associativity if nothing else is used. I.e., if one wishes to change the associativity of an operator, one will have to do it in the post processing as in Nyanga. The expression-node shown in Appendix D: Lua AST contains a list of operators and expressions, where all operators of the same precedence will come in the same node, instead of using the more standard lhs/rhs-setup. When not needing to implement the associativity in the grammar, the result is listing 3.10.

Listing 3.10: Lua expressions with precedence, but without associativity

```
exp         = orexp ;
orexp       = andexp    { or      andexp    } ;
andexp      = boolexp   { and     boolexp   } ;
boolexp     = concatexp { boolop  concatexp } ;
concatexp   = sumexp    {  '..'   sumexp    } ;
sumexp      = prodexp   { sumop   prodexp   } ;
prodexp     = unexp     { prodop  unexp     } ;
unexp       =           { unop }  powexp ;
powexp      = valexp    { '^'     valexp    } ;
```

[8]Especially because one will have to look at some nodes' hash table parts, not only array parts. LuaJIT is slower at going through the hash table compared to the array.

3.4.3 Lexical elements

So far in this section only grammars written in EBNF have been shown, but to be able to use this in a LPEG-parser one will need to write either LPEG functions or a grammar in LPEGs re-syntax. To have an easily readable grammar and because the only speed loss one will get is in the initial compilation from re to LPEG, it was decided to use re. As this project does not follow the lexer/parser set up, one will need to implement the lexer in the grammar. Most of the implementation is straight forward, but a few problems surfaced during the writing and both the difficulties and the solutions will be explained below.

A lexer will normally have some sort of rule telling it to disregard white space and comments, but in LPEG one will have to add this within the grammar. Lua is very lenient when it comes to white space, as there may or may not be white space between every single atomic unit, and since this parser shall parse Lua-code it will need to be as lenient. To accommodate this possibility of white space everywhere, the grammar will quickly get cluttered down by grammatically unimportant objects and it will be very easy to introduce subtle bugs by not allowing white space in some random positions. This lead to the creation of `s`, whose implementation is shown in listing 3.11.

Listing 3.11: Implementation of the white space-rule in the Lua grammar. ws is a rule containing all white space that's not a newline, nl is a newline and cmt is all comments.

```
s       <- (ws / nl / cmt)*
```

By always putting an `s` in front of all string literals and in front of the lexical rules `number`, `string`, `name`, etc., as well all keywords, it is guaranteed that the grammar is fully covered in terms of white space support. Since `s` will only appear in front of string literals in the parser part of the grammar, the grammar will not be unnecessarily cluttered and keep its readability.

3.4.4 Adding actions

To use the grammar for something else than testing if the syntax is correct, one will need to add actions and capture values. This will easily clutter up the grammar and make the parser slower, one will therefore need to find a way of capturing the values that is readable and that does not slow the parser down.

Named captures or functions

There are two main different solutions to how one can create the AST, either by capturing the values and sending them to functions defined somewhere else, or by using named captures.

Listing 3.12: Two different ways of capturing literal nodes

```
literal <- s{:ast_id: ''->'literal':} {:value: nil / false
    / true / number / string / ellipsis :}
```

```
literal <- s{nil / false / true / number / string /
    ellipsis} -> literal
```

```
function literal (val)
  return { ast_id = "literal", value = val }
end
```

The resulting ast node of the two approaches in listing 3.12 will be the same, but the second approach does not clutter up the grammar as much as the first one. Speed is of the essence, especially when it comes to literals and names, as they are the two types of nodes that will be used most of the time. Tests with parsers created with both approaches were made and resulted in both approaches using the same amount of time, which means that the second approach is to be used. This means that one will have an extra file containing definitions of functions to create the AST-nodes.

Expression

As the action-based approach was chosen for literals, this was the approach that continued to be used for consistency. The first, table-capture based approach can be seen below, with a lot of functions omitted because of similarity.

Listing 3.13: Adding actions to exp using table captures

```
exp    <- {orexp }                              -> exp
orexp  <- {andexp} {|( or  andexp  )*|} -> orexp
andexp <- {logexp} {|( and logexp  )*|} -> andexp
```

```
function orexp ( _, val, tbl )
  if #tbl == 0 then return val end
  table.insert( tbl, 1, val )
  return { ast_id = "expr", tbl }
end
```

The approach in listing 3.13 compiled a test file containing 100k lines of random Lua code in 1.10s, almost 3 times slower than without actions. Another approach will therefore have to be chosen. The reason for the speed-loss is partly because of the fact that the table capture creates a table even if the table will be empty. The AST node created with this approach would in fact not be the same node that is described in Appendix D: Lua AST, but since table.unpack is not JIT-compiled it would be even slower if this was implemented; Should the approach be used, the allowed AST nodes would need to be changed as well. The approach in listing 3.14 was an attempt to remove the table creation.

Listing 3.14: Adding actions to exp using simple captures

```
exp    <- { orexp }                      -> exp
orexp  <- { andexp ( or  andexp  )* } -> orexp
andexp <- { logexp ( and logexp  )* } -> andexp
```

```
function orexp ( _, first, ... )
  if ... == nil then return first end
  return { ast_id = "expr", first, ... }
end
```

The simple captures returns the captured string and then all other captures, if any. A parser built using the approach in listing 3.14 ran the test file in 0.85s; a good speed boost compared to listing 3.13, but still twice as slow as the action-less grammar. Profiling using LuaJIT 2.1s profiler showed that this was mostly because of callbacks from LPEGs side, which means that there is not much to do about it and this approach was kept.

tabledef

The main problem in capturing a table definition is to catch the difference between `[a] = 1` and `a = 1`. Two ways of doing it were tried and the fastest was chosen. The attempt in listing 3.15 captures the [.

Listing 3.15: Capturing table fields by capturing [

```
tabledef  <- (s'{' { fieldlist? } s'}') -> tabledef
fieldlist <- field (fieldsep field)* fieldsep?
field     <- {s{'['} exp s']' s'=' exp / name s'=' exp /
   exp} -> field
```

```
function field ( _, op, key, val )
  local kind = "expr"
  if not key then
        kind = nil
    val = op
    op = nil
  end
  if not val then
    kind = "name"
    val = key
    key = op
    op = nil
  end
  return { ast_id = "tblfld", kind = kind, key = key, value
      = val }
end
```

A parser built with listing 3.15 parsed a file of 100k lines of `a = 1, [a] = 1` in 0.84s.

The approach in listing 3.16 is to look through the captured string if the [is there.

Listing 3.16: Capturing table fields, without capturing [

```
tabledef  <- (s'{' { fieldlist? } s'}') -> tabledef
fieldlist <- field (fieldsep field)* fieldsep?
field     <- {s'[' exp s']' s'=' exp / name s'=' exp / exp}
    -> field
```

```
function field ( str, key, val )
  local kind = "expr" and string.find(str, "[") or "name"
  if not val then
    val = key
    key = nil
    kind = nil
  end
  return { ast_id = "tblfld", kind = kind, key = key, value
      = val }
end
```

A parser built with listing 3.16 compiled the 100k lines test file in an average of 0.88s, which means that listing 3.15 is used.

3.5 Unit testing

In any project of some size, one will need some way of testing the code. One way is to use black box testing, where one will run the entire program and see if it works or fails, e.g. by letting the MAD application parse a Lua file and see if it acts correctly. A problem with black box testing is that it does not give any specific information as to what part of the code failed. It is also very difficult to find code that properly tests all parts of a program. A way of fixing this problem is to use unit testing instead. Unit testing is a method where one tests the individual units of code instead of testing the entire system. The units can be of different sizes, but in Lua the most logical units to use are modules and functions. Unit tests only demand that functions have a specified API, as the way they work is along the lines of *"If this function is called with these parameters, the output should be like this."*. As long as the API is decided one may write the unit test, even if the functions they are going to test are not made yet. This way of developing code is often called test driven development.

The standard set up for a set of unit tests is to have one setup and teardown-function for each module, as well as a number of test functions. The setup function will be called before each test function and the teardown function will be called after each test function. Setups responsibility is to set up all dependencies that many functions will share and teardowns responsibility is to remove everything setup made. This is to make sure that all test functions are independent, which is an important part of unit testing.

For smaller projects, one may simply write a test for each function and call the test functions when one wishes to test the code, but for any project large enough

to do anything interesting, one will greatly benefit from having a unit testing framework.

3.5.1 Previous work

There are several unit testing frameworks written for several different platforms and languages, among them JUnit[20] and TestNG[43] for Java and Check[5] for C. There are also frameworks that have been ported to several other languages, like Haskells QuickTest, introduced in [6], which have been ported to C, C++, Java and Python, among many. Another large family of unit test frameworks is xUnit, which contains Java[9], C++, .NET and Matlab-ports. Unfortunately, there are not many different frameworks written for Lua, but the three the author managed to find will be introduced here.

Lunit

Lunit[31] was initially written in 2004 by Michael Roth and released under the MIT licence. In 2008 it got rewritten to support Lua 5.1 and after 2009 there have been no further changes. Lunit introduces 26 different assert-functions, as shown in table 3.2, to be used in the test functions. It also has 8 functions to test the types of values. To write a test function using lunit, one will need to create a function and name it something that contains 'test'[10] either in the beginning or the end of the name and then use the assert_xx or is_xx-functions to test the values that one wishes to test. To run all tests in a file, one will then need to run the lunit shell script with the files name as an argument.

This framework has several problems, the most important being that nothing has happened in its GIT-repository[32] for over 4 years, which is a clear pointer towards there not being any support from the package maintainer should any problems arise. It also uses an old dialect of Lua 5.1, i.e. being a lot closer to 5.0 than to 5.2. An example of this is the module-function which is an important part of this approach and that is deprecated in Lua 5.2. Another problem is the unnecessarily large number of assert- and is-functions. Every single is-function could simply be replaced with a call to assert_equals with type(actual) and nameoftype as its arguments. Most assert-functions could also be replaced with a few less specialised versions.

All in all, to use this framework, one would have to give it a complete rewrite, both to remove its old syntax and to generalise its assert-functions. In the end there would not be much left of the framework and one could therefore just as well write ones own.

Luaunit

Luaunit is based on the initial work by Ryu Gwang[30] and is based on the python unit testing framework[45]. The current version is developed by Philippe Fremy. It

[9] JUnit
[10] Case insensitive

fail([msg])	Always fails.
assert(assertion,[msg])	Fails, if 'assertion' is false or nil.
assert_true(actual,[msg])	Fails, if 'actual' isn't true.
assert_false(actual,[msg])	Fails, if 'actual' isn't false.
assert_equal(expected,actual,[msg])	Fails, if 'actual' is different from 'expected'.
assert_not_equal(unexp,actual,[msg])	Fails, if 'actual' and 'unexp' are equal.
assert_match(patt,actual,[msg])	Fails, if the string 'actual' doesn't match 'patt'.
assert_not_match(patt,actual,[msg])	Fails, if the string 'actual' match 'patt'.
assert_nil(actual,[msg])	Fails, if 'actual' isn't a nil value.
assert_not_nil(actual,[msg])	Fails, if 'actual' is a nil value.
assert_boolean(actual,[msg])	Fails, if 'actual' isn't true or false.
assert_not_boolean(actual,[msg])	Fails, if 'actual' is true or false.
assert_number(actual,[msg])	Fails, if 'actual' isn't a number.
assert_not_number(actual,[msg])	Fails, if 'actual' is a number.
assert_string(actual,[msg])	Fails, if 'actual' isn't a string.
assert_not_string(actual,[msg])	Fails, if 'actual' is a string.
assert_table(actual,[msg])	Fails, if 'actual' isn't a table.
assert_not_table(actual,[msg])	Fails, if 'actual' is a table.
assert_function(actual,[msg])	Fails, if 'actual' isn't a function.
assert_not_function(actual,[msg])	Fails, if 'actual' is a function.
assert_thread(actual,[msg])	Fails, if 'actual' isn't a thread.
assert_not_thread(actual,[msg])	Fails, if 'actual' is a thread.
assert_userdata(actual,[msg])	Fails, if 'actual' isn't userdata.
assert_not_userdata(actual,[msg])	Fails, if 'actual' is userdata.
assert_error([msg],func)	Fails, if 'func' doesn't raises an error.
assert_pass([msg],func)	Fails, if 'func' raises an error.

Table 3.2: Lunits assert-functions

is written in 2004 and not updated post Lua 5.0. It has fewer assert-functions than lunit, only assertEquals, assertDiffers, and the standard assert. It has a similar way of deciding what is a test, as that is defined to be any function that starts with 'Test'. It is run by running the function use_luaunit.lua with a module name given in the command line, or using the default value of all modules starting with 'Test'. The program will then run every function that begins with 'Test' in the tested modules and report the results.

Luaunit is a small framework, its major problem being that it is a 'dead' project, as it is not even updated to the second newest Lua-version, but as it is really small it will not be a problem to rewrite it or to use it as a base for ones own framework. The code is quite chaotic, which means that a clean up will be in order as well.

Build Test Deploy

BTD[3] is an attempt to create an environment for building, testing and deploying Lua, but as of now[11], it only contains an extension of Luaunit. The extension is mostly geared towards utilising Lua Lanes[26] to run several tests in parallel. In the process of extending Luaunit, it was also updated to Lua 5.1. The framework was last updated in january 2013 and is considered final by the author.

BTDs unit testing part consists of 4 files, where only two of them are important to the user. `Test.lua` is the main module, which runs the tests, keeps track of the number of failures and prints the information once the tests are done. It has the option of running all tests for a test class, or only running a single test in said class. `TestApi.lua` contains four assert-functions (equals, differs, fails, and succeeds) for use in test functions and the function wrap, which wraps a set of functions into a test class. The internal modules `TestClass` and `TestMethod` contain constructors for creating test classes/methods and keeps track of the statistics of classes and methods. They also contain functions for printing the results.

In theory, BTD is a good framework to use for MADs unit testing, but it will need heavy modifications both to print the results in a pretty manner (the current way is unnecessarily verbose) and to decide which tests to run, as BTD keeps Luaunits way with forcing test function to start with 'Test'. The possibility of running tests in parallel should give a nice speed boost, but when the author tried to BTD in parallel, he got several errors within lanes which could be fixed by using a newer version of lanes, but BTD is incompatible with the newer versions. Because of this, it was decided to cut the need for lua lanes and lua file system, to lessen the amount of external libraries to rely on.

3.5.2 Implementation

As BTD needed to be thoroughly rewritten, it was decided to write a new framework inspired by BTD, instead of simply changing things within the framework. Some parts were deemed almost complete and were therefore kept almost in the current form, while some where totally changed.

The first thing to be decided was the interface for the modules to use in the tests. As it was already decided that all MAD-modules should be tables, it was a simple addition to say that all modules should have a help- and a test-table as keys. The module.test contains all the test functions as well as the setup and teardown-classes. Module.test can also contain a module.test.self, which tells the unit testing framework (utest) which modules said module relies on to ensure that they are tested as well. As unit tests tend to take up at least as much space as the code they are testing, it is also necessary to be able to have them in separate files. This should be kept hidden from the end user, as module.test should act the same way if it is a part of the file or not. It was therefore decided that if the tests are kept out of the file, they shall be in a folder named test within the same directory as the file and have the same name as the file. I.e., if a function does not find the

[11] And probably forever

tests in mad.module.test, it can look in mad.test.module. The test functions shall be written on the form in listing 3.17.

Listing 3.17: Example of how to write a test function

```
function mod.test:plus(ut)
  local result = ut:succeeds(plus, 1, 2)
  ut:equals(result, 3)
  ut:differs(result, 2)
  ut:fails(plus, 1, nil)
end
```

`succeeds`, `fails`, `equals`, and `differs` are functions that shall be given by the utest framework.

mad.tester

All this shall be kept in the module `mad.tester` which is the module acting as the interface between the code wanting to run tests, the code being tested and the utest framework. Mad.tester contains three public functions: `mad.tester.addModuleToTest`, `mad.tester.run`, and `mad.tester`.

`tester.addModuleToTest(modname)` adds the module by the name modname to the list of modules that need to be tested, if it is not there already. This function is the one that shall be used in modname.test.self.

`tester.run(mod[,fun])` runs all tests for a single module or the test function fun if it is given. Both mod and fun may be either a string, where it will be interpreted as the module/function-name, or a table/function, where it is assumed to be the module or function to be tested.

`tester([listofmodnames])` takes a list of module names and runs the unit tests for them. If it does not get any input, it will use `mad.module` to find all currently loaded modules and run the tests for them.

mad.utest.testObject

`mad.utest.testObject` is similar to BTDs `TestApi`, in that it contains the functions used to assert in the test functions. The same functions have been kept, but with modified names and with the addition of some more statistics, as this module keeps track of how many assertions where made within a test function.

`testObject:fails(function, [args]...)` runs function with args and raises an error if function succeeded.

`testObject:succeeds(function, [args]...)` runs function with args and raises an error if function raised an error. It returns the functions return values if it succeeded.

`testObject:equals(actual, expected)` raises an error if actual differs from expected.

`testObject:differs(actual, expected)` raises an error if actual is equal to expected.

mad.utest.luaUnit

`mad.utest.luaUnit` is the main module to be used by `mad.tester`. By calling it, it returns a utest-object, which keeps track of statistics for one unit test-run. It also contains functions for deciding which unit tests to run and to run said unit tests. Its only public functions are `addModuleToTest`, `addFunctionToTest`, and `run`.

`addModuleToTest(modname)` adds modname to the table of modules to test, if the modname is not already in the table of modules already tested in this run.

`addFunctionToTest(modname,funname)` adds funname to the modnames key in the table of modules to test, if modname has not already been added as a module where all test functions should be called and funname has not already been run.

`run()` will run a module in the table of modules to test, then move it to the table of tested modules. It will keep doing this until the to test-table is empty, whereupon it will print the final results of the testing. The pseudo code of run is shown in listing 3.18.

Listing 3.18: Pseudo code showing what happens when unit tests are run

```
function run()
  while modtotestsize > 0 do
    modtotestsize = 0
    testtable()
  end
  display results
end

function testtable()
  foreach modname, val in modtotesttable do
    if testedmodules contains modname then continue end
    if type(val) == 'table' then
      get module modname and create testObject
      result:startModule(modname, testObject)
      foreach funname in val do
        runtestmethod(funname, module, testObject)
      end
    else
      runTestModuleByName(modname)
      testedmodules[modname] = true
    end
  end
end

function runTestModuleByName(modname)
  mod = modnames corresponding module
  create new testObject
  result:startModule(modname, testObject)
```

```
  foreach fun in mod.test except setUp, tearDown and self
      do
    mod.test:setUp()
    runTestMethod(funname, mod, fun, testObject)
    mod.test:tearDown()
  end
  runTestMethod('self', mod, mod.test.self, testObject)
end

function runTestMethod(name, mod, fun, testObject)
  result:startTest(name, testObject)
  call fun in protected mode
  if error then
    result:addFailure(errorMessage)
  result:endTest(testObject)
end
```

`testObject` is the instance of testObject which shall be made for each class. `result` is the utest-instances instance of `UnitResult` which keeps track of results and statistics and contains functions for printing said results.

The results will be printed in the same manner as in MAD-X, as shown in the example in listing 3.19.

Listing 3.19: Example output of a run of the unit tests

```
[some.module.to.test]
  functionToTest1  ( 0.01s )   4/4   : PASS
  functionToTest2  ( 0.13s ) 192/193: FAIL
some/module/to/test.lua:756: some error message
  self             ( 0.00s )   0/0   : NO TESTS RUN
[other.module.to.test]
  toTest1          ( 0.01s )   6/6   : PASS
  toTest2          ( 0.02s )  14/14  : PASS
  toTest3          ( 0.01s )   4/4   : PASS
  toTest4          ( 0.00s )   2/2   : PASS
  toTest5          ( 0.01s )   4/4   : PASS
  toTest6          ( 0.00s )   3/3   : PASS
  toTest7          ( 0.01s )   4/4   : PASS
Failed tests:
  functionToTest2 failed
  some/module/to/test.lua:756: some error message
Success : 90% - 9 / 10
```

3.6 Generator

To be able to run the code, one will need to generate an output that LuaJIT can run. LuaJIT can run both byte code and Lua code. It is easier to debug the

parsing steps if one can translate the input code to Lua, especially later when the parser will translate mad and madx-files to Lua, therefore it was decided to create a source code generator. The generator that has been implemented builds on the same structure as Nyangas source generator. The only public function is the modules call semantic, which returns one instance of the generator, sufficient to render one AST. The instance has the function `generate`, which calls the function `render` on the AST[12] and then returns the contents of its `writer` instance. The `writer` module is where the code is generated and it contains functions to write to a table where every generated line is kept and to get this table returned as an entire string. The `render` function runs the function corresponding to the nodes ast_id, which in turn may run render on its own nodes. Render also communicates with the errors module, which is introduced later in this chapter.

3.7 Interactive

The application should be able to run all its supported languages in interactive mode. Interactive mode should work just like Lua and LuaJITs interactive mode, i.e. each line is parsed as a chunk and if a line contains an unfinished statement, the application shall prompt for a new line to allow the user to finish the statement.

3.7.1 Previous works

Moonscript

Moonscript ([36] and [35]) is an extension of Lua that removes a lot of what it calls 'keyboard noise' from Lua and replaces it with white space. It is not interactive in itself, but its developer, Leafo, has added a small script written in Moonscript to add that possibility[35, wiki/Moonscriptrepl]. As this project is similar to the MAD project, this would be a solution made in heaven, but unfortunately the Moonscript-fix relies on changing the grammar for the new interactive part. To start a block in interactive mode Moonscript requires you to write \or tab, which means that one will not be able to copy-paste MAD-code into the interactive prompt which is a common way of using MAD-Xs interactive mode that users will be hard pressed to give up on. This approach is therefore scrapped.

Lua-interactive extension for local variables

David Manura and Shmuel Zeigerman have written a LuaJIT extension to make LuaJITs interactive mode support local variables. This extension does not do what is desired, but can work as an example of a way the desired results could be achieved. This extension can be viewed as consisting of two different extensions. One is the pure Lua-extension and another is a 'hack' that has to be done in the LuaJIT-code to make it compatible with the extension, as it relies on libraries that LuaJIT 2.0[13] is incompatible with. The 'hack'-part of the extension is obviously

[12] Which is i.e. the top node of the AST.
[13] And therefore probably not 2.1 either.

dangerous, as changing anything in the LuaJIT-code is both prone to errors that Mike Pall can not help with and a simple update of LuaJIT[14] will remove the hack. Therefore the lbci-library-dependency would need to be removed if this approach were to be used.

Nyanga

When the project was initiated, Nyanga did not have an interactive mode, but one was added in February 2014. This mode works by reading a line from stdin, then trying to parse said line. If the line compiles, it is run, otherwise the error is read. If the error is 'Unexpected end of input', a new line is read, the two lines concatenated and Nyanga attempts to compile the concatenation.

Eduardo Ochs unfinished fix

Eduardo Ochs tried to create a fix[38] to allow Luas interactive mode to be called from the script, which is something that is desired, but not necessarily needed, for MAD. Its pseudo code is shown in listing 3.20.

Listing 3.20: Eduardo Ochs unfinished fix to allow Lua to enter interactive mode from the script

```
while true do
  line = readline()
  code, unfinishedError, normalError = specialloadstring(
      line)
  if not code then
    while unfinishedError do
      line = line..readline()
      code, unfinishedError, normalError =
          specialloadstring(line)
      if normalError then
        break
      end
    end
    if normalError then
      error(normalError)
    end
  end
  run(code)
end

specialloadstring = function(line)
  code, error = loadstring(line)
  if error contains("Unexpected token") then
    return nil, error, nil
```

[14] Which will be needed, as LuaJIT 2.1 is still in a development phase.

```
  else
    return nil, nil, error
  end
end
```

Listing 3.20s approach is unstable, as it will break if one should change anything in the error reporting.

3.7.2 Implementation

The approach in listing 3.20 can inspire an approach that is quite similar to the approach that Nyanga ended up using. Its pseudo code can be seen in listing 3.21

Listing 3.21: Pseudo code for interactive mode

```
while true do
  initialize()
  line = getline()
  if not line then break end
  status, ast = pcall(parser.parse, parser, line, "stdin")
  while not status do
    if unfinished rule then
      newline = getline()
      if not newline then eof = true break end
      line = line..newline
      status, ast = pcall(parser.parse, parser, line, "
          stdin")
    else
      print error message
      break
    end
  end
  if status then
    run(ast)
  elseif eof then
    break
  end
end
```

To use the approach in listing 3.21 some assumptions have to be made about the parser. Mainly that it must manage to tell the difference between an unfinished line and a line that definitely has an error. Without this distinction, one would never be able to get out of an erroneous loop, because the parser would continuously wait for the next line that would make it all make sense. How to add this possibility will be shown in the next section about reporting errors correctly.

This way of implementing interactive mode makes it work like LuaJITs, except that while LuaJIT will quit once an error appears, MADs mode will simply print the error report and allow the user to continue coding. All code that has been run

before the error is still visible, as long as it is in the same lexical scope (i.e. global scope).

3.8 Report errors correctly

There are two different errors one may get when interpreting code: Syntax errors and run-time errors. This section will look into how both kinds of errors are reported to the correct line in the correct file.

3.8.1 Syntax errors

To properly match syntax errors to the correct line, one must transform the main rule of the grammar (chunk) to add error matching-support.

```
-- old rule
-- chunk <- ( block s ) -> chunk
-- new rule
chunk <- ( block s(!./''=>error)) -> chunk
```

This rule says that it will try to read a block and if the block is followed by anything other than white space, a rule has failed. The `error` function will then get the entire input file and the position of the token that was unexpected. Error may then read the file and find the line that failed.

The above rule can not tell the difference between code failing because of a syntax error and code failing because it is unfinished, which is needed because of interactive mode. To accomodate this, a new rule is added.

```
sp <- (''=>savePos)
```

```
function defs.savePos(_, pos)
  defs._lastPos = pos
  if pos > defs._maxPos then defs._maxPos = pos end
  return true
end
```

`sp` should be put in the end of every rule and after every string literal, to save the highest position the parser has managed to parse to before backtracking. If this position is the end of the file, the rule is unfinished and a different error message is given.

3.8.2 Run-time errors

How to properly report run-time errors is closely related to how one should run the compiled Lua code. To make Lua code runnable, one will need to load it using `load`. Load takes Lua code and an optional chunkname, which can be used to specify the name that shall be mapped to the code being run and then also to the error reports, and returns a function that can be run. When a run-time error occurs

during the running of said function, LuaJIT will raise an error with the chunkname given, the line in the loaded code where the error occurs and a string describing the error. If the code is run with `assert`, which is the standardised way of doing it, the program will fail and the error will be shown, but as the error happened in the created code, the information about line and file tells the user nothing about what parts of his code will have to be fixed.

To fix this, one can run loads returned function with a function from the protected call family, `pcall` or `xpcall`. The protected calls will call a function and catch any errors that are raised during the call, before returning `status` and `error`. If the call succeeded, status will be true and error will be the first return value of the called function. If the call fails, status will be false and error will be the error message that was raised. The difference between pcall and xpcall is that while pcall takes a function and then all the arguments to send to the function, xpcall takes the function and then another function to be used as an error handler function, e.g. to provide a proper traceback. This section will look into how to structure the code to use this to get errors reported to the correct line and file.

Information in AST nodes

All information the error reporters needs will have to be put in the AST nodes. The most important piece of information is what line the different nodes appear on. LPEG does not have any way of counting lines built in, one can only get the position in the character stream. Nyangas way of fixing this is to encode the position into the AST nodes and then parse the input file again to find the positions corresponding line. Depending on at which point one does the position to line-translation, one must either parse the input file twice during the parsing, losing valuable time, or keep the entire input file in memory. Both of these will have negative impacts to the performance and are therefore avoided.

The solution was to create LPEG rules to count the lines.

```
s   <- (ws / nl / cmt)*
any <- ch / nl
nl  <- [\n] -> newLine
```

```
function defs.newLine()
  defs._line = defs._line + 1
end
```

`any` will then be used instead of the `.` to mean any character, thus ensuring that when a rule is evaluated and a node is created, `defs._line` is at the correct line. The only problem is that for rules matching code spanning several lines, as e.g. for statements, the line will be the last line in the statement, while an error would normally be called at the first line. It will be shown later that this problem is handled by the error-modules implementation.

To facilitate later extensions to the language, most specifically the C-style include statement and require-functions updated to handle files that need interpreting, the file name of the input file will need to be stored in the AST nodes as well.

As the grammar does not need to worry about the files name and because the file name is consistent through an entire chunk, the file name is added post-parsing by the parser generator. It is then added to the chunk node.

Structure of line map

Now that the source code generator has access to the file names and lines of the different AST nodes, it can map its created output lines to the correct lines in the input file using the error modules `addToLineMap`-function. This function takes an input line, the created output line and the file name and adds this to a map kept in the error module. The maps structure is shown in listing 3.22.

Listing 3.22: The structure of the linemap mapping output lines to input lines and file names

```
linemap[chunkname][lineout] =
    { line = xx, fileName = 'somename' }
-- ex:
linemap['some_module'] =
    { 1 = { line = 1, fileName = 'some_module.mad'   },
      2 = { line = 4, fileName = 'some_module.mad'   },
      3 = { line = 1, fileName = 'other_module.mad' },
      4 = { line = 5, fileName = 'some_module.mad'   },
    }
linemap['some_other_module'] =
    { 1 = { line = 1, fileName = 'some_other_module.mad' },
      2 = { line = 1, fileName = 'some_other_module.mad' },
      3 = { line = 3, fileName = 'some_other_module.mad' },
      4 = { line = 5, fileName = 'some_other_module.mad' },
    }
```

The potential problem with reporting the error to a later line than where it should actually be reported is averted by the implementation of addToLineMap. If the output line already exists in the linemap to the specific chunk name, it will be overwritten if the new line number is lower than the old.

Using line map to correctly report error position

Once the line map has been properly filled by the source code generator, the code can be loaded and run by xpcall. Xpcall is used over pcall, because one may use a function based on `debug.traceback` which receives the current stack trace to get an error message and a stack trace to send to the `handleError`-function. `handleError`s pseudocode is shown in listing 3.23.

Listing 3.23: Pseudo code for handleError

```
function handleError(err,trace)
  chunkName = getChunkName(err)
  name, realLine = getNameAndLine( err , chunkName)
```

```
  errmess = getErrorMessage(err)
  errmess = name..":"..realLine..": "..errmess
  stack = getStackTraceBack(trace)
  stacktable = createStackTable(stack)
  errmess = errmess..createStackErrorMessage(stacktable)
  return errmess
end

function createStackTable(stack)
  stacktable = {}
  for each line s in stack do
    chunkName = getChunkName(s)
    name, line = getNameAndLine( s, chunkName )
    stacktable[#stacktable+1] = "\t"..name..":"..line
        ..":"..getErrorMessage(s)
  end
  return stacktable
end
```

Listing 3.23 translates the line and filename for all modules that have been interpreted by the application, including the stack trace. To further raise the error, one cannot use Luas `error` function, as this function concatenates a new stack trace to the error message, and said stack trace traces back to the function that called the error function. Instead, the error message containing the proper, translated stack trace is written to stderr before os.exit(-1) is called.

3.9 Extending Lua - Making the new MAD

With the application ready to interpret Lua files, the next step is to show the concept by adding a new language to the list of supported languages. The first language to be added is a simple superset of Lua, thus something that should be easy to implement. This section will show how MAD is implemented.

3.9.1 Include

The new MAD should support C-style include statements as this is a feature many of MADs potential users will be used to from C/C++. Another positive side of include is that one may allow several files to share the same lexical scope, sharing local variables without having to make them global. An include statement should take a file name and immediately parse said file, adding the returned AST as a node to the AST the parser is already building. This will then be translated to a single file to be run by LuaJIT. Because of the way the error reporting-module is implemented, errors will still be reported to the correct lines in the correct files.

To use an include statement in MAD, one will need to write `@include "name/of.file"`. The @ is there to set the include statement apart from a regular function call. The

reason for this is that while a function call is evaluated at run-time, include statements are evaluated at compile-time. When the @ is there, users will hopefully not be confused. The new rule added to the original Lua grammar is seen in listing 3.24.

Listing 3.24: Addition to the Lua grammar to allow for include statements

```
stmt    <- s((include s{string} sp) => include
             / ... all other statements)
include <- s'@include'e sp
```

`e` in this grammar is a rule checking whether the next character is an alphanumeric character or not, thus making sure that `@includes` is not read as an include. According to listing 3.24, rules written on the form `@include "filename"` will trigger an immediate call to `lang.mad.defs.include`, with the current string being parsed, the position in the string and the string `"filename"` as arguments. `lang.mad.defs.include` is shown in listing 3.25.

Listing 3.25: The function used support C-style include statements

```
function defs.include( _, _, name )
  name = string.sub(name, 2, string.len(name)-1)
  local lang = require"mad.lang"
  local file = assert(io.open(name, 'r'))
  local input = file:read('*a')
  file:close()
  local parser = lang.getParser(lang.getCurrentKey())
  local ast = parser:parse(input, name)
  return true, table.unpack(ast.block)
end
```

The function `defs.include` from listing 3.25 opens and reads the file given by `name`, before it parses it using a new instance of the same parser that already parses the current file. Its return values are `true`, signifying that the immediate evaluation has consumed no input in the upper level file, and the list of statements that were in the included file. These statements will be inserted into the AST built in the original file.

3.9.2 Lambda

It was desired to have lambda functions in MAD. Lambdas shall either be short form for defining anonymous functions, or functions that may be called without parentheses if they take no arguments. These lambdas without arguments, or deferred lambdas, shall allow the users to not know whether a variable is actually a lambda, or if it is a regular variable. If a script is written for regular variables, it shall be possible to make its inputs deferred lambdas without changing the script. In physics-heavy code as this will be, being able to write `a+b+c/d` will greatly improve readability compared to `a()+b()+c()/d()`.

Grammar

Lambda functions shall be able to be defined in some sort of the form `\x -x`. This shall be translated to a function that takes `x` as an argument and returns `-x`. It is desired to be able to write the lambdas with as few parentheses as possible, but if they are needed to avoid ambiguities they are allowed.

Listing 3.26: Grammar addition to allow lambda functions

```
lambda = '\', [namenospace, [',', namelist]], ( '(',
    explist, ')' | exp );
```

Listing 3.26 shows the rule that was added to the Lua grammar to make the MAD grammar. `namenospace` is a name that does not follow the same standard that was described earlier, as it does not remove white space in front of it. To force the functions first parameter to be immediately connected to the \allows us to define lambdas without arguments in a short and concise manner: `\ 1`. This lambda takes no arguments and always returns 1. To remove ambiguities, one must use parentheses when having several return values, otherwise one would not know how to translate `\a \b a+b, -a`. Should the `-a` be returned by the upper or lower lambda? By forcing parentheses for multiple returns, this will never be a problem. The example can the either be translated to `\a (\b a+b, -a)` or to `\a \b (a+b, -a)`, depending on what was desired.

How to get deferred evaluation in Lua code

By translating a lambda function to a standard function-node, but with its kind set to 'lambda', the Lua code generator will translate it just as if it was an anonymous function. This will work for all kinds of lambdas with arguments, but as lambdas without arguments should act as the deferred expressions described in the section about MAD-X, a better way of translating them will be needed.

One possible solution is to create two functions `lambda.new` and `lambda.eval` which could handle lambda functions. New takes a function as argument and returns a table with its `_lambda`-key set to the amount of parameters the function has and the `_func`-key set to equal the anonymous function that was the argument. It also gives the table a call-semantic, to allow it to be used as a standard anonymous function. The eval-function checks whether its argument is a table with _lambda set. If it is set to zero, eval calls _func and returns its value, and if it is set to something else or not existing, it returns the variable itself.

The problem with this approach is to figure out when the eval-function should be used. With a perfect set up, eval should be called on every lambda function and on no other functions. Unfortunately, because one does not declare variables in Lua, any variable could potentially contain a lambda. If the eval and new-functions should be added to the AST during the parsing step, eval would need to be added to every single variable. This will clutter the created code and create an unnecessary amount of function calls and tests for lambda. By making an amendment to the lambdas documentation saying that a variable can either be a variable or a lambda, i.e. if it has been a lambda once, it will always be treated

as a lambda, and if it has been something other than lambda once, it will never be treated as a lambda, one may append the eval-function only to variables that are lambdas or not yet defined. This may still not be done in the parsing step, as LPEG evaluates the lowest level rules first, meaning that one can not know the context at parse-time. By adding new and eval in the transformer-step instead of in the parser-step, one may keep track of the context and thus only add eval to the necessary variables. This approach is simple to implement, but forcing variables to be a specific kind goes against how Lua works in general and will probably be a source for problems where users assume lambdas work like other kinds of variables, even if it is documented that they do not. The same problem applies to the fact that lambdas will be unable to be passed by reference, which all other functions may.

Storing lambdas as tables with a call semantic may still be used, as one may differentiate between normal functions and lambdas and even add a new type 'lambda' as seen in listing 3.27. Ehat needs to be changed is how lambda functions are evaluated.

Listing 3.27: Adding a new type 'lambda'

```
local oldType = _G.type
_G.type = function(obj)
  if oldType(obj) == "table" and obj.__lambda then
    return "lambda"
  else
    return oldType(obj)
  end
end
```

Knowing the amount of parameters the function needs is not needed and the function will therefore be stored in `__lambda` to not unnecessarily fill the tables.

The approach being used in the implementation builds from the fact that most usages of deferred lambdas will be in arithmetic or logical expressions, which Lua allows us to overload using metamethods. By overloading all the operators to lazily evaluate lambdas, as one can see in listing 3.28, most uses of deferred lambdas are covered.

Listing 3.28: Overloading + operator to support deferred evaluation of lambda

```
local eval = function(lhs,rhs)
    if is_lambda(lhs) then
        lhs = lhs.__lambda()
    end
    if is_lambda(lhs) then
        rhs = rhs.__lambda()
    end
    return lhs, rhs
end
mt.__add = function(lhs, rhs)
```

```
    lhs, rhs = eval(lhs, rhs)
    return lhs+rhs
end
```

The structure from listing 3.28 works for all logical and arithmetic operators, except for == and ∼=. The reason for this is that Lua checks if the types of the two arguments are equal before it looks for metamethods. Tostring have been added for strings, and #, pairs, ipairs, index, and newindex have been overloaded for lambdas returning tables, making them work just as if they are ordinary strings/tables. The `__lambda`-key will be off limits to users, as this is used specifically by the lambda function. If a user tries to use it, the index-metamethod will never be called and the function will be returned instead of what the user looked for.

There are still a couple of uses left where one might use deferred evaluation of lambdas, namely the string, table, and math-modules within Lua. To overload the functions contained in those three libraries, three mirror modules will be created in `lang.lambda`, to be run when `lang.lambda.lua` is run, allowing deferred evaluation to be turned off. All overloaded functions in the two libraries will follow the convention shown in listing 3.29.

Listing 3.29: Overloading math.abs to support deferred evaluation of lambda

```
local mabs = math.abs
math.abs = function(x)
  while is_lambda(x) do
    x = x.__lambda()
  end
  return mabs(x)
end
```

Listing 3.29 will evaluate all embedded lambdas in the arguments, then call the original function with that as its argument. As these functions are not based on metamethods, they will be overloaded for all inputs. Therefore, every time math.abs is called, it will have at least one test more than normal, thus slowing the program a little bit down. As long as it is only one test and because most scripts will not be mostly math/string/table-based , this is not going to be an issue. There are several functions that may not be overloaded in a manner that does not slow the application down too much, mainly functions with a variable amount of arguments, e.g. math.max. Math.max finds the maximum out of all its arguments, which is an unknown number because of the ...-operator. If this were to be overloaded, it would be something closely related to listing 3.30.

Listing 3.30: Overloading math.max to support deferred evaluation of lambdas

```
local mmax = math.max
math.max = function(...)
  local tbl = {...}
  for i,v in ipairs(tbl) end
    while is_lambda(v) do
      tbl[i] = v.__lambda()
```

```
    end
  end
  return mmax(table.unpack(tbl))
end
```

This is slow for several reasons, both because it creates a table unnecessarily and most importantly because it calls table.unpack, which is one of the non JIT-compiled functions. Other functions not working because of this problem are table.pack, string.format and math.min.

Changing Lua generator

Instead of transforming the AST to add support for lambdas, as the first approach would do, the Lua source code generator is changed. The generator gets an option as to whether it should support lambdas or not, defaulting to not. Without support, it will simply translate all lambdas to anonymous functions, while when it supports them, it will translate `\a,b a+b` to `require"mad.lang.lambda"(function(a,b) return a+b end)`, where lambdas call semantic will build the lambda table.

3.9.3 Adding a MAD source code generator

When adding a new language, one may also add a new source code generator. Adding one for MAD is a simple task, as the lambda statement is the only difference between the languages since the include statement is not added to the AST. A new file `mad.lua` is added to `lang/generator/` and this file is added to `lang.generator.lua`s table of generators under the key 'mad'. Mad.lua is an exact copy of the Lua generator, except that mad translates lambda functions to the form `\x,y x+y`.

3.10 Making a MAD-X parser

The last proof of concept for the created framework will be to create an interpreter for a language that does not extend Lua, to show that most languages can be implemented in it. This section will go through how the MAD-X language was implemented in the framework, showing the problems that surfaced and how they were handled. The Lua-translation of MAD-X, which the framework is supposed to translate MAD-X to, is a moving target, as it is being developed in parallel with the MAD-X interpreter. Thus, what to translate to may be viewed as a concept more than a fixed target, and some parts may be changed completely.

3.10.1 Grammar

As shown in the section about MAD-X, the language is context dependent with a very short actual grammar, see listing 2.2. This section is about how the 'basic' structures are parsed and translated, while the more specific parts are handled later.

Lexical elements

There are two major differences between names in MAD-X and names in Lua. The first is that MAD-X is case insensitive, meaning that `Mad` and `MAD` is the same name. The second is that . is allowed in MAD-X-names, while it means table access in Lua. The two places this matters is in the keywords and in the names. The lexical definition of all keywords is therefore written on the form

```
return <- s[rR][eE][tT][uU][rR][nN]sp
```

thus removing the problem. When a name is parsed, all letters are put in lowercase, all underscores are translated to double underscores and all periods are translated to simple underscores. The proper name is needed to communicate with the database containing the structure of the accelerator, which means that it needs to be stored in the AST as well. Therefore, all `name`-nodes also have a leaf named `strname` containing the untranslated name of the variable.

MAD-X allow some short cuts that Lua would not allow and translating these as soon as possible will help simplify the later steps. One of these short cuts is that MAD-X allows numbers to be written .5 or 5., which needs to be translated to 0.5 and 5.0. This can easily be done in LPEG, as seen in listing 3.31.

Listing 3.31: MAD-X grammar to allow .5 and 5. as numbers

```
decnum <- {~((num ('.' num / '.'->'.0')?) / (('.'->'0.')
    num)) ([eE] sign? num)?~}
```

Listing 3.31 translates to *"A decimal number is a number followed by an optional dot then number, or an optional dot which should then be translated to '.0'. A decimal number may also be a . followed be a number, where the dot is translated to '0.'. This can all be followed by an 'e' followed by a number for exponentiation."*.

Another short cut that must be supported is writing strings with no string delimiters, if what takes the string is incapable of taking variables. Thus a similar approach as in listing 3.31 is followed, as seen in listing 3.32.

Listing 3.32: MAD-X grammar to allow strings without string delimiters

```
madstr <- s(string / ((''->'"') {(!(',' / ';') any)+}
    (''->'"'))sp) -> literal
```

Listing 3.32 reads a string, just as in the Lua grammar, but silently inserts a " before and after the string.

Expression

Expressions in MAD-X are implemented the same way they are implemented in Lua, except that &&, ||, and <> are translated to *and*, *or*, and ~=. This will make MAD accept all valid MAD-X-expressions, but it will also allow and- and or-expressions to contain more than one and or or. As this is merely a shortcoming in the MAD-X parser, there is no reason to implement it in MAD. This can rather be used as an 'improvement' to sell the new application to the hard core MAD-X users who do not want to change.

Statements

Listing 3.33: MAD-X grammar for statements

```
stmt       <- (s( assignstmt / macrodef / macrocall /
    defassign / linestmt / lblstmt / cmdstmt / retstmt /
    ifstmt / whilestmt )sp)

assignstmt <- (real? const? assign)
assign     <- (name s'='sp exp      sp) -> assign
defassign  <- (name s':'s'='sp exp sp) -> defassign
retstmt    <- (return explist?      sp) -> retstmt
ifstmt     <- ({if s'('sp exp s')'sp block
                (elseif s'('sp exp s')'sp block)*
                (else block)?} sp) -> ifstmt
whilestmt  <- (while s'('sp exp s')'sp block sp) ->
    whilestmt
```

Listing 3.33 shows the grammar for the simplest statements in MAD-X. Assign, retstmt, ifstmt and while are all translated to assign, return, if, and while AST nodes, while defassign is translated to an assignment statement where the right hand side is a lambda function returning its expression. While and if are also upgrades compared to regular MAD-X, as the earlier versions are only able to take one level of embedded ifs or whiles, while this version manages as many as one would like.

Label and command statements

Listing 3.34: Implementation of label- and command-statements in MAD-X grammar

```
lblstmt    <- (name s':'sp sc name (s','?sp attrlist)? sp)
   -> lblstmt
cmdstmt    <- (sc name s','?sp attrlist?    sp) -> cmdstmt
attr       <- (attrassign / defassign / exp sp) -> attr
attrlist   <- (attr (s','sp attr)*)

attrassign <- (((name=>chkeystr) s'='sp madstr) -> keystr /
    ((name=>chkeystrtbl) s'='sp s'{'?sp (madstr (s','sp
    madstr)*)? s'}'?sp) -> keystrtbl / assign)

sc         <- (''=>saveclasspre)
```

Listing 3.34 shows the implementation of the label- and command-statements from MAD-X. The essence of listing 3.34 (and the AST building functions behind it) is that:

```
label:command1, key1 = val1, key2 = val2;
command2, key3 = val3;
```

Should be translated to:

```
translatedlabel = command1 "label" {key1 = val1, key2 = val2}

command2{key3 = val3}
or, if command2 is a label that has already been defined
command2:set{key3 = val3}
```

The label statement and the second command statement utilizes the object module `mad.object.lua` created by Deniau. Command1 will be the class of the variable translatedlabel, having all the same attributes, as well as having key1 and key2. Calling the class will make translatedlabel a class of its own that can be used to create further children. It also adds the function set, which can be used to update the values.

The rest of the grammar is about making the grammar accept parts of MAD-X that MAD-X should not have accepted in the first place. The main one is accepting the previously discussed MAD-X string without delimiters, as explained above. A problem with the strings without delimiters is that they can only be allowed when the attribute they are connected to is expecting a string or an array of strings. In MAD-X, this is connected to the file mad_dict.c. This file contains a list of all commands accepted by MAD-X. Each command has a list of all keys accepted and what type each key is expecting, as well as what values the keys shall have if they are absent. This application will not be checking whether keys or inputs are correct, rather raising run-time errors when the commands are run with incompatible keys. The only thing needed from mad_dict.c is which command-key pairs take strings as values and which take arrays of strings. To achieve this, a script `mad.lang.madx.translateMad_dict` is created. This script is an LPEG based parser reading mad_dict.c and creating a table containing all command-key pairs, their expected types and their default values, to be stored in mad_dict.lua. This is meant to not be needed to be run again, but in case changes are made to mad_dict.c, the script is kept.

Listing 3.35: Functions to check whether a key should have a string or not as argument

```
local preclasspos, class

function defs.saveclasspre(str,pos)
  class = string.sub(str,string.find(str, "[%w_$%.]+", pos)
      or 0)
  return true
end

function defs.chkeystr(_,_,name)
  if dict[class] and dict[class][name.name]
  and dict[class][name.name].type == 's'
  and not dict[class][name.name].isarray then
    return true, name
```

```
  end
  return false
end
```

Listing 3.35 shows the implementation of some of the actions used in the grammar in listing 3.34. `Saveclasspre/sc` is used to store the name of the class the attributes are added to. It consumes no input and returns no values. It could have consumed the name and returned it, but this would have possibly lead to an LPEG error stating that there are too many match-time captures, as LPEG can only handle a certain amount of match-time captures. After storing the class name, the parser will get to the `attrassign`-rule. This rule will check if a specific key expects a string by calling `chkeystr` on the keys name. Chkeystr will test whether the class/key-pair stored in dict, which is the table contained in mad_dict.lua, expects a string input. If it does, chkeystr returns success and the captured name, otherwise it returns false. The reason why chkeystr and chkeystrtbl returns their captured values, when saveclasspre does not, is that it is faster to only parse the string once and since both chkey-functions most often will return false, there will not be too many match-time captures. If one of the chkey-functions returns true, the attribute will allow strings with or without delimiters, otherwise it will interpret everything without string delimiters as identifiers.

Sequence

Sequences are the main structure in MAD-X, they are where all the magnets are stored and show the entire structure of the accelerator, thus they are among the first things to develop when porting the language to another medium, as this project is doing to MAD-X. The `sequence` and `endsequence` commands will enter and leave the sequence context, thus changing the meaning of the statements within. The sequence part of the sequence created in listing 2.3 should be translated to the Lua-sequence in listing 3.36.

Listing 3.36: A sequence created in Lua

```
local seq = require 'mad.sequence'
bad_accelerator = seq 'BAD.Accelerator' { length = 3 }
bad_accelerator:add{ qpp 'QPpl' {}, at = 0 }
bad_accelerator:add{ dpp 'DPpl' {}, at = 1 }
bad_accelerator:add{ qpm 'QPml' {}, at = 1.5 }
bad_accelerator:add{ dpm 'DPml' {}, at = 2.5 }
bad_accelerator:done()
```

Listing 3.36 utilizes the object- and sequence-modules by Deniau. Mad.sequence can be called as a constructor to initialize a sequence with certain attributes set. One may then add elements to the sequence using add. The elements that are created while adding to the sequence will also be added to the global scope to allow it to be updated later. Done will finalize the sequence.

This is implemented by having a test in labelstatements corresponding action, checking whether the name of the command is sequence or endsequence. If it is sequence, it will save the sequences label to a local variable. Labelstatements

following this will be translated to additions to the sequence instead of as assignments as they normally are. Once an endsequence appears, it goes back to normal. The pseudo code for the implementation of command and label statements AST creation can be seen in listing 3.37.

Listing 3.37: Pseudo code for the implementation of labelstmt and cmdstmt

```
function defs.lblstmt ( name, class, ... ) -- ... =
    attrlist
  __name[name.name] = 'label'
  if seqedit then
    return sequenceAddition(name, class, ...)
  end
  if class.name == 'sequence' then
    seqedit = name
  end
  return ast node for name = class "name" {...}
end

function defs.cmdstmt ( class, ... )
  if __name[class.name] == 'label' then
    return ast node for class:set{...}
  end
  if seqedit and class.name == "endsequence" then
    local name = seqedit
    seqedit = nil
    return ast node for name:done()
  end
  return ast node for class {...}
end
```

Line

To support MAD-X, MAD will also need to support parts of MAD8/9, as MAD-X supports these. Therefore, MAD will need to support line-statements.

Listing 3.38: Grammar definition of line statements

```
linestmt <- (name ls ':' sp line ls '=' sp linector        sp)
   -> linestmt
linector <- (ls '(' sp linedef ls ')'                      sp)
   -> linector
linedef  <- (linepart (ls ',' sp linepart)*)
linepart <- (ls(invert / times / ls name / linector)       sp)
   -> linepart
invert   <- (ls '-' sp linepart                            sp)
   -> invertline
```

```
times     <- (ls((number->literal)sp ls'*'sp linepart) sp)
   -> timesline
```

Listing 3.38 shows the grammatical implementation of the line statement. `ls` is a version of `s` which also allows & , as these where used as line continuation in MAD8/9. They are only allowed in line statements and not in the rest of the grammar as they are unnecessary and should be removed, but are still used in older pieces of code. Line statements are not special in the new MAD and will therefore be translated directly to sequences in the created Lua code.

Macro

There are two different kinds of macro statements described in the section about MAD-X: Calls and definitions. A definition defines a macro by a string containing MAD-X-code and a set of string-parameters describing which strings to substitute with the input parameters. Calls exchange all instances of the macros parameters by its own arguments, before parsing and running the code in the macros string.

A MAD-X macro definition as the one seen below

```
label(one, "two"): macro = { first = one; second = two; }
```

is translated to:

```
label = { str = [===[ first = one; second = two; ]===],
          par = { 'one', "two" } }
```

And the implementation is shown in listing 3.39.

Listing 3.39: Implementation of macro definitions

```
macrodef   <- (name parlist? s':'sp macro s'='?sp
   macroblock sp) -> macrodef
macroblock <- s'{'sp {((!('{'/'}') any) / balanced)*} '}'sp
parlist    <- (s'('sp macrostr (s','sp macrostr )* s')'sp)
   -> parlist
macrostr   <- s(string->literal / {[^%s),]+}) sp
balanced   <- '{' ((!('{'/'}') any) / balanced)* '}'
```

```
function defs.macrodef(label, parlist, str)
  if not str then str = parlist parlist = {} end
  local par = {ast_id = "tbldef"}
  for i,v in ipairs(parlist) do
    if not v.ast_id then
      par[i] = ast node for  "'"..v.."'"
    else
      par[i] = v
    end
  end
  return ast node for label = { str = "[===["..str
     .."]===]", par = par }
```

```
end
```

The parameters sent to a macro are supposed to be strings, but as explained earlier, they may be without delimiters, where the delimiter will be a space, comma or closing parenthesis. The macro string does not need to abide any grammatical rules, except that it needs to have balanced curly brackets to allow embedded if, while, and macro-statements without finishing its string. When run, the Lua code will create a global table named the macros name, which can be parsed and run by the `execmacro`-function.

Macro calls are translated from the form

```
exec, label(2,$num)
```

to:

```
execmacro( label, [[2]], num )
```

The macro calls implementation is quite simple, as one can see in 3.40. One needs to keep track of $, as this means that the following word should be parsed as a name, not as a string, otherwise the translation is straight forward.

Listing 3.40: Implementation of macro calls

```
macrocall  <- (exec s','sp name (s'('sp macroarg s')')? sp)
    -> macrocall
macroarg   <- (s{'$'?(number? ident / number)} sp (s','sp s
   {'$'?(number? ident / number)} sp )*)
```

```
function defs.macrocall(name, ...)
    for i,v in ipairs{...} do
        local nam
        if string.find(v,"$") == 1 then
            nam = ast node for the identifier string.sub(v
                ,2)
        else
            nam = ast node for "[["..v.."]]"
        end
        arg[i] = ast node for tostring(nam)
    end
    return ast node for execmacro(name, arg[1], arg[2],
        ...)
end
```

What makes the implemented Lua macro system work, is the function `execmacro` which is shown in listing 3.41.

Listing 3.41: Pseudo code for the function execmacro

```
__macrono = 1
function execmacro(macro, arg)
  par, str = macro.par, macro.str
```

```
  for key,value in par do
    str = string.gsub(str,value,arg[key] or '')
  end
  ast = parser:parse(str, 'macrono'..__macrono)
  errors.setCurrentChunkName('macrono'..__macrono)
  code = generator:generate(ast)
  loadedCode, err = load(code, '@macrono'..__macrono)
  __macrono = __macrono + 1
  run loaded code and handle errors
end
```

The function takes the name of the macro to be executed and a table of arguments to be exchanged for the macros parameters. It then substitutes every instance of par_i with arg_i and parses the resulting string with the MAD-X parser. It then generates code, runs it and handles run-time errors occurring during the run. The code needs to be generated and loaded with a unique filename, because of how the error handling is implemented. A counter is therefore kept, incrementing every time a macro has been run. Thus, every called macro will have a name on the form `macrono1`, where 1 can be any number.

3.10.2 Finalisation

With all the statements implemented, it is possible to parse .madx- and .seq-files. The files can not be run, as MAD-X contains several global functions that are used extensively in the code. These functions have yet to be implemented in the new MAD application and one may therefore only run the application on MAD-X code, dumping the created Lua or MAD code, and check the dumped code by hand to see if it is the same as the code one should create. By making empty versions of all the missing MAD-X-functions, one may also run the code to see if there are any errors in the sequence or object modules.

This was attempted, using the same grammar/defs/parser-setup as with the mad and lua parsers, running the entire chunk. But when running this on large files, e.g. the sequence definition of the LHC which is 25k lines, LuaJIT raises an error about 'main function has more than 65536 constants'. This error stems from LuaJIT having different constraints than regular Lua, as Lua would manage to run this function. The constraint is on the amount of nested tables one may have in a single table, which the AST has in abundance. This is an issue that there probably will not be a fix for within a discernible future (see [27]), changes therefore need to be made in the MAD-X parser.

The problem appears while building the AST, as the AST is the table that exceeds LuaJITs constraints. There are a few different ways to solve this problem, all of which will be discussed below.

Change LuaJIT.

Since the problem exists in LuaJIT and not in Lua, one could change the limit in LuaJIT and make the program run. This approach has several downsides though.

First of all, it will take a long time to figure out which parts of LuaJIT to change and which parts not to change. This is far beyond the scope of this project, but as it has been discussed to extend LuaJIT to support lambda functions directly (see [12]), this could go into the same project. This change will hurt LuaJITs performance, as the constraint is there for a reason. It will also be more difficult to update LuaJIT, if one shall have local fixes that LuaJIT does not support. A simple bugfix could destroy the change. This is therefore not used and the only possible solutions amount to somehow shrink the size of the created tables.

Change the definition of the AST.

By somehow changing the allowed nodes for the AST, one could theoretically heighten the amount of information stored per table, thus lessening the amount of nodes/tables needed for the total AST. To heighten the storage of each AST node, one will need more and more specialised nodes, thus making the AST less suitable for different kinds of languages and making the modules needing to traverse the AST more complex. One will also need to change several of the previously implemented and tested modules. Lastly, even if one should manage to find a good compromise for the ASTs generality and its amount of nodes, it is unlikely that one would manage to get a bigger gain than removing enough nodes to just barely manage to parse the LHC sequence. When longer sequences as the CLIC and the FCC are to be introduced, the error will pop back up.

Change how to translate the MAD-X code to Lua.

One of the reasons why this problem appears when translating MAD-X and not Lua and MAD is because one simple MAD-X statement is translated to several statements in Lua, thus expanding the AST. Another reason why this is a big problem in MAD-X and not Lua is because (properly written) Lua files should not be 40000 lines long, while sequence definitions for large accelerators written in MAD-X easily approach that size. The size of the AST can be shrunk by translating the MAD-X code to fewer lines. One of the changes that were made because of this is how the sequence definitions were translated. Initially this was done as in listing 3.42, but to trim the ASTs size it was changed to the one in listing 3.36.

Listing 3.42: First attempt at translating MAD-X sequence to Lua

```
bad_accelerator = seq 'BAD.Accelerator' { length = 3 }
qppl = qpp 'QPpl' {}
bad_accelerator:add{ qppl, at = 0 }
bad_accelerator:done()
```

This approach adds the `qppl` to the global environment before adding it to the sequence, while the approach in listing 3.36 assumes that `add` does that job within the function. With this change, one can manage to trim some percentages of the ASTs size, but as with the last solution, this will only make it barely work for the LHC, not for the FCC and CLIC. Changing how to translate the code can be good

solutions if it does not impact the performance too much, but it can never be the final solution as it will never shrink the AST enough.

Call each statement separately.

As all variables in MAD-X are global, scope does not matter as much as it does in Lua and MAD. In Lua, one would break the lexical scoping by calling every statement as its own separate chunk, but with MAD-X one may do it and still keep all variables reachable. Since starting and stopping the parser to generate code and call the statements for every single statement will be slow, it is possible to bundle a certain amount of statements together and call them. The precise number of statements that can be run should be tested to find a number that does not slow the process down too much, but which is low enough to guarantee to a certain extent that the code will be runnable by LuaJIT. As the Lua modules that shall support MAD-X are not ready yet, it is impossible to run the code created and get proper results for the speed. Because of this, 1000 is guesstimated to be around the proper size. Tests with dummy functions instead of the proper mad-lua functions resulted in less than 10% difference in parsing time compared to running everything in one bulk when tested on files small enough to test in one bulk, which is an acceptable trade off for code that actually runs.

There are two different ways of changing the parser to run a certain amount of statements when they are being parsed. Their main difference is whether LPEG or the parser itself should keep track of the position in the file being parsed. The LPEG approach changes the grammar to be as in listing 3.43 and the new function from listing 3.44.

Listing 3.43: Change made to chunk to call every 1000 statements allowing LPEG to handle position

```
chunk    <- (('' => setup) (stmtnum)* s(!./'' => error)) ->
    chunk
stmtnum <- (stmt s'; 'sp(stmt s'; 'sp)^-1000)      => stmtnum
```

Listing 3.44: New function to that runs an arbitrary number of statements.

```
local chunknum = 1
function defs.stmtnum( str, pos, ... )
  if defs._run then
    generate code from AST containing statements from ...
        and filename 'chunkno'..chunknum
    load and run code with name 'chunkno'..chunknum
    if error then handle error end
    chunknum = chunknum+1
  end
  return true
end
```

When using this approach, `parser.parse` only needs to set the variable `defs._run`

before running the code. When disregarding that single variable, the parser will not need to know that the functions are being run while parsing.

The second approach has a grammar (listing 3.45) that returns an AST at once when it has reached 1000 statements, stopping the parsing.

Listing 3.45: Change made to chunk to return every 1000 statements, needing the user to handle the position

```
chunk    <- (('' => setup) (stmtnum .*)? s(!./'' => error))
stmtnum <- (stmt s';'sp(stmt s';'sp)^-1000) => stmtnum
```

Listing 3.46: Function that returns the position and an arbitrary number of statements

```
function defs.stmtnum( str, pos, ... )
  return true, pos, AST containing statements from ...
end
```

To use this approach, the function `parser.parse` will need to be changed as well, as can be seen in listing 3.47.

Listing 3.47: How the parser function needs to look if it shall keep track of the position instead of LPEG

```
local parse = function (self, str, fileName, pos)
  local pos, chunknum, ast = pos or 0, 1, nil
  while pos < string.len(str) do
    pos, ast = self.grammar:match(str,pos)
    generate code AST and filename 'chunkno'..chunknum
    load and run code with name 'chunkno'..chunknum
    if error then
      handle error
      break
    end
    chunknum = chunknum+1
  end
end
```

With both functions, a small change has to be made in `mad.core.exec`, the module running all the interpreters. Where it normally would treat all languages the same, creating an AST, generating code, then running the code, it must now act this way for all languages that are not MAD-X. For MAD-X, it shall only generate the AST and not run any code afterwards as this has already been done. The pseudo code for the changed exec can be seen in listing 3.48.

Listing 3.48: Pseudo code for the function executing the interpreters, disregarding option handling

```
function exec(_, options)
  for _, fileName in ipairs(options.files) do
```

```
    errors.setCurrentChunkName(fileName)
    inputStream = read file fileName
    parser = lang.getParser(file extenstion)
    if evaluating MAD-X then
      ast = parser:parse(inputStream, fileName)
    else
      ast = parser:parse(inputStream, fileName)
      source = generator:generate(ast))
      load and call code with filename fileName
      if error then
        handle error
      end
    end
  end
end
```

When testing the two approaches, it was discovered that the one letting LPEG keep track of its own variables was 10% faster, which was therefore kept.

3.11 Future works

One could write an entire report only describing the future works of the MAD application, as there are a huge amount of modules that need to be added before MAD is a working language, e.g. physics, graphical interfaces, web interfaces, finishing the Lua description of the sequence/modules/etc. This report will only focus on the potential future works to make the authors part of the application better, and come with suggestions to how one can fix these problems.

3.11.1 Make it work on Windows

The most important aspect lacking to be able to properly "sell" the new MAD application to the MAD-X users and to reach a broader user group, is the ability to run on Windows as well as MAC and Linux. The main reason why this does not work as of now, is because LPEG has been made without Windows in mind and does therefore not work out of the box on Windows. This has been mentioned several times on LPEGs mailing list, and `http://lua-users.org/lists/lua-l/2007-05/msg00364.html` contains several solutions to the problem. The specific post that the link redirects to contains a makefile solution, which can be added to LPEGs makefile.

3.11.2 Make a unified makefile system

At the time of writing this report, LPEG is the only external library used in the application and the only module that needs to be built before the application can be run, as all the other modules are pure Lua scripts. It is therefore not a problem to have the documentation telling the users that one will have to make LPEG

before one can run it. Later on, when more external libraries and modules are added, forcing the user to make every single module beforehand will be too taxing and force users away from the system. What one should do then, is to create a makefile system that does this. The makefile should build the different modules and set up the necessary variables to allow the system to be run without problems. As long as the problem in the next section is unfixed, this can also work as a sort of installation, making the user only need to know of the mad-e script.

3.11.3 Single binary

As MAD-X comes packaged in a single binary file, it is desired that MAD should do the same. There are two reasons why this is required/desired.

1. Old MAD-X users will want to have it this way, as this is how it has always been.

2. MAD support and development wants to be able to be in full control of what the users can do. If the users can not change anything in the file, support will not need to try to find bugs that were introduced by the user.

An initial search for how this has been done ended up yielding some questions on stack overflow and some threads on Luas mailing list, which shall hopefully be enough to make this work. [47] contains an article by Han Zhao about how to package scripts, [11] contains a long mailing list discussion with different suggestions as to how it can be solved and links to repositories containing projects using said results. In response to [17], a brief introduction into how to keep LuaJIT-compiled modules when having a single binary is given.

Should one still fail to make a single binary, the second best will be to have a simple installation process that makes the system and builds all needed modules, much like the makefile does.

3.11.4 Adding a bytecode generator

An addition that is unnecessary but practical, is a bytecode generator. Generating bytecode is not needed to make the application run correctly, but it may make it run faster. One may also append debug information to the bytecode, thus allowing the errors-module to store less information and lessening its size. To do this, one may do as the LuaJIT Language Toolkit and extend Nyangas bytecode generator, or one may make a completely new one. One will have to change the logic in the generator no matter what, as Nyanga uses a completely different AST than MAD, but Nyangas module is well structured, which means that it should be easy to use the structure and create ones own bytecode.

3.11.5 Finish the Lua implementation of MAD-X

As the Lua implementation of MAD-X is unfinished, it is hard to properly test the MAD-X interpreter. When this is finished, one can add proper tests and

improve the already existing ones. One should also do a full body of speed tests on the interpreter to find the proper number of statements to call when interpreting MAD-X files, as was described in listing 3.43.

Chapter 4

Tests

As there are many different ways of testing an application of this sort, depending of what one wishes to figure out, one must first decide what question the test shall answer. This chapter will go through some different questions that need answering before one can decide whether the goals stated in this report have been met or not.

4.1 Unit tests

The first and foremost of these questions is of course: *"Does it work as intended?"*. The first way of testing this is to run the unit tests, as the unit tests of a system can be viewed as a specification of the different functions and modules. By running e.g.:

```
./mad -e -utest "mad.lang mad.lang.generator mad.lang.errors"
```

One will get a long list of modules and functions that have been tested and a finishing line:

```
Success : 100% - 239 / 239
```

This tells us that there are no test functions that failed, but it does not guarantee that the entire system works perfectly[1]. The reason for this is that the tests have been made by human beings, which means that they probably do not test all the different possibilities. It does tell us that these parts of the application works as intended for most use cases. To further test the application, one can do some black box testing, interpreting files that one knows are correct and check that the output is correct. When errors are found this way, they should be fixed and the unit tests should be updated to be able to find the error.

One of the files used for this black box testing has been the file `longluafile`. This file contains lots of code from random Lua files found all around the web,

[1] The more mature the system gets, the more does 100% in the unit tests insinuate a fully working system. This is because unit tests shall be written every time a new bug has been found and fixed.

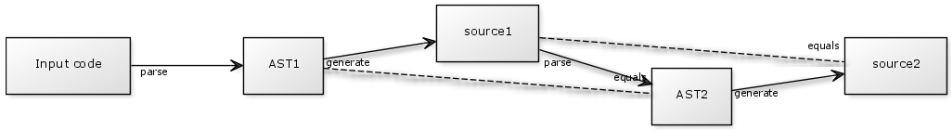


Figure 4.1: Description of an idempotent parser. To be idempotent, a parser needs to always create the same AST from the same input code and the generated code from a certain AST should be parsed to become the same AST. Thus, AST1 and AST2 should be equal, as well as source1 and source2.

mostly defining functions and tables, not running that much. It is 971 lines long and 31.1kB. The generated code has been checked by hand and found to be equivalent[2] to the input code.

Another necessary feature of a parser for it to be correct is that it has to be idempotent. As figure 4.1 shows, an idempotent compiler will be one where, after creating the initial AST from the input code, one can apply the generator→parser-step an arbitrary number of times and always end up with the same AST. This is tested for both MAD and Lua, and is correct. One can not test this for MAD-X, as there is no MAD-X-generator.

4.2 Speed tests

All tests in this chapter have been run on an Intel Core i5-3470 CPU with a 3.20GHz quad core processor. It has 7.8 GiB memory and runs 64-bit Ubuntu 12.04 LTS. All tests of MAD have been made with git HEAD[3] from the MAD git repository[8] and git HEAD from LuaJIT, using v2.1 compiled with Lua 5.2 compatibility on. All tests of LuaJIT language toolkit have been run with LuaJIT git HEAD, using v2.0, not supporting Lua 5.2. Unless otherwise noted, all tests have been run using a setup similar to listing 4.1, although with small changes to make the different applications work properly. If any files need to be created, e.g. because the input shall contain random numbers, this is done before the first line.

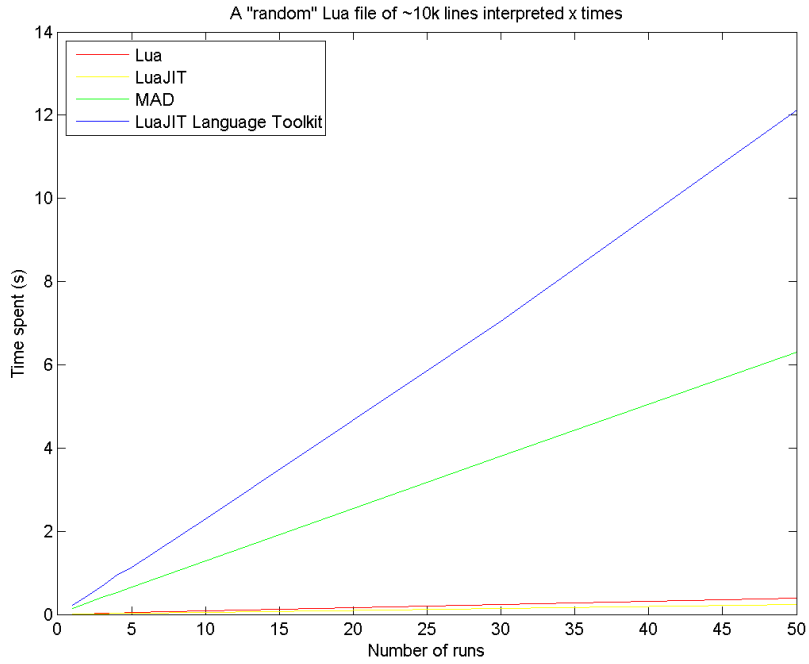
Listing 4.1: Setup for speed tests of different applications

```
local start = os.clock()
for i = 1, number of times to run do
  run application with command line arguments
end
local stop = os.clock()
total time = stop - start
```

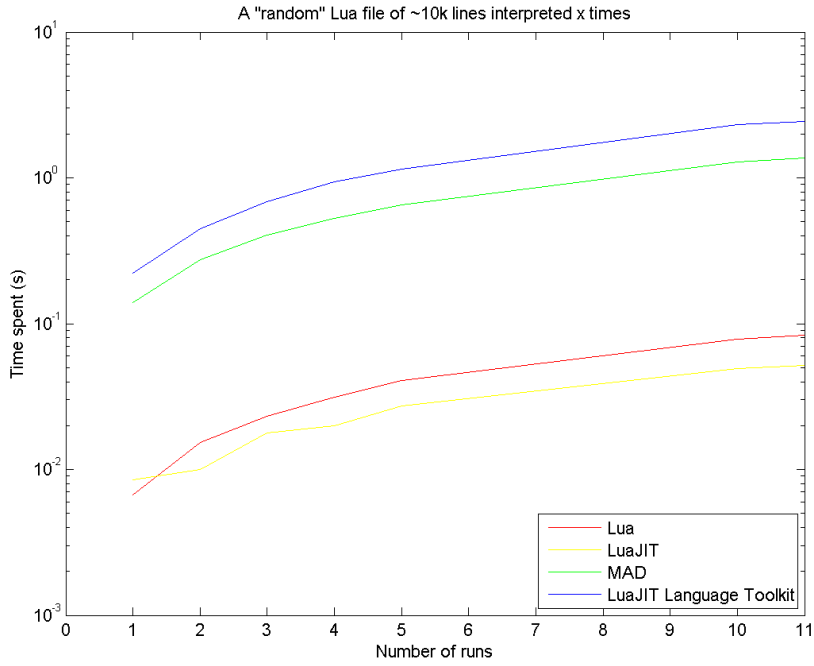
After testing idempotence and a large enough number of black box tests, one must assume that the code is correct and can continue to the next important

[2]Equivalent, not the same. Some changes will have to be made, e.g. removing whitespace and comments, but the code should do the same.

[3]Commit 84789bc2608a6430064baa5717f7307b3b275830



(a) Linear scale



(b) Logarithmic scale

Figure 4.2: A file containing ∼10k lines of assorted Lua code (314kB), run x times using Lua, LuaJIT, MAD and LuaJIT language toolkit. The y-axis shows the number of seconds used.

question: *"How does the application fare compared to similar applications?"*. This question is not an easy question to answer, as it is hard to compare this application to similar ones. Out of the similar applications that have been described earlier, all but one are made specifically to interpret a language different from Lua, even if only a slightly different version. The speed of Nyanga interpreting a Nyanga file is not necessarily equivalent to MAD interpreting a similar Lua file. It is possible to compare the speed of MADs Lua interpreter with LuaJIT language toolkits Lua interpreter. As these two projects both attempt to be simple and extendible applications to interpret languages using LuaJIT, the comparison is very relevant.

Figure 4.2 shows the result of running a file of assorted Lua code x times with Lua, LuaJIT, MAD and LuaJIT language toolkit. The file is the file `longluafile` repeated 10 times to get some length and an actual reading from Lua and LuaJIT. The file fits well for this kind of tests, as it does not contain that much code to be executed, meaning that it mostly tests the speed of the parser and generator, not as much the speed of what runs it, i.e. LuaJIT. The figure shows that Lua and LuaJIT are miles ahead of the two interpreters, as one would expect. This is of course not a problem, as the Lua interpreter is only a small part of the MAD application, not the main part of it, Lua interpretation is only a step on the way to the MAD interpreter. Another thing one might notice, is that MAD runs the file significantly faster than the LuaJIT language toolkit. As one can see from figure 4.3, MAD uses around 0.13 seconds to parse ∼10k lines, while Toolkit uses around 0.23 seconds. MAD also has a smaller standard deviation than Toolkit has. Another curious aspect of the figure is that MAD uses less and less time the more times it runs the runs the file, as the constants related to initializing files only are done once, while the toolkit uses steadily more and more time the more times it runs the file. This might point towards the fact that the toolkit has a lower constant or spends less time on shorter files. This can be further shown by looking at figure 4.4.

As figure 4.4 and figure 4.5 shows, the toolkit spends a lot less time to interpret an empty file[4]. While the toolkit runs on speeds sub-millisecond per run, MAD spends about 3.8 ms per run. This is mostly because no parts of the code actually tests whether it gets input or not. A chunk with no statements is allowed according to the grammar, which means that code will have to be generated for it, the errors module will need to support it (needing to generate tables, even if they are empty) and then it will need to be run.

From the above mentioned figures 4.4 and 4.2, one can read that the toolkit excels at smaller files, where MADs initialisation constant adds up to a large a part of the total time spent. On longer files, LPEG is seemingly faster than the toolkits home made parser, making MAD faster. This can be seen in figure 4.6. The figure shows the results of running a file containing x lines of `a = [random number]` 10 times using both the toolkit and MAD. As the figure shows, the toolkit spends a quite constant time per line parsed, while MAD starts out spending more than

[4]Because of Luas `os.clock` function only having a resolution of ms, the first runs of the toolkit are reported to spend 0 time. This is also why there are no error bars present, as one milliseconds variation on one millisecond is a huge variation, even though it's only there because of the resolution of os.clock.

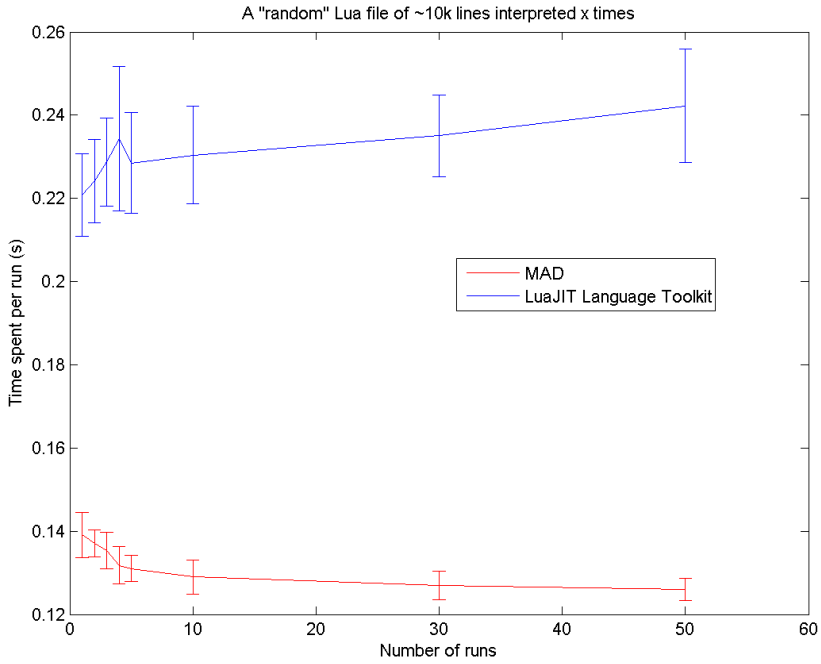


Figure 4.3: A file containing ∼10k lines of assorted Lua code (314kB), run x times using MAD and LuaJIT language toolkit. The y-axis shows the number of seconds used per run. Error bars show 1 standard deviation up and down.

twice the time per line when running only 100 assignment statements, but steadily lowering its seconds per line until it passes the toolkit around 1000 statements. The next part to notice is that MAD does not parse past 7000 lines, as LPEG fails with *"stack overflow (too many captures)"*. This is because all 7000+ statements are supposed to go in the same node, the chunk, and LPEG will therefore need to keep them in the stack until the entire chunk rule has been evaluated. One way of fixing this problem is to make the stack bigger, but no matter how large the stack is, there will be inflated test cases where one manages to overflow it. It is also bad form to fix all memory/stack related problems by simply adding more memory/stack size. One must simply decide what should be the maximum number of statements allowed and disregarding inflated test cases, the author still have not found any files that actually end up encountering this limit, no changes are therefore made to the stack size or to the grammar to fix this problem. Should this prove to be a problem in the future, there is another way of solving it as well which will be described briefly below.

If one needs to allow more than the stacks limit of statements, one can add a similar fix as the one made in MAD-X as shown in listing 3.43. Instead of calling `stmtnum` when the block is finished, it can be returned as a block. One will need

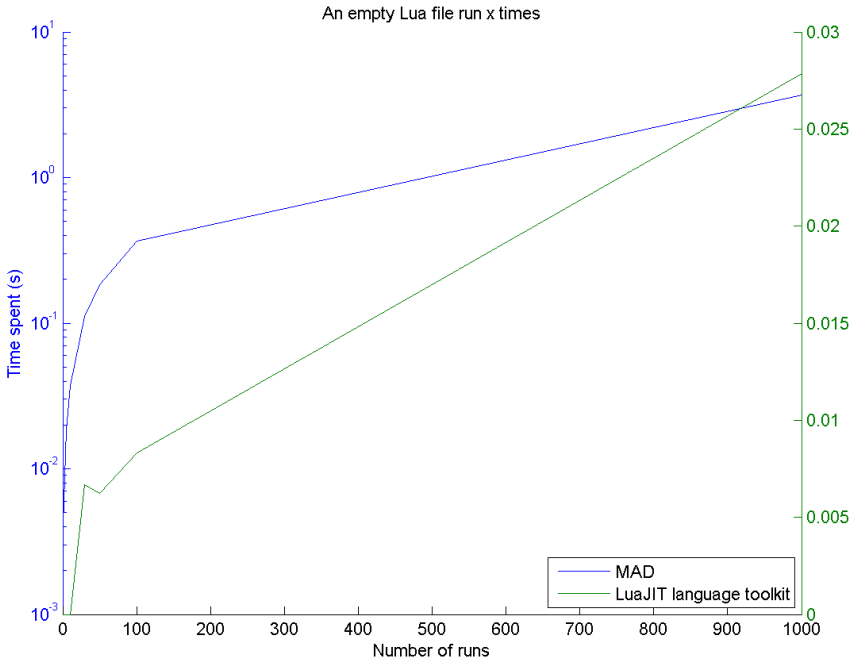


Figure 4.4: An empty Lua file run x times using MAD and LuaJIT language toolkit. The y-axis is the number of seconds spent. NB: MADs scale is logarithmic while the toolkits scale is linear.

to do some changes to the generators to allow blocks as statements, but it should work to fix this problem. By doing this, the allowed grammar of Lua is changed, even if it is a change that should be unconsequential. As the problem is viewed as a minor problem unlikely to show up, the change is not implemented. Enclosing parts of the code in do-end blocks will allow LuaJITs excellent garbage collector to garbage collect unnecessary values earlier on, thus speeding up the application and removing the problem with too many statements as well, as the do-end block will count as one statement.

The question about how MAD fares versus similar applications can be answered building on the tests discussed so far. The author was only able to find one application that was built for something similar that managed to parse Lua code, and these two applications were compared to each other in speed tests. It was found that LuaJIT language toolkit is faster on short files because of it having a small initialisation constant, while MAD, with its larger initialisation constant, was faster than the toolkit on larger files. MAD also has trouble parsing files containing over 7000 statements in the main chunk, while the toolkit manages this nicely. MAD does manage to parse arbitrarily large files if this constraint is met though, while the author had problems running files of over ∼1MB using the toolkit. The toolkit

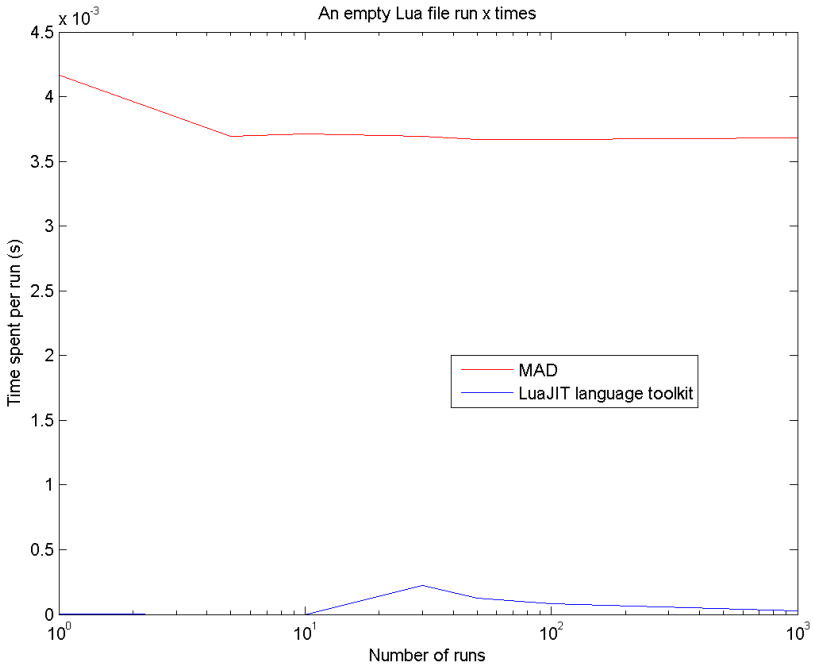


Figure 4.5: An empty Lua file run x times using MAD and LuaJIT language toolkit. The y-axis is the number of seconds spent per run.

also seems to be younger than MAD, which means that there are still bugs that appear while attempting to parse some random files the author tried to parse for some different tests. Since it is viewed as less important to be a bit more instantaneous in files that are interpreted almost instantaneously, than to be significantly faster[5] on long files, MAD is viewed as performing well compared to similar applications. The constraint on the number of statements is also deemed unimportant and a suggestion as to how one can fix it, should it prove to be significant, has been made.

"But how do the different interpreters work compared to the Lua one?", one might ask. As the only interpreter one can test compared to other applications is the Lua interpreter, it makes sense to ask whether the other interpreters are following suit. One interpreter it is easy to compare to Lua is the MAD interpreter, as MAD is a simple super set of Lua and should therefore be able to parse all Lua files. The MAD interpreter should also not be significantly slower than the Lua interpreter. This was tested by interpreting a file containing ∼100k lines of arbitrary Lua code (3.1MB) with two different endings, .mad and .lua. The results can be seen in figure 4.7 and figure 4.8. Figure 4.7 seemingly shows that Lua is

[5] 2x as fast in a file of 314kB (figure 4.2)

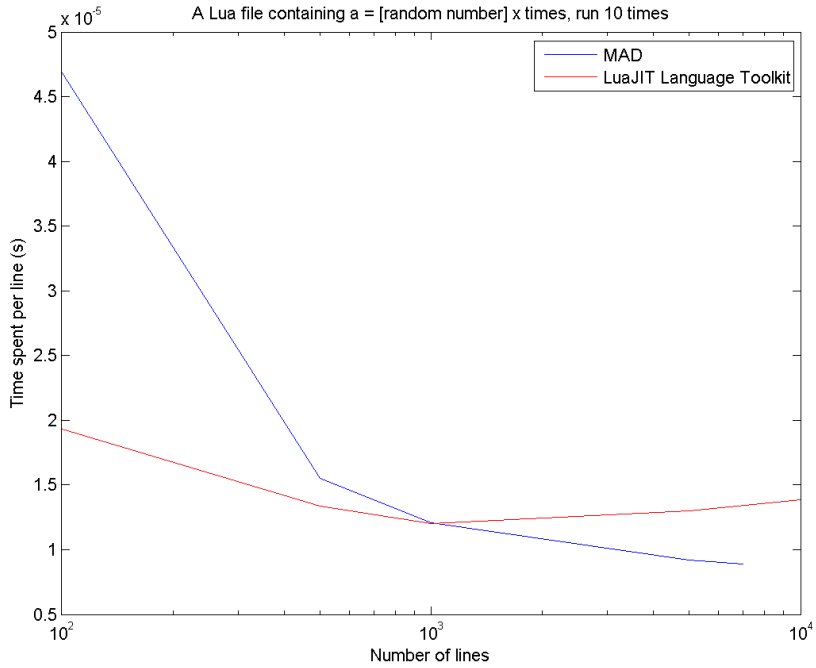


Figure 4.6: A Lua file containing x lines of `a = [random number]` run 10 times using MAD and LuaJIT language toolkit. The y-axis is how many seconds were used to parse each line. MAD gets stack overflow when parsing more than ∼7000 lines, while the toolkit manages far higher numbers.

slightly faster, as it shows the mean values of 10 runs, but as one can see from figure 4.8 which also shows the minimum and maximum values of the two different interpreters, their seconds per line-count is within the same margin of error.

The MAD language also contains support for C-style include statements whose test results are also shown in figure 4.7 and figure 4.8. The include statement is tested by testing using an intermediary file containing the line

```
@include "file/to/test.lua"
```

instead of the file `file/to/test.lua`. Using include should be slower than without, as the interpreter will need to read two files instead of one and run two parsers instead of just one. The question is rather how much slower it is. And as one can see from figure 4.8, the include tests seem to be slower than MAD and Lua, but within what one might view as the margin of error from the quite varying execution times. An explanation for why the difference is this small is that the file tested is quite big (3.1MB), meaning that the time it takes to parse that file far outweighs the extra time spent to open an extra file and initiate an extra parser. Figure 4.9 shows the results of parsing files of different length using the MAD interpreter

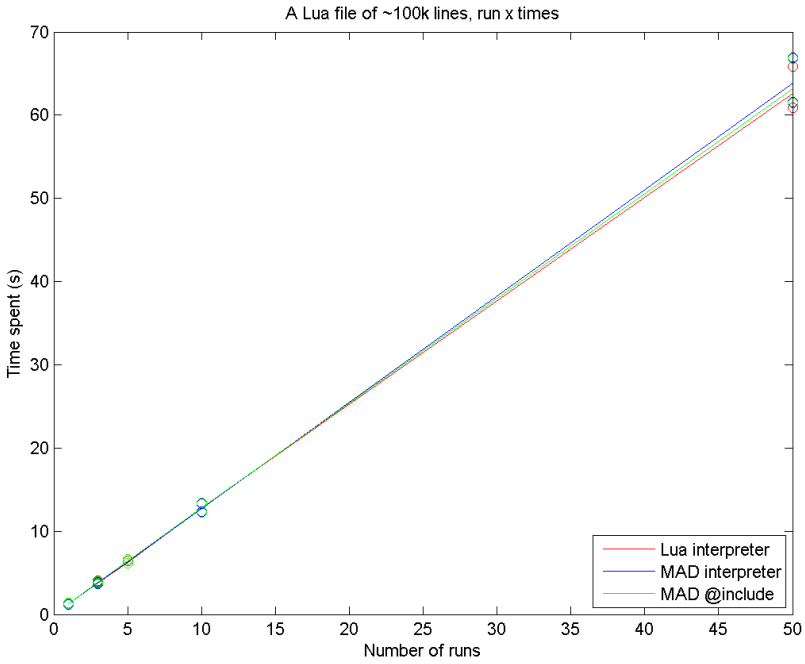


Figure 4.7: A Lua file of ∼100k lines (3.1MB) run x times using both the MAD and the Lua interpreter. y is the time spent in seconds. The circles show the maximum and minimum time spent for each test.

including the test file and by using the Lua interpreter. The figure clearly shows that the extra time spent is significant for smaller files. For the empty file it uses almost twice the time, not surprising as the application needs to do almost twice the work. The difference is still notable for the file consisting of 1k lines (31.4kB), but for the larger files, the difference is almost non existing. As include will most likely be used to shorten and simplify code, it is unlikely that one will include files larger than a couple of kilobytes, include statements will therefore slow the code down. The speed is still competitive though, as it is still significantly faster than LuaJIT language toolkit for the 31kB file.

To be able to decide whether the MAD implementation works, the speed of the lambda definition must be tested. There are two different ways of translating lambda functions implemented in the Lua generator. One can choose between these two by sending the command line argument `-lambdatable` to the application. If `-lambdatable` is set, functions nodes with `kind == lambda` will be translated to tables with their `__lambda`-key set, as described in the implementation chapter. If `-lambdatable` is not set, they will be translated to standard anonymous functions. The difference between these two is that the lambdatables may be evaluated without using parentheses in some some cases, as described in the implementa-

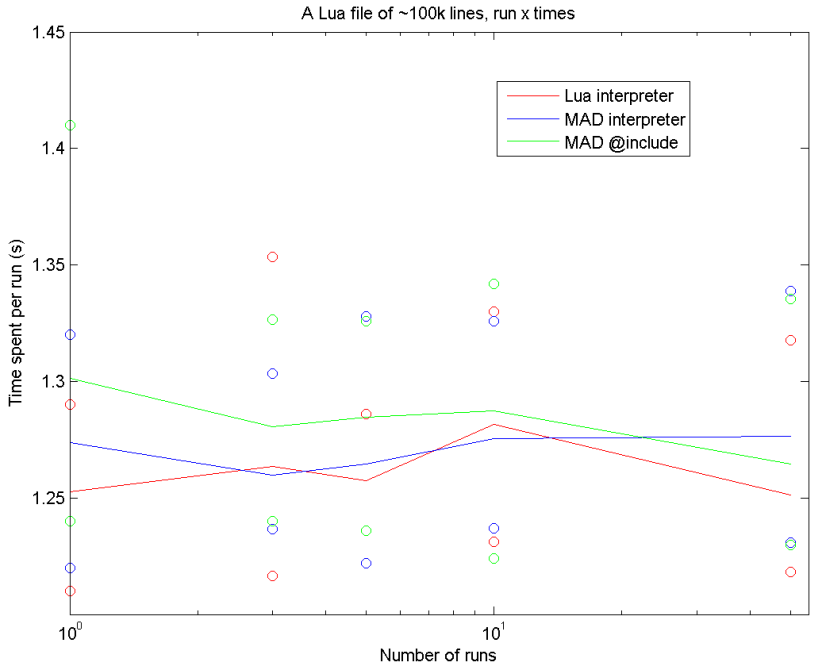


Figure 4.8: A Lua file of ~100k lines (3.1MB) run x times using both the MAD and the Lua interpreter. y is the time spent per run in seconds. The circles show the maximum and minimum time spent for each test.

tion chapter. This is implemented using metamethods for all the operators, which means that using this does not do anything with the speed of parts not using lambda. However, the math, table and string-libraries are overloaded completely, meaning that every time one of their functions[6] is called, they will need to check if their arguments are lamdbatables or not. This might slow down the code and will therefore need to be tested.

To test this, the code in listing 4.2 is used. The code creates a lambda function without arguments that is supposed to return the value of the global variable `a`. The three assignments are made to test three different usages of the function: Regular call, explicit call with a function that is not overloaded, and explicit/deferred call with a function that is overloaded for lambda.

Listing 4.2: Code used to test the speed of different usages of lambda

```
local l = \ a
a = 1
--Code below is repeated x times
```

[6]Except for a few chosen ones where deferred evaluation is impossible. A list is included in the implementation chapter and in the lambdas documentation.

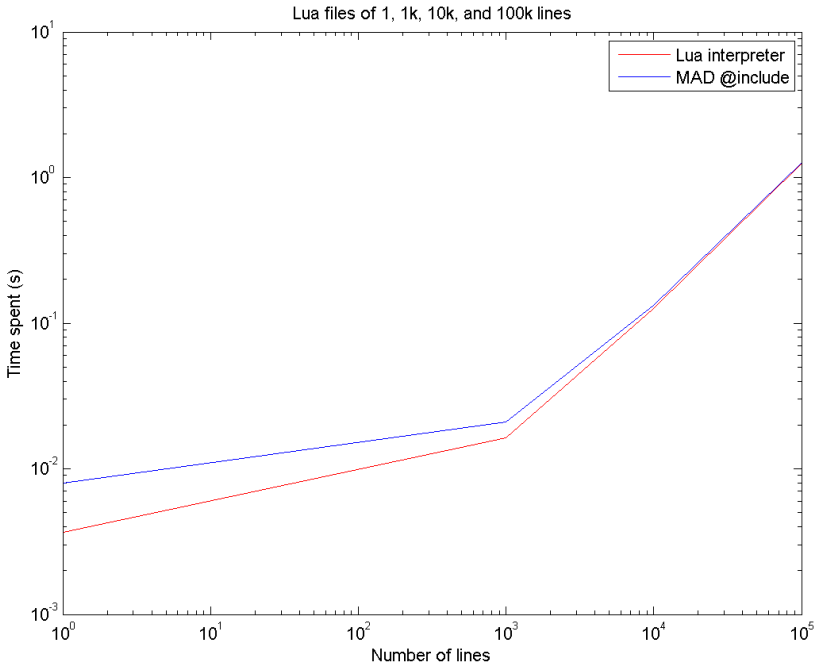


Figure 4.9: 4 files containing 0, 1k, 10k, and 100k lines of code (0B, 31.4kB, 313.9kB, 3.1MB) run by the Lua interpreter and the MAD interpreter trough a file including the actual test file through the `@include` statement. x is number of lines, y is time spent.

```
b = l()
b = math.max(l(), 0, 5)
b = math.sin(l())
--end repeated code
```

Figure 4.10 shows the speed of running the code in listing 4.2 in three different ways. The three different ways are tested with code of different length, repeating the three last lines of the code x times. The different ways of using lambda are:

1. Exchanging the last line of the code in listing 4.2 with `b = math.sin(l)` to have deferred evaluation of `l`. The code is run with the argument `-lambdatable` on, to make the generator translate the code to a lambda, not a regular function.

2. Explicitly calling `l` in all three cases, but calling the application with `-lambdatable` to make the generator translate the code to a lambda function.

3. Explicitly calling `l` in all three cases, but calling the application without `-lambdatable` to make `l` be translated to an anonymous function instead.

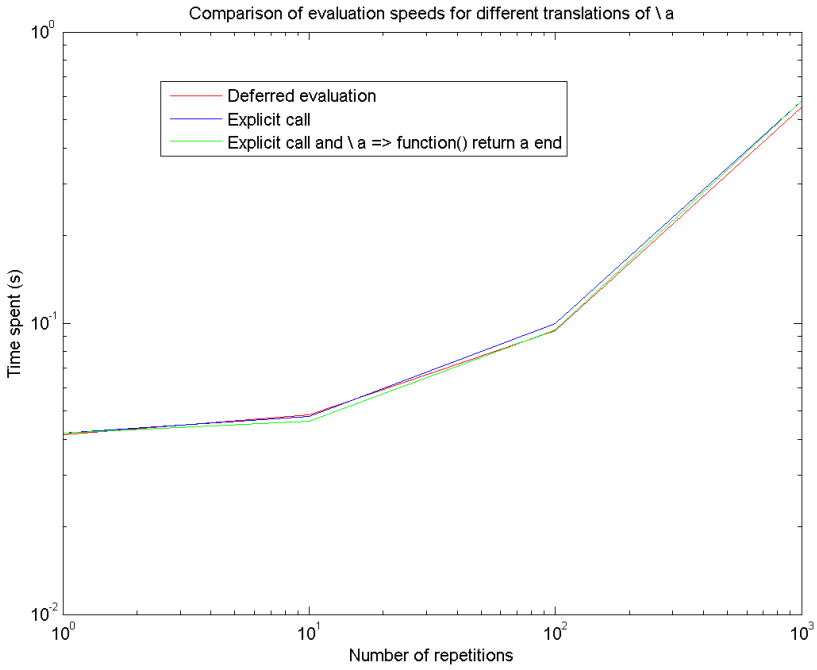


Figure 4.10: Comparison of speed for different usages and translations of the lambda function `\a`. A file containing the definition of a lambda function `l = \a` and `a = 1`, immediately followed by x repetitions of `b = l() b = math.max(l(), 0, 5) b = math.sin(l())` as seen in listing 4.2. Deferred evaluation means that the parentheses are omitted in the sine function.

> This also makes the math libraries not become overloaded, thus perhaps becoming faster.

As figure 4.10 shows, there are no discernible speed differences between the three ways of calling and translating the lambda. In the case of the longest file, the test that did not overload the math library is in fact the slowest, albeit with a tiny fraction and well within the margins of error. From this, one can conclude that overloading the math, table and string libraries does not significantly slow down the code and can therefore be kept.

Chapter 5

Discussion

Several design decisions were made during the implementation of the application, some were good and some might have been less fortunate. This chapter will discuss the different choices made, try to find the positive and negative consequences the choices had, and discuss whether the choice was a good one, or if something should be done differently.

5.1 Initial design decisions

Before starting the implementation of this project, some initial decisions were made as to how the application should be structured.

5.1.1 Using LPEG

The most important of these decisions was that the application should have more or less the same structure as Nyanga, as shown in figure 3.1. This means that LPEG should be used instead of a more traditional lexer/parser approach. As it is also shown when describing the LuaJIT Language Toolkit, it is possible to implement the Nyanga structure using a lexer/parser approach instead of using LPEG, which raises the question of what is the best solution.

As the tests from the previous chapter shows, the two implementations are competing on similar terms, the toolkit being the fastest on short files while MAD outperforms it on larger files. All in all, these two different implementations both seem to work and neither is particularly bad at what they are made to do. The main difference between them then, is that MAD uses LPEGs re module for the grammar, while the toolkit uses a handwritten lexer and parser. The positive effects of writing ones own lexers and parsers is that one will be in total control of what happens, while the positive effects of using LPEG is that it is unnecessary to be in control of what happens. The author believes that the re module provides a lower initial step to understand and start implementing ones own grammars. What makes MAD slower on the short files is its initialisation cost, which is introduced

by the bulk of the application, not as much by LPEG[1]. Therefore using LPEG and re was a good choice, as building the AST from the re-grammar is simpler than writing one in a handwritten parser.

5.1.2 Not using LpegLJ

After the choice had been made to use LPEG, the choice of moving to LpegLJ opened. LpegLJ was briefly introduced when discussing left recursion. The main difference between LPEG and LpegLJ is that LpegLJ uses the suggestion made by [34] to allow definition of grammars with left recursion, thus keeping the necessary changes to the grammar at an absolute minimum. The other difference is that LpegLJ is a pure LuaJIT + FFI implementation of LPEG, made to potentially be faster. As LPEG is a lot more mature than LpegLJ, LpegLJ is about 2.5 times slower than LPEG run with LuaJIT, according to LpegLJs own tests. This slowness is in itself a good reason not to use it, just as it was when the library was considered, and when it is possible to easily remove left recursion from the grammar, using LpegLJ is completely unnecessary. The conclusion is therefore the same that was reached when LpegLJ was first discussed: If a more stable solution, steadily outperforming regular Lua, is introduced, it will be a small change to move from LPEG to LpegLJ. This does not seem to be the case in the immediate future however, as there has been no changes to LpegLJs git repository the last two months, except for an update to its TODO file.

5.2 Changes made to the grammar

As LPEG does not support left recursion, the grammar needed to be changed. This section will look into the changes that were made and discuss whether they could have been handled differently.

5.2.1 Splitting the variable into definition and expression

Listing 3.7 shows the final implemented solution to the problem with left recursion in variables/function calls. The solution creates different rules for defining and using variables, as well as a stand-alone rule for using function calls as statements. The positive effects of this is that one will save a lot of time parsing variables, as these would need to be parsed twice when following the rules from listing 3.6. The potential negative effects of the fix is that one might have broken the grammar. By using the grammar defined by Lua, one is secure that one will have a proper grammar that allow only what it is supposed to allow. Following the rules defined by [40], one can also know that the grammar is equivalent. When the grammar is split as was done in the case of variable and function call, the grammar might become different, thus disallowing something that should work, or allowing something that should not work. Assuming that the rules in listing 3.7 are equivalent to the ones coming from the original grammar, the author concludes that the chosen solution

[1] Except for the one time compilation cost to compile the re-grammars to LPEG.

was a good choice, as it is a lot faster than the solution made only by using the algorithms described by [40].

There are two ways of showing that the implemented rules are equivalent to the original rules. The first one is to simply look at the grammar. Listing 3.6 clearly show that the only difference between a variable and a function call is that a function call must end with a callsuffix, while a variable must end without one. Therefore, the variable definition and the function call statements from listing 3.7 are equivalent to the ones from listing 3.6. The use of variables and function calls in expressions are equivalent as well, as the original rule simply states that one may have a variable or a function call, while the new rule says that one can have a base, which has the same structure for both variables and function calls, and an optional call suffix in the end, which will make it into a function call instead of a variable.

The second way of testing if the new implementation works is to test it thoroughly, which is greatly simplified by the unit testing framework implemented earlier. By running the unit tests of `mad.lang.lua.parser` one can see that both luas defs and grammar is thoroughly tested, especially the varexp rule, which is the cornerstone of this rule set. Testing can of course not prove that something is 100% the same or not, but it can give a clear pointer towards it, which this does.

5.2.2 Not having associativity

As listing 3.8, the rule set that properly describes the grammar with both precedence and associativity, contains left recursion and therefore can not be compiled by LPEG, it was decided to use the rule set from listing 3.10. This rule set does not make any assumptions about the associativity, and is helped by the AST which is built with all similar operators in the same node, thus creating a node free of associativity. The application never does anything with the expressions until it reaches the generator which generates Lua code to be run and when this is reached the expressions will be generated in the same order that they came in. All in all, this ends up forcing all operators to have the same associativity as they have in Lua, or one will have to process the node later to properly balance it, as Nyanga does. With the current size of the application and the fact that it still does not use a transformer, this has not been a problem. Operators follow the same rules for most programming languages, having the same associativity in all of them, which means that it is not a problem to use Luas associativity. However, should one wish to do any transformation on the tree, one will probably need to balance the nodes correctly. To fix this, one can either use the solution from listing 3.8, which is not doable without LpegLJ, which is slow, or by waiting for LPEG to implement support for left recursive grammars, which is in their pipeline, but with no deadline. A more sound solution is to do as Nyanga does and balance the tree after building it, to balance it properly. It is possible to create a transformer that reads the AST as it is now as well, without associativity, meaning that this is problem is an unimportant one.

In conclusion, not having associativity can potentially create problems when implementing a transformer to trim the AST, or if one for some reason wishes to

have an operator with a different associativity than Lua, but there are simple ways of fixing the problems should they arise. The chance of the problems appearing are assumed to be small, and no work has been put into fixing them before it is proven that they do appear.

5.2.3 How to check if two grammars are similar

The changes that have been made to Luas grammar have not all followed the algorithms proposed by [40] and the grammar that is being used in the application is therefore not proven to be the same as the original grammar, as has been discussed above. As described in [41], it is impossible to prove if two distinct grammars represent the same language. This means that it is impossible to prove that the language parsed by MADs Lua interpreter is in fact Lua, or if it is a language similar to Lua. It has been discussed earlier how one can show that the grammar describes a language that acts as Lua within all reasonable constraints. Since this application is made to work, it is not that important if it is 100% correct, both because eventual errors found on a later stage can be fixed easily and because as long as it works for all cases that are needed, it works as intended.

5.3 The final product

This section will discuss the questions that arise during testing, as well as the viability of the entire product. One of the questions that arise when discussing this, is how can one check the viability of a product of this scope. The final product has several different applications, and one will therefore need to discuss the viability of all the different applications to be able to come to a conclusion about whether the requirements have been met fully, partly, or not at all. The different applications will therefore be discussed below.

5.3.1 Lua interpreter

The main goal of this project has been to create a Lua interpreter to be run using LuaJIT. As is shown during the implementation and test chapters, the Lua interpreter is implemented and working. Its performance can compete with other projects made with a similar goal in mind. Some questions have come up during testing and these will be discussed below.

Tests comparing MADs Lua interpreter to the LuaJIT language toolkits Lua interpreter shows that MAD is significantly slower on empty files, and does worse in general on short files. Is it acceptable that the interpreter is that much slower on empty files?

In a perfect world, the interpreter should of course be the quickest in all aspects, but having a rather large initialization constant is less of a problem than having a large constant associated with the size of the file, thus making the parsing slow on larger files. If someone should use this application to interpret short files, they will still not notice a difference, as the files will still be interpreted at a speed that

can safely be called instantaneous by humans. If a file is parsed instantaneously by MAD or twice as instantaneously by the toolkit really makes no difference.

A problem arose with MAD not being able to parse chunks that contained more than 7000 statements at the top level. This is something that might become a problem, as there is no way of guaranteeing that users create files of a small enough size[2]. However, most Lua modules should be quite small, not spanning more than 100 lines as a general rule, and this constraint will therefore not be met in most cases. When one starts to implement sequences in MAD instead of in MAD-X, the barrier might be met though, which means that this might become a problem in the future. A suggestion as to how this can be changed is therefore made, and the author leaves it up to the MAD custodian if this should be implemented or not, depending on whether this proves to be a problem in the future or not.

5.3.2 Extensibility of the application

An important focus of the project has been to implement the interpreters in a way that is easily extendible. This is supported by the modular layout of the project, where one will only need to do changes in the file `mad.lang.lua` to allow for a new language to be added to the list of supported languages. The designer of the new language can then follow the examples from the mad, lua and madx folders and add grammar.lua containing the re-based grammar of the language, defs.lua containing the functions used to build the AST, and parser.lua, containing the parser constructor. The AST has a short and concise definition, containing few and generalised nodes, thus allowing a large range of potential languages to be implemented. It is also the authors belief that writing grammars in LPEGs re syntax is really simple, as long as the language in question has a context free grammar. If the language creator figures out that he needs to change something somewhere else in the files to make the language work, e.g. allowing a new 'kind' in a specific AST node, thus needing to change the Lua source generator, the unit testing framework makes it simple to check whether the change made any part of the application stop working.

5.3.3 MAD interpreter

The MAD language is a strict superset of Lua, with only two additions, which means that the MAD interpreter shall behave quite like the Lua interpreter both in usage and speed. This has been shown to be correct, both when parsing a Lua file and when parsing a MAD file containing lambda functions. Using the include statement has been shown to be slightly slower than having all the code in a single file, but having a slight speed loss from inclusion is something one will have to live with, as opening two files will be slower than opening one. The speed loss is higher than what is expected simply from opening two files instead of one though, which means that there is something in the code that makes slows it down. This is assumed to be the table.unpack-function that is used when the

[2]Using the word size here might be a bit misleading, as the interpreter does handle arbitrarily large files, as long as they have less than 7000 statements in a single block.

included AST is added to the top level AST. Unpack is one of the functions that are not JIT compiled, thus making it slower than other functions. Should the AST be redesigned to allow blocks and chunks to have blocks as statements, this unpack can be removed as well, and the entire chunk can be inserted directly into the AST, potentially saving time. As long as this is not done, the author does not know how to speed the function up. The loss of speed is small enough to be viewed as insignificant and the author therefore believes that it is unproblematic and unnecessary to change the layout to remove this tiny speed bump. Should the allowed AST be redesigned to allow blocks as statements in blocks, the change one will have to make in the implementation of include is very small and should obviously be implemented, as every gain in speed is positive.

5.3.4 MAD-X interpreter

Implementing MAD-X posed a brand new challenge compared to the two previous languages. While Lua, and then also MAD, had a clear and concise context free grammar, MAD-X comes with no actual grammar in addition to being context dependent. The first thing that had to be done with MAD-X was to look through all the documentation made for it to figure out what was the allowed syntax. This was a difficult task, as all documentation was written by and for accelerator physicists, and was usually structured as a users manual telling users handy tricks for accelerator design, not as something that shall teach someone a new language. After some time, some kind of a grammar was made and implemented and a new problem arose. When implementing the Lua grammar, it was possible to find files that were guaranteed to be correct and check if the parser could parse them, but because the current MAD-X parser is built without an actual grammar, there are tons of things that it allows that should not be allowed. Every single error the implemented grammar reported, would therefore need to be checked in the documentation to find out if it should be allowed or not. When an error was found, it was still necessary to hear with the users to see if this was yet another case of "known bug is feature", which some super user needed for his specific modules implementation to work. Working like this is time consuming and the grammar gets so full of exceptions that it ends up being as prone to errors and bugs as MAD-X. It was therefore decided that the only undocumented exception that would be allowed in the grammar would be the madx-strings, as discussed in the implementation chapter. This decision helped keep the grammar readable and less error prone. The "features" that were removed are small enough to be overshadowed by the improvements that came for free, e.g. that boolean expressions can contain more than one and or or.

Writing the context free grammar for a context dependent language is another problem that has had to be overcome during this implementation. The implementation of the command statements as shown in listing 3.34 and listing 3.35 could have been solved by hard coding the different accepted tags and commands into the grammar, thus speeding up the parsing, as LPEG would handle a strict grammatical implementation better than the one that exists now. This would become a huge rule that would also be harder to maintain than the current function, both because

unseen errors easily creep into huge strings, but also because changes to MAD-Xs definition file would need to be inserted into the grammar, while the current implementation comes with a script that parses MAD-Xs def-file and updates the file used by the grammar. A slight speed loss is therefore acceptable compared to the range of potential errors and difficulty of maintaining that would be introduced with the fastest implementation.

5.3.5 Errors module

To have a module making sure that errors are called to the correct line and file is important when using a framework like this. If the user needs to do his debugging in the created Lua file anyway, he could just as well have used Lua instead of MAD or MAD-X. There are a few questions that need to be discussed about the implementation and usage of the errors module, which will be discussed below.

The modules backbone is the linemap, a table mapping all lines from all output files to their corresponding file and line. At first glance, this could become very memory heavy, as one will need an int and a string for each line in the outputted code. Because of how Lua is made, this is not the case. In Lua, there exists only one occurrence of a given string, all actual string objects are merely pointers to that string. Because of this, the linemap will only use a couple of bytes per line in the output code. This leads to that after parsing some thousand lines of code, the linemap will only take up about 10 kilobytes of memory, which is far below anything that could actually trouble LuaJITs 2GB memory limit.

Lua allows users to insert white space and comments almost everywhere, meaning that knowing which line to report the error to can be a difficult task. To report an error, the MAD implementation of the parsers will keep a variable that contains the current line the program has parsed to. This line is updated continuously and work well for expressions and statements that are on a single line. But because of LPEG being a top down parser, it will evaluate the functions corresponding to a rule at first once the entire rule has been validated. This means that the line that is inserted into the rules corresponding AST node will map to the last line in a statement, not the first. One could of course simply write this in the documentation, but an important part of the usability of an application like this is that it behaves as one would expect. Mapping errors to the last line of a statement when most other languages will map them to the first line, will be the opposite of working as expected. It will probably make users spend ages looking for errors in the next statement, before giving up and leaving MAD forever. This problem is partially fixed, making a specific output line always map to the first lowest line number in the output lines. This will fix the problem for most cases, e.g. if the statement failing was the one below:

```
for i = 1, 1 do
  a = i
end
```

The line corresponding to the for statement would be 3, thus making the generator initially map output line 1 to input line 3, but when the expression that also

belongs at line 1 appears, the linemap would be updated, mapping output line 1 to input line 1. There may still be occurrences of errors that end up mapped to the end of the statement, but those bugs will be far enough apart for them to be insignificant. "Everyone" knows that the position the errors are mapped to might be wrong anyway, even when using more mature languages[3], and one will therefore need to look in the errors surroundings to find its actual position. Having an error sometimes called to the wrong position is therefore still within the term "works as expected". If this is deemed inappropriate some time in the future, this can be fixed by using Nyangas version instead. Nyanga captures the position in the file and parses the file to count the number of line changes that have happened before the position is reached. This will slow down the AST creation significantly though, and the current solution with a few errors mapped incorrectly is therefore a better alternative.

5.3.6 Unit testing framework

For the users to write unit tests, the unit testing frameworks API needs to be simple and easily understandable. Using as few functions as possible makes that the case, as there are only 4 functions to learn how to use, with names that properly state what they do. Writing unit tests is therefore deemed to be easy to learn and do, especially since the users may look at the previously implemented code to see examples of how it should be done. The users do not have to bother with keeping track of any statistics, as utest will do it for them. Had the project followed Lunits implementation more closely, having several specialised test functions instead of 4 general ones, it would be harder to write the tests, as the users would need to learn more functions, even if the functions are really similar as these ones are. Allowing the users to call their test functions whatever they like, as long as they are kept in the modules test table, will allow the user to give the functions helpful names. Using descriptive names is one of the tips given by [21].

5.4 Using the application

Most of the issues that have been discussed above are elements that the end user does not care about, as long as they work to a reasonable extent. What is important to the end user is that the application easy to use and that it behaves as one would expect it to behave and that there are as few special cases as possible, as these are guaranteed to be forgotten.

The application does not contain many of these special cases, but there are two significant ones that the users may encounter, both of which have to do with the command line options given to the application. The options can be seen in Appendix B: Mad-e command line arguments. The options in question are that the `-profiler` option needs to be the last option given before the files, and that one can not give the option `-utest filename` without having something following it. Utest acts this way because there is no way for the module handling options to

[3]MAD-X users most of all, as they have lived with it for 15 years.

know if the filename should be a filename that should be executed and utest should be run without arguments, or if one wishes to test the filename and not execute any files. This therefore defaults to executing the file instead of testing it, as is also stated in the documentation. The profiler needs to be the last because of how its arguments are parsed.

The profiler should not be a major problem for the users, as profiling will mainly be used by MADs developers, not by the end users. It is also kept separate in the documentation to further show that it needs to be the last. The utest-option can prove to be a problem, but as it is impossible to know which behaviour is desired by the user, a default behaviour has to exist and defaulting to executing the file was chosen. This is clearly stated in the documentation, but it is not as one would expect. There is not a lot to do about it unfortunately, and for the special case to appear, the user has to only test a single module, not have any other flags after the utest-flag and not run any files, which is a rare event.

Chapter 6

Conclusion

To aid in the ongoing Methodical Accelerator Design project at CERN, a new MAD application has been implemented to replace the current MAD-X language, as MAD-X is too weak to implement the new generation of accelerators that CERN will build in the future, as well as being unacceptably bug ridden and time consuming to maintain.

Different ways of implementing an extendible interpreter have been discussed, concluding that the best approach is to use the LPEG framework to create a grammar straight from the EBNF grammar of the language in question. It is further discussed how to properly design an Abstract Syntax Tree to be able to represent as wide a range of languages as possible and a definition for a generic AST has been created. A parser based on LPEG has been implemented for the Lua language, translating Lua code to the AST previously defined. A source code generator translating nodes with the ASTs structure to Lua has been implemented.

The parser and generator have been combined to create an interpreter that reads Lua code and returns equivalent Lua code to be run using LuaJIT. The speed of this interpreter is tested and shown to be fast enough to compete with other projects that have been made with a similar goal. Some potential problems the implementation can have have been discussed and solutions to the different problems, should they arise, have been suggested.

The three main unit testing frameworks for Lua have been reviewed, all of which were deemed unsuitable either because of them being unfinished, outdated, or both. Building on their strengths and learning from their weaknesses, a new unit testing framework has been implemented. The implemented framework is easy to use and output simple and readable output. The output provides the user a report of how many tests have succeeded as well as information about the test(s) that failed, to help pinpoint the error(s). The API to write the test functions is simple and minimalistic, providing users with a quick and concise way of writing tests and testing their modules. The framework makes few assumptions about the system it is being used to test and may therefore, with a few minor adjustments, also be used in projects with no connection to the MAD project, even if it is initially designed for MAD.

How the interpreter may correctly report syntax- and run time-errors, keeping the rate of incorrectly reported errors as low as possible, while still maintaining an adequate speed, have been discussed and a solution has been implemented. The proposed solution to further lower the rate of misreporting has been discussed and deemed to be too slow and therefore not worth implementing. The implemented solution correctly reports an adequate number of errors for it to be well within what could be called acceptable behaviour.

Different approaches to implementing interactive mode in a language framework like MAD have been discussed and a solution has been implemented. This solution uses the fact that the parsers manage to differentiate between syntax errors that are unfinished statements and those that are just plain wrong. The implemented solution works as Lua and LuaJITs interactive mode. The only difference in their workings is that in MAD, one may continue in the same environment after writing a line that ended in a syntax error, while Lua and LuaJIT will force a restart when syntax errors are met. This is a clear improvement and may save users from both wasted time and loads of frustration.

The extensibility of the system is shown by implementing a new language, MAD, that is a strict superset of Lua. The additions to the language are C-style include statements and lambda functions that can be lazily evaluated. The implementation is tested compared to the Lua interpreter and found to be interpreting Lua code at the same speed. The MAD-specific parts are tested as well, and are shown to not be slowing the interpreter down more than one must accept.

To further aid the new application in becoming used, a MAD-X interpreter has been implemented. The implementation is made by translating the MAD-X code to Lua code specified by MADs Lua implementation made by the MAD custodian in parallel with this project. The implemented interpreter can translate MAD-X code to Lua code and is checked by hand to translate the code correctly. The code needed to run the Lua code has still not been fully implemented, and doing proper tests on this implementation can not be done. When the code is complete, the interpreter will work as it should.

The implemented solution works as intended and has been shown to fulfil its goal of being easy to extend. The extensions work as they should, making it possible to interpret both files made in the new MAD language and, once the physics behind it is implemented, in the old standard language MAD-X.

The final product runs on Linux and Mac-OS, and solutions for porting it to Windows have been suggested. It contains a stand alone framework for unit testing and an easily extendible framework for implementing interpreters. The application can interpret Lua, MAD, and MAD-X files, running them at speeds comparable to equivalent projects.

Appendix A

How to install MAD

Installing LuaJIT 2.1 with 5.2 compatibility

The application uses LuaJIT 2.1 and 5.2 compatibility, which needs to be installed. The easiest way to install it on Linux is explained below and it should be similar on mac.

1. Run the command

   ```
   git clone http://luajit.org/git/luajit-2.0.git
   ```

2. Change to version 2.1 by running the following command in the luajit-folder just created

   ```
   git checkout v2.1
   ```

3. Open the file `luajit-2.0/src/Makefile` and exchange the line

   #XCFLAGS+= –DLUAJIT_ENABLE_LUA52COMPAT

   with

   XCFLAGS+= –DLUAJIT_ENABLE_LUA52COMPAT

4. Make and install the program

   ```
   sudo make & make install
   ```

Install MAD

1. Download MAD by cloning the git repository

   ```
   git clone http://github.com/ldeniau/mad
   ```

2. Make LPEG

make linux [LUADIR=/path/to/luajit/include]

or

make macosx [LUADIR=/path/to/luajit/include]

LUADIR defaults to `usr/local/include/luajit-2.1`. If you get an error `fatal error: lua.h: No such file or directory`, your LUADIR is wrong.

Your copy of MAD should now be ready to run!

Appendix B

Mad-e command line arguments

One can run the mad-e script with several command line arguments. The different flags and their meaning will be discussed here.

Mad-e is run by:

```
./mad–e {args} [-profiler profilerArgs] {filename}
```

The filenames are the files that are to be run in batch mode.

`-profiler "profilerArgs"` runs the application with the LuaJIT 2.1-profiler. The accepted profilerArgs are the same arguments as would be allowed if one were to run LuaJIT with the profiler manually, as described in `http://repo.or.cz/w/luajit-2.0.git/blob_plain/v2.1:/doc/ext\textunderscoreprofiler.html`. Ex: `./mad-e -profiler 3si4m1 file/name.lua` will run name.lua while sampling 3 levels deep in 4ms intervals and, when finished with the program, shows a split view of the CPU consuming functions and their callers with a 1% threshold.

`-utest "space separated list of module names"` starts unit tests of the modules given. If no modules are given, unit tests will be run on all loaded modules. Module names must be written on require-form, not on file path-form. I.e. file/name.lua can be unit tested by running `-utest file.name`. Potential pitfall: If only testing a single file and not running any files, -utest can not be the last argument, as the module name will be interpreted as a file name to be run and utest will be interpreted as coming with no arguments.

`-dump "to_be_dumped"` dumps the AST or source code of the parsed files. `to_be_dumped` can be ast, mad, or lua.

`-interactive` or `-i` starts interactive mode once all files have been run.

`-benchmark "list of names"` - looks in mad.benchmark after a file with name and runs it. Not really used, but the possibility is still in.

Appendix C

MAD-X example sequence

The definition of SPS, as shown by [19].

Listing C.1: Definition of the 8km long SPS accelerator

```
QF:QUADRUPOLE,...; // focusing quadrupole
QD:QUADRUPOLE,...; // defocusing quadrupole
B1:RBEND,...; // bending magnet of type 1
B2:RBEND,...; // bending magnet of type 2
DS:DRIFT,...; // short drift space
DM:DRIFT,...; // drift space replacing two bends
DL:DRIFT,...; // long drift space
The SPS machine is represented by the lines
SPS: LINE=(6*SUPER);
SUPER: LINE=(7*P44,INSERT,7*P44);
INSERT:LINE=(P24,2*P00,P42);
P00: LINE=(QF,DL,QD,DL);
P24: LINE=(QF,DM,2*B2,DS,PD);
P42: LINE=(PF,QD,2*B2,DM,DS);
P44: LINE=(PF,PD);
PD: LINE=(QD,2*B2,2*B1,DS);
PF: LINE=(QF,2*B1,2*B2,DS);
```

Appendix D

Lua AST

`Chunk:`

ast_id 'chunk'

block block_stmt

`Block:`

ast_id 'block_stmt'

... list of statements

`Break:`

ast_id 'break_stmt'

`Goto:`

ast_id 'goto_stmt'

name 'name'

`Label:`

ast_id 'label_stmt'

name 'name'

`While:`

ast_id 'while_stmt'

expr 'expr'

block 'block_stmt'

`Repeat:`

ast_id 'repeat_stmt'

expr 'expr'

block 'block_stmt'

`For:`

ast_id 'for_stmt'

name 'name'

first 'expr'

last 'expr'

step -optional- 'expr'

block 'block_stmt'

`Generic for:`

ast_id 'genfor_stmt'

name 'name'

expr list of 'expr'

block 'block_stmt'

`If:`

ast_id 'if_stmt'

... list of 'expr' or 'block_stmt'

`Return:`

ast_id 'ret_stmt'

... list of 'expr'

`Assignment:`

ast_id 'assign'

kind nil or 'local'

lhs

rhs

`Expression:`

ast_id 'expr'

... list of operator or 'expr'

`Table access:`

ast_id 'tblaccess'

kind '.' or nil

lhs 'expr'

rhs 'expr'

`Function call:`

ast_id 'funcall'

kind ':' or nil

name 'expr'

arg list of 'expr'

`Group expression:`

ast_id 'grpexpr'

expr 'expr'

`Function definition:`

ast_id 'fundef'

kind 'local' or nil

name 'expr'

selfname 'name'

param list of 'name' or '...'

block 'block_stmt'

`Table definition:`

ast_id 'tbldef'

... list of 'tblfld'

`Table field:`

ast_id 'tblfld'

kind 'expr' or 'name' or nil

key 'expr'

value 'expr'

`Name:`

ast_id 'name'

name string

`Literal:`

ast_id 'literal'

value string

Appendix E

Lua Grammar

```
chunk = block

block = {stat} [retstat]

stat =  ‘;’ |
 varlist ‘=’ explist |
 functioncall |
 label |
 break |
 goto Name |
 do block end |
 while exp do block end |
 repeat block until exp |
 if exp then block {elseif exp then block} [else block] end |
 for Name ‘=’ exp ‘,’ exp [‘,’ exp] do block end |
 for namelist in explist do block end |
 function funcname funcbody |
 local function Name funcbody |
 local namelist [‘=’ explist]

retstat = return [explist] [‘;’]

label = ‘::’ Name ‘::’

funcname = Name {‘.’ Name} [‘:’ Name]

varlist = var {‘,’ var}

var =  Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name

namelist = Name {‘,’ Name}
```

```
explist = exp {‘,’ exp}

exp =  nil | false | true | Number | String | ‘...’ | functiondef |
 prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp = var | functioncall | ‘(’ exp ‘)’

functioncall =  prefixexp args | prefixexp ‘:’ Name args

args =  ‘(’ [explist] ‘)’ | tableconstructor | String

functiondef = function funcbody

funcbody = ‘(’ [parlist] ‘)’ block end

parlist = namelist [‘,’ ‘...’] | ‘...’

tableconstructor = ‘{’ [fieldlist] ‘}’

fieldlist = field {fieldsep field} [fieldsep]

field = ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp

fieldsep = ‘,’ | ‘;’

binop = ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘%’ | ‘..’ |
 ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ |
 and | or

unop = ‘-’ | not | ‘#’
```

References

[1] G. et al. Aad. "Observation of a new particle in the search for the Standard Model Higgs boson with the {ATLAS} detector at the {LHC}". In: *Physics Letters B* 716.1 (2012), pp. 1 –29. ISSN: 0370-2693. DOI: http://dx.doi.org/10.1016/j.physletb.2012.08.020. URL: http://www.sciencedirect.com/science/article/pii/S037026931200857X.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[3] *Build Test Deploy Development Environment*. URL: http://users.skynet.be/adrias/Lua/ut/index.html (visited on 03/26/2014).

[4] S. et al. Chatrchyan. "Observation of a new boson at a mass of 125 GeV with the {CMS} experiment at the {LHC}". In: *Physics Letters B* 716.1 (2012), pp. 30 –61. ISSN: 0370-2693. DOI: http://dx.doi.org/10.1016/j.physletb.2012.08.021. URL: http://www.sciencedirect.com/science/article/pii/S0370269312008581.

[5] *Check*. URL: http://check.sourceforge.net/ (visited on 03/26/2014).

[6] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ACM SIGPLAN Notices*. ACM Press, 2000, pp. 268–279.

[7] Laurent Deniau. *MAD-X progress and future plans*. 2012. URL: https://cds.cern.ch/record/1483257?ln=en.

[8] Laurent Deniau and Martin Valen. *MAD Git Repository*. URL: https://github.com/ldeniau/mad.

[9] Zachary DeVito et al. "Terra: A Multi-stage Language for High-performance Computing". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 105–116. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462166. URL: http://doi.acm.org/10.1145/2491956.2462166.

[10] Patrick Donnely. *Getting Rid of Left Recursion in the Lua BNF*. URL: http://lua.2524044.n2.nabble.com/getting-rid-of-left-recursion-in-the-Lua-BNF-td5840402.html (visited on 03/26/2014).

[11] *Embedding scripts in executable - Mailing list.* URL: http://lua.2524044.n2.nabble.com/Embedding-scripts-in-executable-td6838157.html (visited on 03/26/2014).

[12] *Extension for LuaJIT - Mailing list.* URL: http://www.freelists.org/post/luajit/extensions-for-luajit (visited on 03/26/2014).

[13] *flex: The Fast Lexical Analyser.* URL: http://flex.sourceforge.net/ (visited on 03/26/2014).

[14] Bryan Ford. "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking". Master. Massachusetts Institute of Technology, 2002.

[15] Bryan Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 111–122. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964011. URL: http://doi.acm.org/10.1145/964001.964011.

[16] W. Herr and F. Schmidt. *A MADX Primer*. 2004. URL: http://madx.web.cern.ch/madx/doc/madx_primer.pdf0.

[17] *How to compile Lua scripts into a single executable, while still gaining the fast LuaJIT compiler? - stackoverflow.* URL: http://stackoverflow.com/questions/11317269/how-to-compile-lua-scripts-into-a-single-executable-while-still-gaining-the-fas (visited on 03/26/2014).

[18] Roberto Ierusalimschy. "A text pattern-matching tool based on Parsing Expression Grammars". In: *Software: Practice and Experience* 39.3 (2009), pp. 221–258. ISSN: 1097-024X. DOI: 10.1002/spe.892. URL: http://dx.doi.org/10.1002/spe.892.

[19] F. Christoph Iselin. *MAD9 user guide*. 2000. URL: http://mad9.web.cern.ch/mad9/doc/mad9_usrguide.pdf.

[20] *JUnit Git Repository.* URL: https://github.com/junit-team/junit/downloads (visited on 03/26/2014).

[21] Jesse Lorenz. *How to Write Good Unit Tests*. 2009. URL: http://wiki.developerforce.com/page/How_to_Write_Good_Unit_Tests (visited on 03/26/2014).

[22] *LPEG Home Page.* URL: http://www.inf.puc-rio.br/~roberto/lpeg/ (visited on 03/26/2014).

[23] *LPEG Recipies.* URL: http://lua-users.org/wiki/LpegRecipes (visited on 03/26/2014).

[24] *LpegLJ Git Repository.* URL: https://github.com/sacek/LPegLJ (visited on 03/26/2014).

[25] *Lua Home Page.* URL: www.lua.org (visited on 03/26/2014).

[26] *Lua Lanes Project Page.* URL: http://luaforge.net/projects/lanes/ (visited on 03/26/2014).

[27] *LuaJIT and large tables - Mailing list.* URL: http://lua-users.org/lists/lua-l/2010-03/msg00238.html (visited on 03/26/2014).

[28] *LuaJIT Language Toolkit Announcement Post.* URL: http://www.freelists.org/post/luajit/LuaJIT-Language-Toolkit (visited on 03/26/2014).

[29] *LuaJIT Language Toolkit Git Repository.* URL: https://github.com/franko/luajit-lang-toolkit (visited on 03/26/2014).

[30] *LuaUnit.* URL: http://www.gpgstudy.com/gpgiki/LuaUnit (visited on 03/26/2014).

[31] *lunit - Unit Testing Framework For Lua.* URL: http://www.nessie.de/mroth/lunit/ (visited on 03/26/2014).

[32] *Lunit Git Repository.* URL: http://repo.or.cz/w/lunit.git (visited on 03/26/2014).

[33] Sérgio Medeiros and Roberto Ierusalimschy. "A Parsing Machine for PEGs". In: *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS '08. Paphos, Cyprus: ACM, 2008, 2:1–2:12. ISBN: 978-1-60558-270-2. DOI: 10.1145/1408681.1408683. URL: http://doi.acm.org/10.1145/1408681.1408683.

[34] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. "Left recursion in Parsing Expression Grammars". In: *Science of Computer Programming* 0 (2014), pp. –. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2014.01.013. URL: http://www.sciencedirect.com/science/article/pii/S0167642314000288.

[35] *Moonscript Git Repository.* URL: https://github.com/leafo/moonscript (visited on 03/26/2014).

[36] *Moonscript Project Page.* URL: http://moonscript.org/ (visited on 03/26/2014).

[37] *Nyanga Git Repository.* URL: https://github.com/richardhundt/nyanga (visited on 10/01/2014).

[38] Eduardo Ochs. *Angels Egg Interactive.* URL: http://angg.twu.net/LUA/interactive.lua (visited on 03/26/2014).

[39] Mike Pall, ed. *LuaJIT Home Page.* URL: http://luajit.org/ (visited on 03/26/2014).

[40] James Power. "Notes on Formal Language Theory and Parsing". In: *National University of Ireland, Maynooth, Kildare* (2002).

[41] Michael Sipser. *Introduction to the Theory of Computation*. 2012.

[42] *Terra Project Page.* URL: http://terralang.org/ (visited on 03/26/2014).

[43] *TestNG.* URL: http://testng.org/doc/index.html (visited on 03/26/2014).

[44] *The GSL Shell Project.* URL: http://www.nongnu.org/gsl-shell/ (visited on 03/26/2014).

[45] *unittest - Unit Testing Framework.* URL: http://docs.python.org/2/library/unittest.html (visited on 03/26/2014).

[46] Niklaus Wirth. "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Commun. ACM* 20.11 (Nov. 1977), pp. 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883. URL: http://doi.acm.org/10.1145/359863.359883.

[47] Han Zhao. "Lua Script Packaging". In: *Lua Programming Gems*. Ed. by Luiz Henrique de Figueiredo, Waldemar Celes, and Roberto Ierusalimschy. AU@. Rio de Janeiro: Lua.org, 2008. ISBN: 978-85-90379-84-3. URL: http://opac.inria.fr/record=b1130035.