Universiteit Leiden

Master Computer Science

AMOOSE: A Domain Specific Language for the Astrophysical Multipurpose Software Environment

Name:	Miguel O. Blom
Student ID:	s1801104
Date:	14/06/2022
Specialisation:	Advanced Computing & Systems
1st supervisor:	Kristian F. D. Rietveld
2nd supervisor:	Harry A. G. Wijshoff
Ext. supervisor:	Simon F. Portegies Zwart

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Preface

To my family and friends.

We would like to thank Ashmara Wederfoort, Michel Kavermann and Moritz Sturm for participating in our interview and providing us with example code.

Abstract

The Astrophysical Multipurpose Software Environment (AMUSE) Python library offers a framework that allows the simulation of step-wise time-integrated evolution of astrophysical models. A Domain Specific Language (DSL) is desired for quicker development of these models and would allow for a lower learn-ability threshold for new users. In order to derive code in the target language in the AMUSE framework from a DSL as the source language, a transpiler is required that performs this translation. This thesis describes the process of incrementally creating the transpiler in question, which is to be equipped with innovative features. For one, multiple different variants of code in the target language will be derived from the source code, to expose the different choices that can be made during the translation. These different derivatives can then be tested against one another to show the best translation regarding an error-performance trade-off.

Contents

1 Introduction

Modeling and simulating astrophysical systems is fundamental to the study of computational astrophysics. The Astrophysical Multipurpose Software Environment (AMUSE) [23] is a framework written in Python. It provides structures to represent particle sets to store e.g. planetary systems and encapsulates community-developed codes that implement dynamics, such as gravity, by approximating the continuous behavior of astrophysical systems iteratively. When multiple community codes are used simultaneously, bridges [10, 24] manage the alternate application, taking into account different internal time-steps of each community code. Bridges have their own time-step that has to be set properly by the programmer, as its value corresponds to the performance-accuracy trade-off: larger time-steps results in fewer iterations, but discretizing continuous behavior with larger intervals leads to a higher error.

AMUSE provides the right tools for modeling and simulating astrophysical systems, but stemming from an imperative programming language, some drawbacks occur in using the framework. Experienced users find themselves writing large chunks of recurrent code structures that can not be solved by simply writing functions. Parameterizing these structures and generalizing them in a huge function would discard the important structure which gives insight into the workings of the function to the programmer. Namely, providing the semantics of a large body of code through positional arguments works for a few arguments, but for many positional arguments, this would impose a high cognitive load [16, 26]. The amount of information that is required to be learned about AMUSE makes novice users feel overwhelmed by the possibilities and constraints that come with it. As a consequence, they write sub-optimal code and have trouble picking appropriate community codes and configurations.

Both of these problems can be solved with the introduction of a Domain Specific Language (DSL) specific for AMUSE. This DSL, a language designed for a specific domain, would compile to the AMUSE framework, which is written in Python. A well-designed DSL helps researchers set up experiments in less time, as it abstracts away recurrent code structures, and at the same time lowers the barrier for novice users by providing a straightforward syntax avoiding Python particularities.

In this thesis, we introduce AMOOSE, a DSL designated for both novice and experienced AMUSE users. We will identify issues advanced users experience with AMUSE (such as recurrent code structures), by looking at a number of scripts that are considered "correct". To identify issues novice users have with the framework, we hold interviews with students that are working on a computational astrophysics course project in AMUSE. From the set of principles and requirements, we design the AMOOSE DSL.

To implement AMOOSE, we developed a transpiler, named ANTLER, that compiles AMOOSE to Python code in conjunction with the AMUSE framework. We describe the design and implementation of ANTLER. A key feature of this transpiler is the particular intermediate representation, which allows us to compile to another source language. AMOOSE becomes an enabler for the introduction of new functionality, such as the calibration of parameter values, selection of community codes, and making informed decisions for the user during translation. We end this thesis

with an experiment where we compare the number of tokens that are required to write equivalent AMOOSE and AMUSE scripts.
This thesis is structured as follows: In Section 2, we will discuss related work. We will introduce some important background knowledge about programming languages and compilation in Section 3, along with a description of important notions about AMUSE, as we need to first understand the framework for which we will be generating code. In Section 4 we explain our general approach to identifying requirements and designing the DSL, named AMOOSE. Section 5 goes specifically into the designing of the AMOOSE language, with explanations on how we translate AMOOSE to the target framework AMUSE. We report on the results we were able to obtain in this early stage of AMOOSE in Section 6; more results will be coming as AMOOSE is an enabler for more interesting features. We explain this in Section 7. Lastly, we conclude in Section 8.

2 Related Work

A Domain Specific Language (DSL), as opposed to General Purpose (Programming) Language (GPL), is a language within a specific domain. Even before programming was known as the modern concept it is today, punch-cards were used in the Jacquard machine, invented by Joseph Marie Jacquard, to describe different patterns used when weaving a loom. The language used for imprinting these punch-cards can be seen as a DSL, in the way that this language is constrained within a certain context. Modern programming DSLs exist for a very long time [9], but have been coming up more and more for about the last fifteen years. This upward trend comes from the desire to program more productively in specific fields. DSLs achieve this by being easy to understand (even for people who do not know a lot about programming) and quick to work with. A GPL like C or Python, differs from a DSL, in the way that these languages are not domain specific, meaning that they are not intended for a single range purposes constrained by a given domain. However, the reason for needing a DSL in certain domains, is that sometimes the generic structures (functions, objects, libraries) GPLs offer do not translate well to notions or constructs in a certain domain [15].
For example, having a system of many objects with many parameter configurations would translate to a function in some GPL with many parameters, given that we want to write the least amount of code to set some the system. However, with a variable system size to set up, the number of positional parameters of the function needs to be variable as well. This becomes difficult for users, as they need to keep track of which parameters map to which position in the function call. Furthermore, a DSL is an enabler for running analytical or optimizational procedures for the user in the back-end, to provide more insight and/or consequently better code.

2.1 Examples of DSLs

There are multiple categories of DSLs: markup languages like HTML, modeling languages like ATMOL, and domain specific programming languages, such as Chisel. The oldest DSLs have evolved into GPLs, such as the COmmon Business Oriented Language (COBOL), starting out as a language mainly used in a business, finance and administration setting for processing large files. Most people would probably agree upon XML, a markup language for structuring data, being the oldest DSL, but as the border between DSL and GPL is blurry, there is no definite answer. We will give an overview of some examples of important DSLs that have been developed over the past couple of years.

2.1.1 Markup Languages

Markup languages are not always considered programming languages, as they do not allow users to compute anything, but rather structure information. It is debated that these languages are more of a format, rather than a true programming language. An example of a DSL is the HyperText Markup Language (HTML), a markup language for the purpose of structuring websites, interpreted and displayed by a webbrowser. HTML consists of different kinds of "tags", which serve different

Listing 1: Example of an HTML document.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Website title!</title>
    </head>
    <body>
        <h1>Hello world!</h1>
        <p>What a wonderful day!</p>
    </body>
</html>
```

specific purposes and can be nested. An example of an HTML program is shown in Listing 1. First, we specify that this document is of type HTML. The website is nested within two tags, the `<html>` and `</html>`, these sorts of tags are pairs of opening and closing tags. The opening tag defines the start of the section, where the corresponding closing tag ends the section. Within, we find the `head` tag, in which we define metadata, such as the `title` ("`Website title!`") of the website, and a `body` tag, defining the contents displayed on the page, such as a heading `h1` ("`Hello world!`") and a paragraph `p` ("`What a wonderful day!`") of distinct styles. The purpose of markup languages range from describing how content should be shown on a given platform, or how content is structured when communicating over an internet connection. Besides XML, PDF, LaTeXand many others, Dot is another markup programming language, with the purpose of structuring graphs.

2.1.2 Chisel

Chisel [5] is a DSL that allows users to design and construct hardware in a Scala embedded language. An interesting aspect about Chisel that corresponds to our project is that it allows users to generate simulation software, however of a different kind. Chisel, like AMOOSE, provides abstractions that correspond to the specific domain to define the simulated system. Special data-types are provided to describe system states at certain points in the modeled circuit. Furthermore, as Chisel is embedded in Scala, circuits can be generated using configurations with many parameters; in VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL), one of the most popular language for hardware synthesis, this parameterization is not as straightforward.

2.1.3 SESSL

The Simulation Experiment Specification via a Scala Layer (SESSL) [7] is another DSL, which focuses on performing experiments in general. By adding layers of abstractions, users can write code as they usually would to run experiments, but with the addition of reusing functionality and reproducibility of simulation experiments. These features are missing in the Simula [18] simulation programming language, one of the first languages to introduce Object Oriented Programming

(OOP). Simula was inspired by SIMSCRIPT, making it the first languages for simulations, in some way fitting the description of a simulation DSL, but not quite, as they were intended to be general purpose.

2.1.4 GraphIt

GraphIt [30] is a DSL for the creation of graphs and performing graph analytics, which require algorithms that are difficult to optimize by hand. GraphIt solves this by decoupling the analytics algorithm, such as PageRank, from the optimization. It does so in a declarative manner, where the user specifies what to compute, not how exactly the result is computed (an imperative description). We are also writing a declarative DSL in our project, where the user lets the compiler decide the exact method of how our desired system is modelled. If the user wants to make use of a certain particle constellation, it would not need to know which functions are called and how they are exactly called. This lets us offer the user all tools that are available, but in a simple way. The user does not have to know the exact tools that are used, but will be able to use them. For instance, much like a car, the user does not have to understand the engine in order to operate the complete system.

2.1.5 Tensorflow Eager

TensorFlow Eager [3] is a multi-stage, Python-embedded DSL for modeling for the purpose of automatic optimization of machine learning models in TensorFlow. This language is built on top of TensorFlow [2], a framework for implementing and running machine learning models in Python, created by Google. TensorFlow Eager takes an imperative approach to give more control to users, as TensorFlow is declarative, it abstracts away from detailed specifications that are otherwise expected from the user. AMOOSE does the same, but the other way around, where it provides the tools of an initially imperative language in a declarative manner. The declarative nature from TensorFlow itself allows for models being optimized later, the user is restricted to defining the machine learning model first, before any applications. TensorFlow Eager compiles to a Python script accompanied by the TensorFlow framework. The code is reordered to fit the declarative requirements, during the compilation, such that model definitions precede the application once again. This framework relates to our project, as both abstract away from a certain Python framework and generate the Python code that would have been written in the original framework.

2.1.6 ATMOL

The ATmospheric MOdeling Language (ATMOL) [6] is a DSL for atmospheric modeling, designed to improve the productivity in creating and maintaining models, yield reliable results, and serve as an extensible tool for the future. The use of this framework specifically lies in modeling the climate, ocean circulation and the weather. ATMOL is assisted by the Code-generation Tool for Applications based on Differential Equations using high-level Language specifications (CTADEL) [27], a tool for

generating specific high performance scientific codes for different target architectures and performs symbolic manipulation and code synthesis. ATMOL allows for the integration of high-level and low-level models by means of CTADEL's code synthesis, achieving reliable results and improved productivity and maintainability of modeling. ATMOL relates to our project, as both are DSLs developed as a framework in which simulations are described by the user. Both try to optimize the system configurations for yielding reliable results, while allowing for high user productivity. In general, both systems provide a language for defining models and running simulations with them.

3 Background

In this section, we will provide background knowledge on programming languages and compilation for the understanding of our method of creating a DSL. Furthermore, we will also provide background knowledge on AMUSE, as we will translate our DSL into code that uses this framework. We will explain the general idea of AMUSE and some specific interesting notions that we come across while writing code in this framework.

3.1 Programming Languages and Compilation

Programming languages would not be useful if there existed no machines that can interpret them. To understand the way in which computers process instructions originating from a program written in a particular programming language, we will first have to understand the architecture of the processing unit, of which we give a brief description.
In the von Neumann architecture [29] and Harvard architecture, the two most common computer architectures, the Central Processing Unit (CPU) of a computer consists of an Arithmetic/Logic Unit (ALU), for performing arithmetic or logic computations, and a Control Unit (CU). The Program Counter (PC) points to the current instruction in memory, consisting of an operation on source registers and a destination register. The CU decodes the instruction and instructs the ALU, memory unit or a device to perform some action, such as performing an arithmetic operation on two source registers and storing the result in a destination register. Sequences of such instructions come in the form of programs and are represented as binary strings using high or low voltage on the circuits of the bare-metal hardware of the computer. This machine code must follow the exact specifications of the instructions of the machine language that is used for the corresponding machine.
Assembly is a generalisation of machine languages in the form of a human-readable language. However, assembly can not be directly interpreted by the target machine as it is written in ASCII strings that are human-readable representations of operations. These strings form mnemonics for their corresponding instructions in directly interpretable machine language. Assembly is written according to some specific instruction set, detailing the different instructions, their format and effect. Different implementations exist for different variations of assembly, such as x86. Assembly is human-readable, but abstract in the least amount, as it is often the case that each instruction maps one-on-one to a machine code instruction; this translation is performed by the assembler.
Higher-level programming languages like C are much less platform dependent and introduce many helpful structures that support the productivity in developing programs. If-statements, for-loops and functions are all translated to assembly code, later to be assembled in machine code to be interpreted by the CPU. The process of translating code from one language to another is called compilation. Here, we often mean the translation from a higher-level language to a lower-level language, the opposite direction would often be called decompilation. In some cases we want to translate the source-code in some language to source-code in some other language, in this case we use source-to-source compiler or transpiler [4].

Mind that not all programming languages are *compiled languagess*. *Interpreted languages*, such as Python, are meant to be interpreted by some other program, an interpreter. The interpreter executes instructions on the code's behalf, rather than the code being translated into a different format that can be executed directly by the hardware.

3.2 Declarative versus Imperative Languages

There is a distinction between *declarative* and *imperative* programming languages [8], that is often explained by the difference between *what* the program does and *how* the program should do something, respectively. In imperative languages, programmers mainly write commands that form a procedure. Such a sequence of commands forms a recipe that explicitly defines *how* an initial program state is processed into consecutive intermediate states, and finally the resulting state. For example, given that we want to draw a line, we would provide commands that describe at which exact coordinates pixels are drawn on the screen, given a fixed for-loop. A declarative language such as OpenGL provides a more abstract interface by which the user can achieve the same. The user would only supply the coordinates of the start and end of the line, a description of *what* the final program state should be. Here, the exact way of performing this operation is left to OpenGL to decide.

3.3 Formal Languages

In order to implement a DSL, we design a language, containing a defined grammar, and implement a parser that makes use of this grammar. To express the grammar in the parser implementation, we make use of regular expressions. These let us define tokens, so that they can be extracted from the source code supplied by the user. To understand regular expressions, we first have to understand formal languages, as they serve as a tool for defining which words are accepted or rejected in a certain set.

A formal language L is a set of words (or string, a sequence of symbols) defined by a finite, nonempty set of symbols Σ, called an alphabet, and a set of rules that define the pattern words must follow to be an element of the language. For instance, suppose a language L' is defined by the set of characters $\Sigma' = \{a, b\}$, meaning that the language only consists of a's and b's. Now suppose that this language consists only of words that start with a, then the language consists of the words $\{a, aa, ab, aaa, aab, aba, abb, \dots\}$. The set $\Sigma^*_{a,b} = \{\lambda, a, b, aa, ab, ba, bb, \dots\}$ is the set of all strings over $\Sigma_{a,b}$ as indicated by the Kleene star operator $(*)$. Any language over the alphabet Σ is a subset of Σ^*. The string λ is the "empty string" and does not contain any characters.

The union, concatenation and closure operations are defined for languages, because languages are generators of sets. For any languages L and M, the union $L \cup M$ results in a set of elements, where $x \in L \cup M$ if and only if either $x \in L$, $x \in M$ or both. For any languages L and M, the concatenation LM results in a set of elements, where $x \in LM$ if and only if $x = lm$ where $l \in L$ and $m \in M$. The closure L^i of a language L is the concatenation of $LL \dots L$ for i times, where L^0 is defined as $\{\lambda\}$. The Kleene closure L^* of language L is the union of L^i for all non negative i.

The Positive closure L^+ of language L is the union of L^i for all positive i, the difference from the Kleene closure being that this operation does not automatically include λ in the closure, unless it is an element of L.

3.3.1 Regular Expressions

Regular expressions are a powerful tool that let us match sequences of characters. Fundamentally, they test whether a given word is an element of a given language. A regular expression "x" only matches the string consisting of a single character "x", forming the language L, where $L(x) = x$; the same holds for λ. If "r" is a regular expression for the language $L(r)$, then assuring precedence of "r", in the form "(r)", results in a regular expression for $L(r)$. Precedence is used to force the priority of one operation before another. If "r" and "s" are regular expressions for the languages $L(r)$ and $L(s)$ respectively, then "$(r)(s)$", "$(r)|(s)$", "$(r)*$", "$(r)+$" and "$(r)?$" are regular expressions for the languages $L(r)L(s)$, $L(r) \cup L(s)$, $L(r)*$, $L(r)+$ and $\{\lambda\} \cup L(r)$, respectively. The "dot wildcard", such as in the regular expression ".", is a special character that matches any character, except for a newline. We can escape special characters by prepending a backslash such as "$\backslash*$", "$\backslash.$", "$\backslash+$" and "$\backslash\backslash$". Prepending a backslash to some other characters denote special characters instead, such as "$\backslash t$" for a tab character or "$\backslash n$" for a newline. In some regular expression interpreters, the sequence "$\langle\langle \mathrm{EOF} \rangle\rangle$" matches the end of the input "$\wedge r$" matches an "r" at the start of a line, while "$r\$$" matches an "r" at the end of a line. The expression "$[rs]$" matches either an "r" or an "s".
We can use these definitions to describe the language over the alphabet $\{a, b\}$, where every word must start with an "a". First, we start with an "a", which is concatenated with a sequence of any length, containing the characters "a" and "b"; we end up with the regular expression "$a[ab]*$". Specifying an inclusive range of characters in square brackets, such as "$[0 - 9]$", matches all characters from 0 to 9.

3.4 Compilation

In order to convert a high level language into a machine interpretable format, we create a pipeline [4, 11] that receives source code, a character stream in a given language, and outputs machine code. The pipeline usually consists of a pre-processor, a compiler, an assembler and a linker. The pre-processor generates a modified version of the source code, by handling code imports, global definitions and conditional compilation. The compiler usually translates the source code into assembly code. The assembler generates relocatable machine code from the assembly, as assembly is written in a format in which addresses can still be changed. Lastly, the linker combines the relocatable machine code and library files with additional relocatable object files to generate the resulting target machine code that can be directly executed by the CPU.
In general, a compiler consists of a pipeline containing a front-end and a back-end, each consisting of a number of consecutive pipeline stages. The front-end consists of a lexical analyzer, syntax analyzer, semantic analyzer and an intermediate code generator. The back-end consists of a machine-independent code optimizer, code generator and a machine-dependent code optimizer.

The front-end is responsible for translating the code of the source language to a universal format (intermediate code), from which code can be generated in different target languages by the back-end.
During the compilation pipeline, a symbol table is used, which contains information about variables, such as their name, their line of declaration and sometimes also their type. The latter holds true for programming languages that are *strong typed* (as opposed to *weakly typed*), meaning that types are enforced on the variables that are used. While processing a statement, the compiler may expect some variable to be of a type derived from its context, the *expected type* of the variable. If the expected type does not match with the *actual type* of the variable, the compiler will try to find a definition describing the conversion from the actual type to the expected type. If it manages to find one, the conversion is used, otherwise a warning or error occurs.
We now describe the basic notions of syntax and semantics of programming languages, and the the compilation stages that are relevant to this thesis.

3.4.1 Syntax and Semantics

Programming languages are defined by their syntax and semantics, between which there is an important distinction. The syntax of a language describes the sequence of characters that make up tokens, as recognized by the lexical analyzer, and the structure of these tokens as defined by a grammar. The semantics of a language defines the interpretation of a structure of tokens, making the language dynamic. Compilers use a grammar and a definition of the semantics in order to determine what operations are determined in the target language. For example, we define that "`42 * 64`" contains the tokens "`42`", "$\star$" and "`64`" where the first and latter are considered to be numbers and the second describes an operation. The "$\star$" operation operates on the two numbers, regarding the definition of the semantics this could have different outcomes. Say "$\star$" is defined as the multiplication operator, we would generate target code that multiplies "`42`" by "`64`". However, if we define "$\star$" to be an operation that concatenates numbers in order, we generate target code that concatenates "`42`" and "`64`".

3.4.2 Lexical Analyzer

The lexical analyzer uses patterns, in our case in the form of a regular expression, to separate input into lexemes, sequences of characters that are represented by tokens. For instance, the lexeme `my_variable` is the name of a variable in our code and is represented as an **id** (identifier) token name with the attribute value of "`my_variable`" to specify the string representing the variable name; the resulting token is represented by $\langle \mathbf{id}, \mathrm{my\_variable} \rangle$. Finite Automata (FA) are machines which recognize regular languages, the languages that can be described by regular expressions. An FA is often visualised using a transition graph, containing labeled nodes and labeled arcs. A Deterministic Finite Automaton (DFA) is defined by a set of states Q, one of which being the initial state q_0, a subset F of Q containing final states, an alphabet of nonempty symbols Σ and a transition function $\delta : Q \times \Sigma \to Q$. The transition function δ defines a function that maps the elements (q, s) for all $q \in Q$ and $s \in \Sigma$ to exactly one $q' \in Q$. The labeled nodes

in the transition graph represent the states in Q, where q_0 is indicated by a start arc. The final states of F are represented by nodes with a double line. Each transition is represented by an arc, labeled with the symbol that must be read in order to transition from the node representing the current state to the node representing the successor state.
Non-deterministic Finite Automata (NFA) may include null moves, which transition from one state $q \in Q$ to another $q' \in Q$ with an empty input symbol λ. Also (q, s) may map to a subset of Q, instead of exactly one element.
Flex, the lexical analyzer we will be using, makes use of finite automata to tokenize the input. Regular expressions are supplied by the user, which have a trivial translation to an NFA. This NFA can be transformed into a DFA by redefining the states and transitions in a procedural manner. The DFA is represented in a tabular structure, ready to be used for the recognition of different tokens in the input code. This token stream is handled by the parser, which structures the tokens given a grammar.

3.4.3 Parser

A parser uses a grammar to enforce a structure on the tokens that are supplied by the lexical analyzer. In turn the parser reports back for verification, and a recovery policy in case of syntax errors. A Context-Free Grammar (CFG) $\mathcal{G}$ is defined by a set of terminals, a set of nonterminals, a start symbol from the set of nonterminals, and a set of productions. A terminal represents a token that is received by the lexical analyzer, represented by its token name and the exact value. Nonterminals are declared in the productions and represent variables that are used throughout the set of production rules. Productions map nonterminals to a set of syntactic expressions of sequences of terminals and nonterminals. These sequences are matched with the input, with respect to their order, to enforce structure on the tokens. As an example, consider a lexical grammar with regular expressions for `letter` ("$[a-zA-Z\_]$") and `digit` ("$[0-9]$"), suppose we have the tokens "**id**", "**num**", "**endl**", "=", "+" and "−", where "**id**" is given by "`letter`(`letter`|`digit`)$*$", "**num**" is given by "`digit`+" and "**endl**" is given by "$\backslash n$". We can define the following grammar to parse a sequence of assignments of simple expressions:

$\langle S \rangle$ ::= $\langle A \rangle$ **endl** $\langle S \rangle$
  | $\langle A \rangle$

$\langle A \rangle$ ::= **id** '=' $\langle E \rangle$

$\langle E \rangle$ ::= $\langle T \rangle$ '+' $\langle E \rangle$
  | $\langle T \rangle$ '−' $\langle E \rangle$
  | $\langle T \rangle$

$\langle T \rangle$ ::= **id**
  | **num**

Nonterminals S, A, E and T all have their production rules defined, separated by pipe characters ("|"). The starting symbol of the grammar is determined by the nonterminal on the left hand

Listing 2: Example code of assignment language.

```
a_1 = 5
a_2 = 40 - a_1 + 7
```

side of the first production rule. In our case, this is S, which defines a set of statements of either nonterminal A followed by a newline and more statements, or ending in A. An assignment is specified by the production rules of A, where an identifier is followed by an assignment operator and an expression E. The expression E either contains a term T added to another expression, or the expression subtracted from the term or just a term. A term is either an identifier of a variable or a number. This grammar specifies a language in which we can write assignments of variables and numbers, either added to or subtracted from each other.
The following program, found in Listing 2, can be derived from this grammar, we can simply follow the rules that make up this instance. For representation we make use of a parse tree (shown in Figure 1), a tree graph that describes the structure of the tokens with respect to the grammar. Each internal node represents a non-terminal, with at the root the start symbol. The children of the internal node are nodes that represent the terminals and nonterminals (in the same order from left to right) from the production rule used for the derivation. The leaf nodes represent terminals, as they do not have production rules of their own that would derive children.
Sometimes the grammar may be infeasible, such as in the example: "`a = b (meters)`". Whether we assign to "`a`" the result of a call to a function named "`b`" with the argument "`meters`", or a quantity consisting of a value `b` in some unit "`meters`" is not clear in this context. To make a grammar feasible from these ambiguities, we may need to add more tokens to clarify the context of the expression.
Within this thesis we will use an existing parser generator, which generates a parser for our language given a set of grammar rules. These grammar rules are annotated with instructions on how to create an Abstract Syntax Tree (AST), which is similar to a parse tree. However, where a parse tree contains every single character of the input, the AST only contains those parts that are required during the compilation. Also, an AST (as shown in Figure 2) may contain more abstract notions of the input, such as information about the type of variables, which a parse tree does not. The internal nodes of the AST are annotated with a specifier, such as an operation, the children are usually arguments to this operation. It is a much more a practical than it is a theoretic approach to representing the operations performed in the source code. During this process we can enforce and verify semantics, such as assigning types to variables. When assigning some value to a variable, we may make up an actual type from the value, which may need conversion when the actual type of the variable is different. We do need to be careful when designing our grammar, as it should be feasible. Furthermore, even if a grammar is feasible, the method of parsing the grammar needs to be unambiguous. A pitfall when defining a grammar is *ambiguity*. Given the product rules

$\langle S \rangle ::= \langle S \rangle \ \langle S \rangle$
$\quad | \quad$ **id**

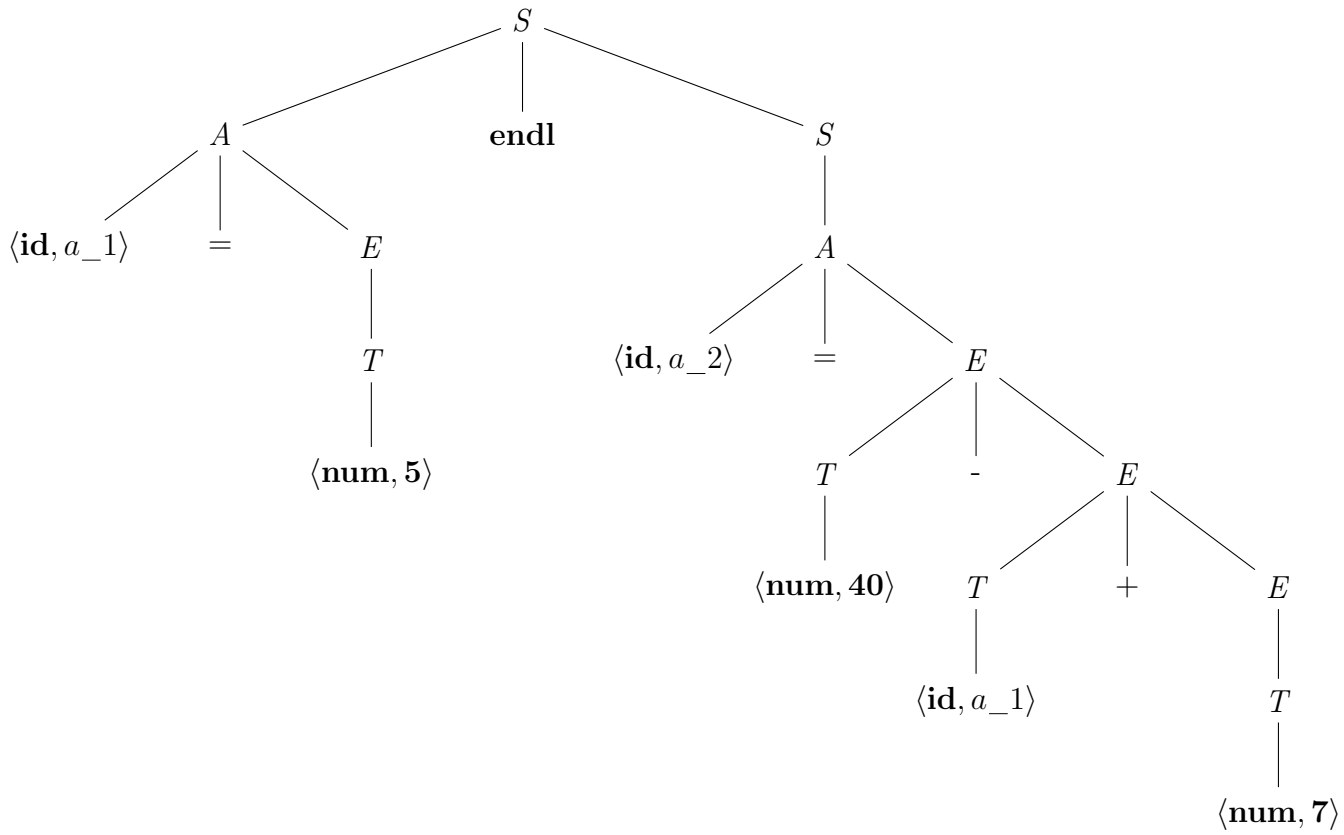

Figure 1: The parse tree generated from the code shown in Listing 2.

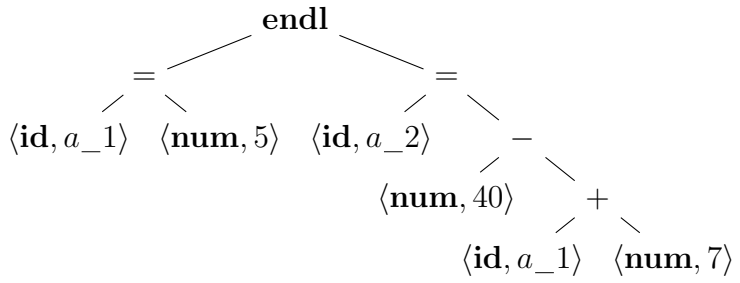

Figure 2: The AST generated from the code shown in Listing 2.

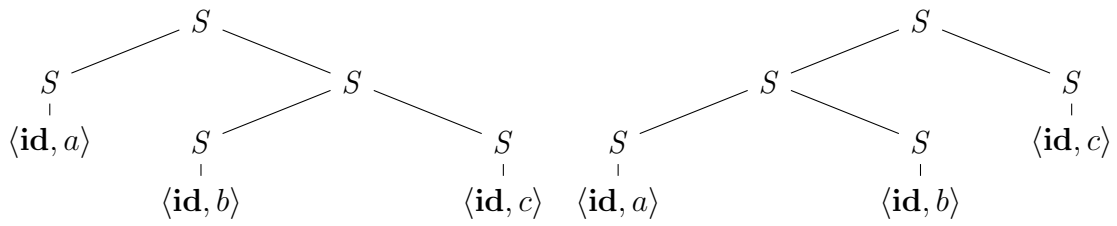

Figure 3: Ambiguity in a simple grammar.

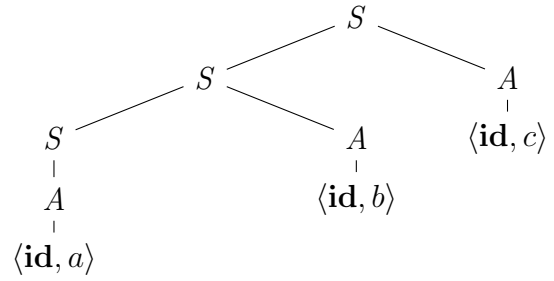

Figure 4: Resolved the ambiguity shown in Figure 3.

we may derive a either of the parse trees shown in Figure 3 for "`a b c`". There is no concrete answer as to which parse tree is correct, given the provided information, while the evaluation of one order may have a very different effect from an evaluation of its mirror image. Since the derivation of the parse tree is ambiguous, we call the grammar ambiguous as well. We can eliminate this problem by redefining the product rules as

$$\langle S \rangle ::= \langle S \rangle \ \langle A \rangle \mid \langle A \rangle$$

$$\langle A \rangle ::= \textbf{id}$$

of which we can see the result in Figure 4.
There is still another problem, the grammar is left recursive, meaning that the nonterminal S may end up producing another S as its first nonterminal, either directly (as in this case) or indirectly. If we would try to use a left recursive grammar, then we may end up in an infinite loop of a nonterminal referencing itself. There exists an equivalent set of production rules

$$\langle S \rangle ::= \langle A \rangle \ \langle S \rangle \mid \langle A \rangle$$

$$\langle A \rangle ::= \textbf{id}$$

that eliminate this recursion.

Our parser generator will generate an LR parser, which makes use of shift-reduce parsing. The parser takes an action based on the current state, defined by the position in the input and a stack that contains terminals and non-terminals. The stack is empty at the beginning of the parsing process, and the parser starts at the first position of the input. The "shift" action shifts tokens from the input onto the stack. If the top of the stack contains a sequence of tokens that matches with one of the production rules, the "reduce" action is taken to replace those tokens with the left-hand non-terminal of the corresponding production rule (applying the production rule in reverse). When the input is empty and the only non-terminal on the stack is the starting non-terminal, then the parser accepts the input. If the parser finds a syntax error, it may try to recover, but will not accept the input after.
As we are using an LR parser, we will face the challenge of adjusting our grammar, such that no conflicts occur, of which there exist two types in a shift-reduce parser. A shift/reduce conflict may be present in the grammar, where the parser can not decide whether we should reduce the topmost sequence of tokens on the stack in the current state, or whether we should shift more tokens, ultimately resulting in a different reduction. A reduce/reduce conflict may also be present in the grammar, where the topmost sequence of tokens matches with multiple production rules. In this situation, the parser is not able to determine which of the production rules should be applied, resulting in undefined behavior. These conflicts may be eliminated by rewriting a part of the grammar.

3.4.4 Intermediate Code Generator

The AST is converted by the intermediate code generator into a particular intermediate code, a representation that allows for optimization and generation of target code in different languages. The intermediate language chosen influences the possibilities in both optimization and generation, as representations with many small statements make it easy to optimize, while they also tend to result in stretched out target code. The intermediate representation may be defined as a sequence of statements, as (tree-like) objects[1] or even as an existing language (like C being the intermediate representation of C++ programs in the past). In the scope of this project, an abstract model representation models defined objects, which contain attributes that are supplied in the input code; the intermediate representation in our scope embeds abstract models in a sequence of nested statements. For different kinds of abstract models exist different ways of handling the attributes, such as models describing animals (with attributes that describe the number of heads, the sound it makes and whether it can jump) are handled differently than models describing buildings (with attributes that describe the surface area and the number of floors).
For imperative languages, three-address code is often chosen for its simplicity, consisting of sequences of statements, each consisting of a single operation and optionally two sources and a destination variable, indicating where the result of the operation is stored. The sources that may be used are either constant values, variables or temporary variables generated during the intermediate code generation. For example, a statement could be "`a = b`", having one source and one

[1] These are similar to the data structures found in Python's "ast" (Abstract Syntax Tree) library.

destination variable, where the value of "b" is copied to "a". Another example, "`a = 8 * b`", where the two source values "`8`" and "`b`" are multiplied and the result is assigned to the destination variable "`a`". We could also have a statement "`exit`", which exits the program, and has no source nor destination values.
It is evident that allowing at most three values in a single statement will make the target code span many lines. The simplicity makes code easy to optimize and simplifies target code generation for different hardware architectures; however, the target code will lack human-readability. In general, for a DSL, the intermediate representation should be in a format from which we will be able to easily construct code in different target languages (not necessarily assembly), while staying faithful to the abstract structure of the original input format. Representing model-like structures from the input code as a sequence of fixed operations would narrow the set of target languages that can be easily derived from this representation. Namely, by choosing what operations would be obvious for one target language, would confine some other target language which operates differently.

To show why this is important, suppose we had the following statement in an imaginary language $\mathcal{BFG}$:
`Runt is a friendly giant.`
Then suppose that Python is our target language, and we use a library module that provides the "`Giant`" class, which has a member "`mood`":
`Runt = Giant(); Runt.mood = "friendly".`
The first statement assigns the result of a function call to "`Giant()`" (a constructor) to a variable "`Runt`". The second statement is translated to an assignment of the string "`"friendly"`" to the attribute `mood` of "`Runt`". This sequence of operations would be translated to:
`assignment(access(Runt, mood), "friendly").`
The front-end of our compiler is fixed and can not be changed to translate the source code from $\mathcal{BFG}$ to a fixed intermediate representation. The back-end of our compiler is variable and may be swapped out to translate to different target languages. Now suppose we wanted to translate this specific intermediate statement to a C target program, where the `Giant` class stores all of its attributes in a mapping, specified by the "`attributes`" attribute, like so:
`Runt = Giant(); Runt.attributes["mood"] = "friendly".`
The intermediate representation "`assignment(access(Runt, mood), "friendly")`" does not align with the second statement in the C target code. It seems we could solve this by handling this intermediate statement with an additional compilation stage to adapt to the new target language. We would translate the intermediate representation to:
`assignment(index(access(Runt, attributes), "mood"), "friendly").`
We effectively created a mapping $\mathcal{M}$ from "`access(Runt, mood)`" to "`index(access(Runt, attributes), "mood")`". Now suppose we had the following statement in the $\mathcal{BFG}$ source language:
`Runt snoozes.`
Both in Python and in C, we would expect the following target statement:
`Runt.snooze().`

However, since we have defined the mapping $\mathcal{M}$, we get:
`Runt.attributes.snooze()`.
Which does not have the intended effect in the target language, and could have been avoided by maintaining an abstract description of models in $\mathcal{BFG}$, rather than concretizing statements in the front end. In other words, by coming up with a definite structure of the target code in the front-end, we will confine the freedom of the back-end.

3.4.5 Back-end

The back-end consists of the code optimizer and code generator. The back-end is different depending on the target language, so it will not necessarily generate assembly code. When optimizing code, dependencies among variables must be taken into account. Swapping locations of certain statements may break these dependencies, so it is discouraged to do so within a representation that does not provide enough overview, one that is very abstract. For three-address code it takes the compiler a small transformation to come up with an appropriate assembly program, as names of register can be substituted for variables in most cases. Still, the availability of the registers must be respected during this stage. Certain operations like function calls require more code in assembly than their three-address code counterparts, but can be easily generated nevertheless.

3.5 Astrophysical Multipurpose Software Environment

The Astrophysical Multipurpose Software Environment (AMUSE) [23] is a framework written in the Python programming language for modeling and simulating astrophysical systems. AMUSE adds many features to the programming environment, such as systems of units, pre-defined constants, particles and particle sets, community codes and bridges. These features are contained in a variety of modules that are part of the library. The Python language itself is not altered by AMUSE whatsoever, so the Python grammar [1] applies to this framework. Particle sets are used for storing e.g. planetary systems, a static description of the model. This and the description of the dynamics of the system, in the form of community codes, form the *precondition* of the simulation. In Figure 5 we see an example of a system, where we have a particle set consisting of the sun, earth and moon. The edges between the sun and earth, and earth and moon represent community codes that simulate the physical gravitational force onto the respective partakers. Since these community codes result in discretized simulations of continuous events, time-steps are used as intervals of discretization. Using multiple community codes at once, calls for the synchronization of the different time-steps, bridges are used to evolve the system in an interleaved manner to handle these distinct community codes with different time-steps. We will elaborate on the features of AMUSE after explaining the n-body problem, which is one of the many problems that astrophysics tries to solve.

3.5.1 N-Body Problems

The n-body problem is a recurring problem in physics that is unsolved for $n > 3$. However, computational astrophysics comes close by approximating a solution by modeling and simulation. The problem is stated by Rosenberg [25, p.364] as: "Each particle in a system of a finite number (n) of particles is subjected to a Newtonian gravitational attraction from all the other particles, and to no other forces. If the initial state of the system is given, how will the particles move?"
In other words, suppose we get a description of the states S_0 of n interacting celestial bodies, can we predict the state S_t of these bodies after a duration of time t? The n-body problem comes with its own set of n-body units in its computations, proposed by Hénon [12], such that the computed values are independent from the mass, dimensions and the number n amount of bodies of the system. This is desirable as the invariability makes the results applicable to systems with other numbers of n bodies. By choosing appropriate units for mass (U_m) and length (U_l) we obtain the independence from the mass and dimensions of the system. To make the results independent from the number of n bodies in the system, a basic unit for time (U_t) is defined; from here n can be eliminated from the derived units.
With these units, the evolution of the system during some time-step Δt is computed from the initial state, where Δt should be much smaller than the time for the system to end up in an equilibrium. As the state changes and bodies may escape the system, the time-step changes accordingly, thus rectifying this iterative approach. Given the updated time-step, we evolve the system to its second derived state.

3.5.2 Units, Quantities and Constants

Values in physics come with units, which describe what type of quantity is being specified. The International System of Units (SI) is a system of measurement that is preferred in science. This system offers a list of base quantities and a set of derived quantities. The base quantity "length" is measured in the base unit meters, where the symbol for meters is "m". The rest of these base units can be seen in Table 2 (in the Appendix). All of these units are present in AMUSE, among some derived quantities, seen in Table 3. Derived quantities are common combinations of base quantities, also given their own names. For instance, Coulomb is derived from Ampere and seconds. AMUSE also provides the definitions of units that are based on measured quantities or constants (Table 5), as well as astronomical units (Table 4); these are often used when working with astronomical objects. AMUSE also offers temporal units derived from seconds (Minute, Hour, Day, Year, Julian year, Meter per second, Kilometer per second), imperial units (Inch, Foot, Mile), centimetre-gram-second system units (Gram, Centimeter, Erg, Barye), angle units (Revolutions, Degree, Arcminutes, Arcseconds), and percent being a factor of one-hundredth. AMUSE also offers prefixes from yotta (10^{24}) to yocto (10^{-24}) for scaling units by a factor of certain powers of 10.
Units are imported from the module found in `amuse.units.si`, `amuse.units.derivedsi` and `amuse.units.units`. A value "`<value>`" expressed in terms of some unit "`<unit>`" are declared by writing "`<value> | <unit>`, for example `1 | units.yr`". A vector quan-

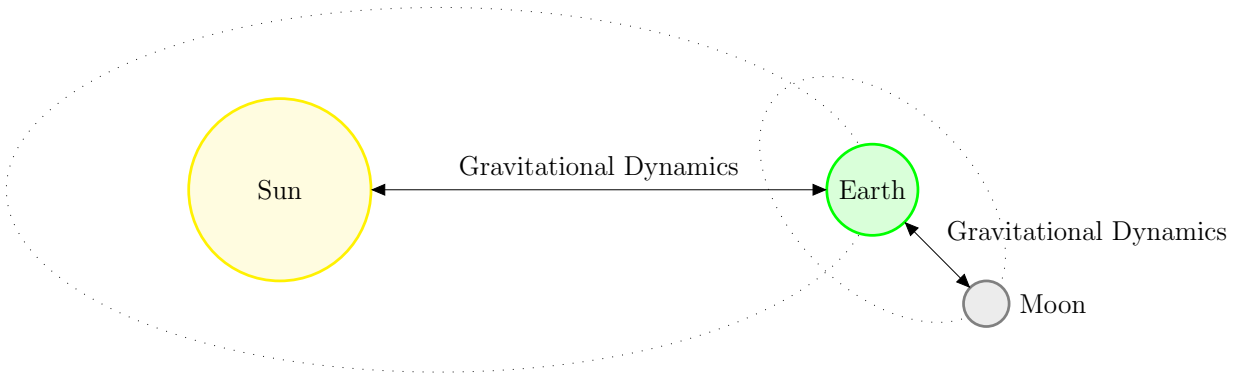

Figure 5: Graphical representation of an example system that can be created in AMUSE.

tity is created by writing " `[] | <unit>`", which is different from a list of individual quantities, as operations on a vector quantity are performed element-wise. For example, multiplying a vector quantity by 2 multiplies every element by 2, while multiplying a list of quantities by 2 will result in a concatenation of two lists with the same elements. Quantities can also be tested for its unit and can also be converted to a numeric value. The file `amuse.units.constants` provides many constants commonly used in (astro)physics. The code in `amuse.units.trigo` provides "unit aware" trigonometry operators and short-hands for converting an angle to radians, degrees or revolutions. The code in `amuse.units.generic_unit_converter` provides code for converting between unit systems, e.g. SI units and N-body units, by using converters. These converters require reference points for the conversion of masses and the conversion of length or time, in the form of a mass quantity and a length or time quantity. Getting the order of magnitude wrong on these values will negatively influence the integrity of the simulation results.

3.5.3 Data model and Particle Sets

AMUSE uses containers as data model for storing object sets. The advantage of containers is that operations are implicitly performed on all elements in the container, this eliminates the need for explicitly iterating over all elements. Containers are provided as either one-dimensional sets, called "Particles", or as a static multidimensional container, called a "Grid", which represents a grid of "GridPoint" entries. The set of attributes of the objects stored in containers is dynamic, new attributes can be defined by a simple assignment. Containers can be stored in memory, in a community code or on disk (using HDF5). Particle sets can be sliced to obtain subsets, which make use of references to the original set, making subsets of one element different than an individual Particle, which stores its own values.

3.5.4 Community Codes and Bridges

Community codes are source code modules that describe the transitioning of a single state of variables to another state, regarding a dynamic, such as gravitational dynamics. These codes are

developed (to this day) by the astrophysics community, possibly written in another language than Python, however interface with the AMUSE framework. These community codes have an internal time-step that discretizes the continuous evolution, as we have described for the n-body problem. An iterative approach approximates the state of a system after the evolution for some period, with a time-step that may vary each iteration. Computing the state of a complex continuous system with many interactions at time t from an initial time t_0 is relaxed into a problem of iterative simulation, at the cost of a larger error. This accuracy-performance trade-off will be a common theme when we use time-steps; a small time-step yields more accurate results, but will result in a worse performance due to more iterations, and vice versa. The cumulative configuration of community codes describe the dynamics of the system.

As we have mentioned before, particle set values can be stored in a community code, where it will locally advance the values to form successive system states. To obtain these values in the main context of our AMUSE program, we make use of channels. Channels allow AMUSE to interface with community codes and are implemented as objects that copy values from particle sets to community codes, or the other way around. Each time we would like to read values of a system after applying some community code, we copy the values back into their original particle sets before any read.

However, systems that describe a sole interaction are not interesting, so the simultaneous use of community codes is essential to AMUSE. The internal time-step of a community code may vary among community code instances, even if the instances are of the same type, depending on the part of the model they are applied on. To synchronize different community codes that run simultaneously and differ in time-step, the user is provided bridges [10, 24]. Bridges make use of the leapfrog scheme, an algorithm that updates the position and velocity of a system alternately. Suppose we are given a function $f(t) = \exp(tD_H)f(0)$, $f(t)$ is the state of a system after time t, $f(0)$ is the initial state of a system and $\exp(tD_H)$ is the operator that describes the evolution from $f(0)$ to $f(t)$ given the differential of Hamiltonian H (an operator that describes the nature of the evolution). Then, for a Hamiltonian of an n-body system, we end up with an operator that may be split up into a Hamiltonian describing the Keplerian motion of all bodies and a Hamiltonian describing the interactions of the bodies. We end up with a second-order leapfrog scheme, describing the updates to the acceleration, velocity and position of bodies. We may use this system of equations to define an evolution that first applies a "velocity kick", a change in momentum due to interactions between the bodies, secondly a "drift", a change in position due to the e.g. Keplerian motion, followed by another velocity kick. This principle of having interleaved kicks and drifts describes the general idea of a bridge, where the community codes are also applied in a kick-drift-kick manner. The evolution does not have to start at $f(0)$, we can simply have $f(t + \Delta t) = \exp(\Delta t D_H)f(\Delta t)$, where Δt is the time-step of the bridge itself. The bridge time-step must be greater than the internal time-step of all contained community codes; we apply the community codes an integer amount of times, if the bridge time-step would be smaller than that of a community-code, we would apply it zero times. The time-step is set by the programmer and is subject to the same performance-accuracy trade-off as we have mentioned before.

4 Language Design

In this section, we explain the steps we have taken in order to come up with a design of the AMOOSE language. We first identified the issues of both novice and seasoned users. There are some additional requirements that we came up with, in order to make AMOOSE an enabler with features that open up possibilities that can not be achieved in AMUSE, such as the calibration of parameter values. Next, with our findings, we can reason about performing transformations on existing scripts. We use these transformations to make generalizations for a new grammar, which will ultimately define the AMOOSE language.

4.1 Identification of Issues

In order to generate code in the AMUSE framework, we need to become acquainted with AMUSE ourselves. We have looked at the AMUSE tutorials [19], which teach us about the different tools AMUSE has to offer. These tutorials include example scripts for creating particle sets, using community codes and applying bridges. From these tutorials we can identify recurrent code structures, which should be reduced as much as possible. We also held interviews with computational astrophysics students, where we had an opportunity to look at code that was written for their course projects. This allowed us to gather the common stumbling blocks of the students and ask them about what they think could be improvements. Besides students, there are also astrophysics researchers who use AMUSE as a tool, but use it for their occupation in a research setting. For these experienced users, there are known issues that were identified before the start of this project.

4.1.1 Interviews

Students develop sub-optimal code, because the framework is overwhelming for them. Their main focus becomes studying the programming environment, instead of studying the theory of simulating astrophysical systems. We interviewed students to ask them about problems that held them back significantly, which could be solved by any improvement to the AMUSE framework, disregarding the theoretical content of the course. The findings can be summarized as follows:

- A precondition in the AMUSE framework (a description of the static model and of the system dynamics) usually requires the configuration of some parameters, of which the effect on the simulation and intended value are difficult to determine.
- The general structure of configuring bridges may not be intuitive for inexperienced programmers. Picking a community code for some use case leads to guesswork, as it is not clear how most of them operate.
- The application of a bridge is done using a for-loop and gets confusing with the time-step required by the bridge, model time and time interval of each loop iteration. Besides, creating

channels to copy values used by the bridge back to user written code is easily overlooked. These values have to be gathered manually in order to generate a plot.

- When an internal time-step for a community code is too small, the evolution shows undefined behavior, while the user is still allowed to execute the code without warning. Even though the control-flow and general structure of the script is correct, it gets confusing when unexpected results are obtained due to a wrong configuration.

The main problems users have with the framework can be traced back to the grammar. Users have to describe exactly *how* computations are performed, opposed to *what* they want to accomplish. Some parameters are unclear to novice users, while other structures are trivial and perceived as excessive. This calls for a declarative approach which reduces the amount of decisions users have to make and reduces the amount of labor required to define a system.

4.1.2 Recurrent code structures

Some known issues experienced users have with the framework have to do with reoccurring statements. For example, as seen in Listing 3, the way in which we specify units should be shorter, it now requires a pipe character ("|") to specify the creation of a quantity, and it also requires the specification of the units package. This relatively large segment of code describes the creation of only two bodies that are positioned relative to each other. We should be able to write a simpler description, using only the essential information. Sometimes the user needs to provide values or conversion strategies, that seem evident to astrophysicists, but not to the Python language. We should incorporate domain knowledge in our translation process, in order to be applied automatically rather than manually.

4.1.3 Requirements

Besides what we have learnt from the interviews, there are some additional requirements that are to be considered. ANTLER should automatically configure parameters; as a proof-of-concept we will start out by automatically determining the time-step parameter used in bridges. We require the language to be easily configurable and extendable, as AMUSE is still regularly updated with new community codes and other features, such as models. Therefore, we will provide entry points to the compiler, that do not require any in-depth knowledge of the compiler itself. The generated target code in the AMUSE framework should be human-readable so that users can easily interpret it when making manual changes. This means that temporaries should have comprehensible names, that the target code represents the source code clearly and that the target code is structured in a way a programmer would.

4.2 Designing AMOOSE

Now that we have identified the issues users have with the current AMUSE framework, we can start designing a programming language with a more declarative approach rather than one that is

Listing 3: Particle set creation in an AMUSE script.

```
from amuse.units import units
from amuse.lab import Particles

sun_and_earth = Particles(2)
sun = sun_and_earth[0]
sun.mass = 1 | units.MSun
sun.position = (0,0,0) | units.au
sun.velocity = (0,0,0) | units.kms
earth = sun_and_earth[1]
earth.mass = 1 | units.MEarth
earth.position = (1, 0, 0) | units.au
sun_and_earth.move_to_center()
```

imperative, to attempt resolving the issues that were mentioned. Since we have worked with some toy programs in AMUSE, from the tutorials [19], and have had astrophysics students explain their thoughts on programming, we are able to design the language closer to the perspective of an astrophysicist. We will be targeting the design of static models, system dynamics and the execution of the simulation, in that order. We do so, because static models are most straightforward to design, we will be able to apply the same ideas of transforming the information on the other two later. Furthermore, these three segments are usually present in this specific order in an AMUSE script, it makes more sense to handle them in the same order, regarding our task of determining which statements are redundant.

4.2.1 Particle Sets

When designing AMOOSE, we came up with a simple procedure that transforms the imperative description into a declarative one. We take for example the excerpt in Listing 3 from the official AMUSE tutorial on particle sets [22].
First, we take all specifications that are required to set up the intended system. In this case, we create an instance of Particles named "`sun_and_earth`", a particle set of size two. The given size of the particle set is implicit, since it equals the number of particles defined. Therefore, the size is not required to be specified, but we should still allow an explicit specification. Next, the particle "`sun`" is specified to be the first element of the set, but this is implicit given the particles are defined in this order. The "`mass`", "`position`" and "`velocity`" properties of the "`sun`" particle are assigned quantities, a pair of a value and a unit. In these assignments, only the property name, value and unit are required. The same goes for the specification of the "`earth`" particle. Lastly, the particle set "`sun_and_earth`" is moved to the center using an explicit call to the "`move_to_center`" function.
In our next step, we filter out and restructure the informative tokens in a way where we can still recognize the general idea. For the running example, the result is shown in Listing 4. We can structure the code by indentation, where a tab character is placed to create a horizontal offset

Listing 4: Filtered and restructured informative tokens.

```
sun_and_earth Particles
    sun
        mass 1 MSun
        position (0,0,0) au
        velocity (0,0,0) kms
    earth
        mass 1 MEarth
        position (1, 0, 0) au
    center
```

that specifies an indented block. The statements in indented blocks (and all its subordinates) apply to the line that precedes the indented block to introduce hierarchy. For example, all lines succeeding the first line in Listing 4, add information that corresponds to the "`sun_and_earth`" particle set. In the same way, the property specifications add information to the particles "`sun`" and "`earth`", which are part of the "`sun_and_earth`" set. Lastly, the "`move_to_center`" function (now dubbed the "`center`" action) also regards the "`sun_and_earth`" set.
Next, we should take into consideration that the code should be grammatically feasible, such that we can retrieve the original AMUSE code. For instance, there is no clear distinction between "`sun`" and "`center`" in the case that "`sun`" would not contain property specifications. We can differentiate between "`particles`", "`properties`" and anything else by adding *keywords*. Keywords are reserved tokens that have predefined additional semantics in a programming language. We ought to be careful when picking keywords, because reserved words can usually not be used by the user for any other than the intended purpose.
For example, given that at some point we want to specify that some particle is a satellite of another particle; we would rather use "`around`" as a keyword than "`satellite`". Being a noun, "`satellite`" is more likely to be used to indicate a satellite object instance, as opposed to "`around`". We add a number of keywords for grammatical feasibility as seen in Listing 5. We also add more keywords to make the code more readable; the "`with`" keyword indicates that a certain configuration will be specified that comes *with* the declared variable. The result is shown in Listing 6. In Listing 7 we show that we may add some rules that provide more freedom in the structure of the code. Because the "`units`" module and "`Particles`" class were used in this excerpt, we automatically generate statements that import the corresponding packages and modules in the environment of the target program.

4.2.2 Configuration file

A variety of parameters need to be set in an AMUSE script for certain community codes, as seen in the excerpt in Listing 8. These parameters may depend on the architecture of the machine. We add a feature where AMOOSE allows users to isolate the code from the parameter configuration, in a separate configuration file. As a result, an AMOOSE code can be used with different

Listing 5: Adding keywords for grammatical feasibility.

```
sun_and_earth Particles
    particles
        sun
            properties
                mass 1 MSun
                position (0,0,0) au
                velocity (0,0,0) kms
        earth
            properties
                mass 1 MEarth
                position (1, 0, 0) au
    center
```

Listing 6: Adding keywords for readability.

```
sun_and_earth are Particles with
    particles
        sun with
            properties
                mass 1 MSun
                position (0,0,0) au
                velocity (0,0,0) kms
        earth with
            properties
                mass 1 MEarth
                position (1, 0, 0) au
    center
```

Listing 7: Relaxing the structure of the code.

```
sun_and_earth are Particles with
    particles
        sun with properties
                mass 1 MSun
                position (0,0,0) au
                velocity (0,0,0) kms
        earth with properties
                mass 1 MEarth
                position (1, 0, 0) au
    move_to_center
```

Listing 8: Parameter configuration in an AMUSE script for a community code instance called "hydro".

```
hydro.parameters.use_hydro_flag = True
hydro.parameters.radiation_flag = False
hydro.parameters.gamma = 1
hydro.parameters.isothermal_flag = True
hydro.parameters.integrate_entropy_flag = False
hydro.parameters.timestep = 0.01*Pinner
hydro.parameters.verbosity = 0
hydro.parameters.eps_is_h_flag = False
hydro.parameters.gas_epsilon = eps
hydro.parameters.sph_h_const = eps
```

Listing 9: Parameter configuration in an AMOOSE script for a variable called "hydro".

```
hydro:
        use_hydro_flag True
        radiation_flag False
        gamma 1
        isothermal_flag True
        integrate_entropy_flag False
        timestep 0.01 * Pinner
        verbosity 0
        eps_is_h_flag False
        gas_epsilon 0.1 | units.au
        sph_h_const 0.1 | units.au
```

parameter configurations without having to change the AMOOSE code itself. An example of the isolated configuration is shown in Listing 9. Note that the values assigned to the parameters are specified in Python notation, as the user may want to use expressions containing function calls, which are not currently supported by AMOOSE. We see that the variable "`hydro`" is specified, followed by a colon ("`:`"). On separate lines we specify parameter configurations: the name of a parameter followed by white-space and the value assigned to the parameter. For example, the "`use_hydro_flag`" parameter of "`hydro`" is set to "`True`".

4.2.3 Bridges

In Listing 10, we see the creation of a bridge in AMUSE. The first three lines import the "`bridge`" module, along with the community codes "`Huayno`" and "`Fi`". Next, we declare an instance "`gravity`" of community code "`Huayno`", using a converter which has been declared outside of the current scope. Particles are added to the community code instance, followed by the declaration of a channel mapping. The channel mapping is assigned multiple mappings from a key (in the form of a string, e.g. "`from_disk`") to a channel object, which specifies the source and destination

Listing 10: Bridge creation in an AMUSE script.

```
from amuse.couple import bridge
from amuse.community.huayno.interface import Huayno
from amuse.community.fi.interface import Fi

gravity = Huayno(converter)
gravity.particles.add_particles(bodies-moon)
channel = {"from_stars": bodies.new_channel_to(gravity.particles),
           "to_stars": gravity.particles.new_channel_to(bodies)}

hydro = Fi(converter)
hydro.particles.add_particles(disk)
hydro.dm_particles.add_particles(moon)
channel.update({"from_disk": disk.new_channel_to(hydro.particles)})
channel.update({"to_disk": hydro.particles.new_channel_to(disk)})
channel.update({"from_moon": moon.new_channel_to(hydro.dm_particles)})
channel.update({"to_moon": hydro.dm_particles.new_channel_to(moon)})

gravhydro = bridge.Bridge(use_threading=False)
gravhydro.add_system(gravity, (hydro,))
gravhydro.add_system(hydro, (gravity,))
gravhydro.timestep = 0.2*Pinner
```

for the copy procedure. A similar code segment follows for the instance "`hydro`" of the "`Fi`" community code. Note that "`moon`" is added to "`dm_particles`", a particle group additional to "`particles`", in the environment "`hydro`" acts upon. Also note that the channel mapping has been declared earlier and may only be updated; A redefinition of the channel mapping will discard all prior mappings. Next, the bridge "`gravhydro`" is created, to which the community code instances are added, with respect to their antagonists. Lastly, a "`timestep`" attribute is set as the time-step used for the bridge itself. The "`category`" keyword can be used to generate multiple final target codes, each with a different instantiation of a community code of a chosen category. For example, if the user specifies "`category(gravitational_dynamics.pure)`", a final target code is generated for every subcategory of the pure gravitational dynamics codes.
In Listing 11 we show the restructuring of the essential information that is required to generate the code in Listing 10. As can be seen, the definition of a bridge can be greatly simplified. We removed the module imports, as these are implicit when using certain community codes. As we will explain later, we can store information about imports in a supplementary file that comes with the transpiler. In this file we can look up the module and package corresponding to some community code that is used. We can automate the creation and configuration of the channel mapping, as this is implied by the addition of particle sets to community code instances. As bridges contain community codes, the community code declarations are subordinate to the declaration of the bridge. The value of the time-step can still be specified by the user as a subordinate to the bridge declaration using the "`properties`" keyword, however we will add a feature where this

Listing 11: Bridge creation in an AMOOSE script.

```
gravhydro is Bridge with
        codes
                code Huayno as gravity on particles bodies - moon
                code Fi as hydro on particles disk; dm_particles moon
```

value can be automatically determined. In our interviews we have seen user issues with choosing an appropriate community code. With our implementation, the manual selection of non-trivial configurations becomes much easier, as well as the creation of bridges, solutions to two issues we discovered from our interviews. In the future we will be able to suggest community codes to the user automatically, without the involvement of any manual selection.

4.2.4 Bridge Application

Listing 12 shows the application of a bridge on the precondition. A number of calculations are specified to find the total energy of the system, followed by variables used for determining the duration and period of the application of the bridge. In the while loop, the current model time is updated and some output is printed. The model is evolved, after which we compute the new error and show it to the user. The channels are copied from the community codes to the original particle sets and more output is displayed to the user. After the application loop, the codes are stopped. In Listing 13, we show an equivalent AMOOSE variant of Listing 12. Since the code in between the actual application of the bridge (loop, evolution and data retrieval) varies greatly per use case, we let the user specify what happens in between, using literal code sections. These are copied straight to the target code at their respective positions. To apply the bridge, we only require its name, the simulated duration and the name of the variable that holds the error of the simulated system. In the future, we plan to generate the error of the system, as we shall then look more into the heterogeneity of the calculation of the error. The keyword "`evolve`" indicates the evolution of the bridge and "`copy`" specifies the retrieval of the particle values from the community code instance.

4.2.5 Current AMOOSE Statements

Now that we have shown examples of how we would transform segments of AMUSE code to AMOOSE, we show which statements can be used in the current AMOOSE language. These are generalisations of structures that will be compiled to their corresponding equivalent in the AMUSE framework. The exact grammar rules that describe AMOOSE and the regular expressions used for the tokens that are allowed are listed in Appendix C and Appendix B respectively.

Read/write. Reading an object from a file is achieved by writing:
`read <name> from <string>`,
while writing a model to a file is achieved by:

Listing 12: Bridge application in an AMUSE script.

```
from amuse.ext.composition_methods import *
from amuse.ext.orbital_elements import orbital_elements_from_binary

gravity_initial_total_energy = gravity.get_total_energy() + hydro.
    get_total_energy()
t_end = 1.0 yr
dt = 0.1 * t_end
model_time = 0 | units.Myr

while model_time < t_end:
        model_time += dt
        orbit_planet = orbital_elements_from_binary(bodies[:2], G=constants.G)
        orbit_moon = orbital_elements_from_binary(bodies[1:3], G=constants.G)
        print("Planet:", "ae=", orbit_planet[2].in_(units.AU), orbit_planet
            [3])
        print("Moon:", "ae=", orbit_moon[2].in_(units.AU), orbit_moon[3])
        gravhydro.evolve_model(model_time)
        dE_gravity = gravity_initial_total_energy/(gravity.get_total_energy()+
             hydro.get_total_energy()) - 1
        print("Time:", model_time.in_(units.day), "dE=", dE_gravity)
        channel["to_moon"].copy()
        channel["to_bodies - moon"].copy()
        channel["to_disk"].copy()
        print("S=", bodies[:3])
        print("g=", gravity.particles)
        print(gravity.particles.y.in_(units.au), moon.y.in_(units.au))
hydro.stop()
gravity.stop()
```

Listing 13: Bridge application in an AMOOSE script.

```
`-
from amuse.ext.composition_methods import *
from amuse.ext.orbital_elements import orbital_elements_from_binary
gravity_initial_total_energy = gravity.get_total_energy() + hydro.
   get_total_energy()
-`
apply gravhydro for 1.0 yr with error dE_gravity
        `-
        orbit_planet = orbital_elements_from_binary(bodies[:2], G=constants.G)
        orbit_moon = orbital_elements_from_binary(bodies[1:3], G=constants.G)
        print("Planet:", "ae=", orbit_planet[2].in_(units.AU), orbit_planet
           [3])
        print("Moon:", "ae=", orbit_moon[2].in_(units.AU), orbit_moon[3])
        -`
        evolve
        `-
        dE_gravity = gravity_initial_total_energy/(gravity.get_total_energy()
           + hydro.get_total_energy()) - 1
        print("Time:", model_time.in_(units.day), "dE=", dE_gravity)
        -`
        copy
        `-
        print("S=", bodies[:3])
        print("g=", gravity.particles)
        print(gravity.particles.y.in_(units.au), moon.y.in_(units.au))
        -`
```

`write <name> to <string>.`
In both statements, "`<name>`" name corresponds to the affected object. The involved file is specified by its filename in "`<string>`".

Literal code sections. As there are still aspects of Python that were out of scope for this project, we can achieve backwards compatibility[2] by introducing literal code sections. This feature is very useful for allowing heterogeneous code, as well as making use of class declarations, function definitions, function calls and other unsupported features. These sections consist of Python code surrounded by markers. With respect to its position, a literal code section is copied straight to the target code. Literal code sections are created by writing:

```
`_
<code>
_`
```

Here, "`<code>`" may span multiple lines and can contain a mix of statements and imports. Import statements are separated from the rest of the literal code section, to place them at the start of the target code. All duplicate imports are removed during the generation phase to avoid redundancy. A literal code section starts at "`` `_ ``" and ends with "`` _` ``", isolated on separate lines.

Assignment. We support assignment statements, which describe the assignment of the result of an expression on the right hand side of the assignment operator, to a variable on the left hand side. For consistency, we adopt the grammar of the Python language, to recognize its resemblance to the AMOOSE grammar later. Expressions are sequences of unary and binary operations on different atoms. Atoms are either identifiers that represent variables, the Boolean *True* and *False* values, a string of characters, a number, a tuple or an expression enclosed in round brackets. The operations supported in expressions are "`a ** b`", "`+a`", "`-a`", "`~a`", "`a * b`", "`a / b`", "`a // b`", "`a % b`", "`a @ b`", "`a + b`", "`a - b`", which represent exponentiation, unary plus, unary minus, Boolean negation, multiplication, division, floor division, modulo, matrix multiplication, addition and subtraction respectively. Different assignment operators exist, as we may also directly use the variable on the left side to be used in the expression, while also storing the result in the variable on the left side. For instance, the code "`a += b`" is equivalent to "`a = a + b`". These are called "augmented assignment operators" and we support "`+=`", "`-=`", "`**=`", "`@=`", "`//=`", "`/=`", "`%=`", "`&=`", "`|=`", "`^=`", "`<<=`", "`>>=`", representing the addition, subtraction, exponentiation, matrix multiplication, floor division, division, modulo, logical AND, logical OR, logical XOR, logical shift left and logical shift right assignment operators respectively. We should also note that we support AMUSE quantities, where a quantity is an expression followed by a unit. Simple units consisting of a single word are supported, however combinations of multiple units, such as "meters per second" ("`units.meter / units.second`") are not. This is left as future work, however, using literal code sections, this can still be achieved.

[2] In C this is done similarly regarding assembly code.

We also implemented a feature that translates "a += b" to the function "add_particles" in AMUSE, given that a is a particle set and that b is a particle or particle set. This function adds the particle "b" to the particle set "a".

Particle set creation. Particle set creation is performed with the following structure:

```
<var> [is|are] <model> [(<size>)]
    [<specifiers>]
```

An example is shown in Listing 6. Here, "<var>" is an identifier, by which the particle set can be referenced later. This is followed by the keyword "is" or "are", followed by "<model>", which references the specific particle set model the user wants to use. The "<size>" attribute of the particle set is optional and can be specified inside round brackets after the model name. The optional "<specifiers>" are a mix of actions, initializer lists or particle creations. If not used by the model, these specifiers retain their order in the target code. Specifying the action "center" centers the particle set. The action "around <particle>" positions the model relative to "<particle>".
An initializer list is specified by the keyword "properties", followed by an indented block containing intializers. These initializers follow the grammar "<name> <quantity> [as <rename>]", where "<name>" is the name of the property, which is assigned the value of "<quantity>". The optional as <rename> declares a variable "<rename>", by which the value "<quantity>" can be referenced later. A particle creation list is specified by the keyword "<particles>", followed by an indented block containing particle creations. These are of the form:
"<name> [<initializer_list>]",
where the particle is given the name "<name>" and initialized using an initializer list.

Actions. Similar to the "+=" operator in AMOOSE, resulting in a call to "add_particles" in the resulting AMUSE code, we sometimes want to make use of special actions that are stand-alone. When adding new particles to a particle set, we may want to re-center the set as a whole. For this purpose, we support actions. The center action is invoked by writing "center <name>", where the variable specified by "<name>" is centered. For example, the resulting code for "center bodies" will be "bodies.move_to_center()".

Bridge creation: Similarly to particle sets, bridges are created with the following structure:

```
<name> [is|are] bridge
    [<specifiers>]
```

In this case, the "<specifiers>" include initializer lists and code application lists. A code application list is specified by the "codes" keyword and consists elements of an optionally renamed code and a particle group list "<optionally_renamed_code> on <particle_group_list>". An "<optionally_renamed_code>" represents "<code> [as <name>]". A name may be assigned to reference the created community code instance directly. Here, a code is specified

by "code <name>", or a code category "category <code_categorisation>". The "<particle_group_list>" specifies to which group of particles the particles should be added to (e.g. "particles", "dm_particles", ...), followed by a list of particle expressions. Particle expressions may be just a name of a particle or particle set, but can also be the addition or subtraction of particles. This is used for including or excluding particles of the set when adding to the community code. For example, we can have:
"code Huayno as gravity on particles bodies - moon, earth".
Where we reference the "Huayno" code and give it the name "gravity". The particles that the community code acts upon are the sets "earth" and "bodies" excluding "moon".

Bridge application. Applying a bridge is achieved by writing:

```
apply <bridge> for <quantity> with error <error_var>
    [<literal code>]
    evolve
    [<literal code>]
    copy
    [<literal code>]
```

Here, "<bridge>" is the name of the bridge that is applied. The "<quantity>" that follows specifies the simulated time the simulation should run; this could be, say, a year ("1 yr"). Next, "<error_var>" specifies the variable that contains the error of the model, this variable should be updated in one of the literal code sections that follow. The first literal code section is executed before evolving the bridge for a given fraction of the model time. Just before copying the values from the community codes back into their original particle sets, we may add a stopping condition in the second literal code section. After copying the values, the user may want to print some information to the console, this can be performed in the third literal code section. Of course, the user may deviate from this usage of the literal code sections, but these suggestions are based on the usual case. We chose to use literal code sections here, because the custom user code may differ greatly from one use to another, there is no generalization possible that does not restrict the user. In the future we will add a solution that blends the literal code in with the rest of the AMOOSE statements, perhaps by means of callback functions.

5 Compiler Implementation

We implemented AMOOSE by developing a prototype of a compiler we named ANTLER. ANTLER transpiles AMOOSE to Python code in conjunction with the AMUSE framework. The architecture of ANTLER resembles that of a traditional compiler, but as we compile source-to-source (transpile) as opposed to source-to-machine code, we will incorporate models, abstract representations of a desired system, on the back-end and more components attached to the pipeline that handle these components. There are a total of three lexers and three parsers, one of each for the AMOOSE code, one of each for reading the configuration file and one of each for reading templates, which are parameterized specifications (stored in a JSON file) of what target code is output for a given models. Variables that are declared in the AMOOSE code, are stored in a symbol table that is able to handle re-declarations of identifiers, as this is possible in Python as well. We want to stay faithful to the features of Python as it will help us create cleaner target code in the future. This poses specific design considerations for the code generator. The AMOOSE code and configuration are handled by an intermediate code generator, which is able to create a sequence of simple intermediate statements and abstract intermediate statements, which resemble models. Models require additional information in the form of templates, which we have mentioned before. The AMUSE code generator manages the back-end of the compilation process; it resolves temporaries, uses a model handler that processes abstract intermediate operations and uses metadata about community codes to generate their imports and usage.

5.1 Tools

We will develop our transpiler in C++ rather than Python, since the compilation process of the C++ language performs checks that are not performed when interpreting a Python script. The one that is most important to us is type checking; Python only performs this check at runtime if explicitly stated or when the code is executed. We prefer these checks to be performed at compile time, so that the program is more reliable when deployed, we want to know these errors beforehand. Despite Python providing simpler tools for development, C++ supports the same kinds of tools, only requiring a little more effort. C++ also gives the developer more control over memory management and generally has a much better performance. We will not be using the C language, since C++ offers Object Oriented Programming (OOP), which simplifies developing the abstract structures we will be defining. We will be using Flex as our lexical analyzer and Bison for the generation of our parser [13]. We make use of the JSON library for C++ by Niels Lohmann [14] to deserialize the JSON structures we read from different files.

5.2 Overall Architecture

We will describe the specific architecture in the remainder of this section, but there are some notions for the general concept we need to describe first. The final output files contain the *final target code*. If we generate Python code files in between for intermediate use (by calibrating), we call these *intermediate target codes*. The input code to the program is called the *source code*. As

shown in Figure 6, our transpiler parses both a configuration and a source code file. This separation is performed as configurations are *machine dependent*, as we would like our code to be separate for the reason of sharing *machine independent* astrophysical system setup implementations. The fields shown in blue represent intermediate data structures, those in orange show compilation stages. Lexers generate token streams, parsers generate abstract views of the information contained in these streams. Parsers may have a symbol table for later access to temporaries. As mentioned before, the front-end outputs an intermediate, made up of a mix of operands and abstract models; these are later used by the back-end. In Figure 8 we see the compilation of templates into the template object intermediate, representing the information for all available tokens. Figure 7 shows the back-end of ANTLER, with the same color coding. As the models are handled by the model handler, by filling in values to the corresponding templates, more operands are generated on the back end. To reference information about prior models, a global context is used to store and retrieve attributes of these models. Operands are processed into the final target code while referencing variables and resolving temporaries from the symbol table.

5.2.1 Symbol Table

Traditionally, symbol tables store information about the name, line of declaration and type of variables. However, this can be extended to contain much more meta-data. In our case (Figure 6 and Figure 7), we want to be able to support redeclarations in the Python language, thus we built a somewhat different type of symbol table. Our symbol table maps the name of a variable to a set of lines at which the corresponding variable has been declared, referencing a variable has to be done by its name and the current line of interest. We implemented this set using a binary tree, with a simple algorithm retrieving the correct declaration. We always look for the latest line of declaration, which number is smaller or equal to the current line number. This works as conditional statemenets are not yet considered, where declarations may or may not be considered during execution time. In our binary representation, an infix search is easily implemented, retrieving the correct variable and its meta-data.

5.2.2 Intermediate Representation

For the intermediate representation, we need to strike a balance between the abstract representation given in the AST and a general structure that can be compiled to any other language relatively easy. Therefore, we need to split up the abstract operations into smaller general operations that we see in everyday languages. However, since we want to generate readable code, the operations may be nested to allow for multiple operations per line; this is contrary to a single operation, or instruction, per line in regular compilers. This eliminates a huge number of temporary variables that are used in other compilation techniques, for say, compiling C to assembly. For these languages, every operation may take two inputs and often stores its value in a temporary variable that is created on the fly. For us, this means that we may compile to another language in the future and allow for optimization techniques that are performed in this intermediate language.

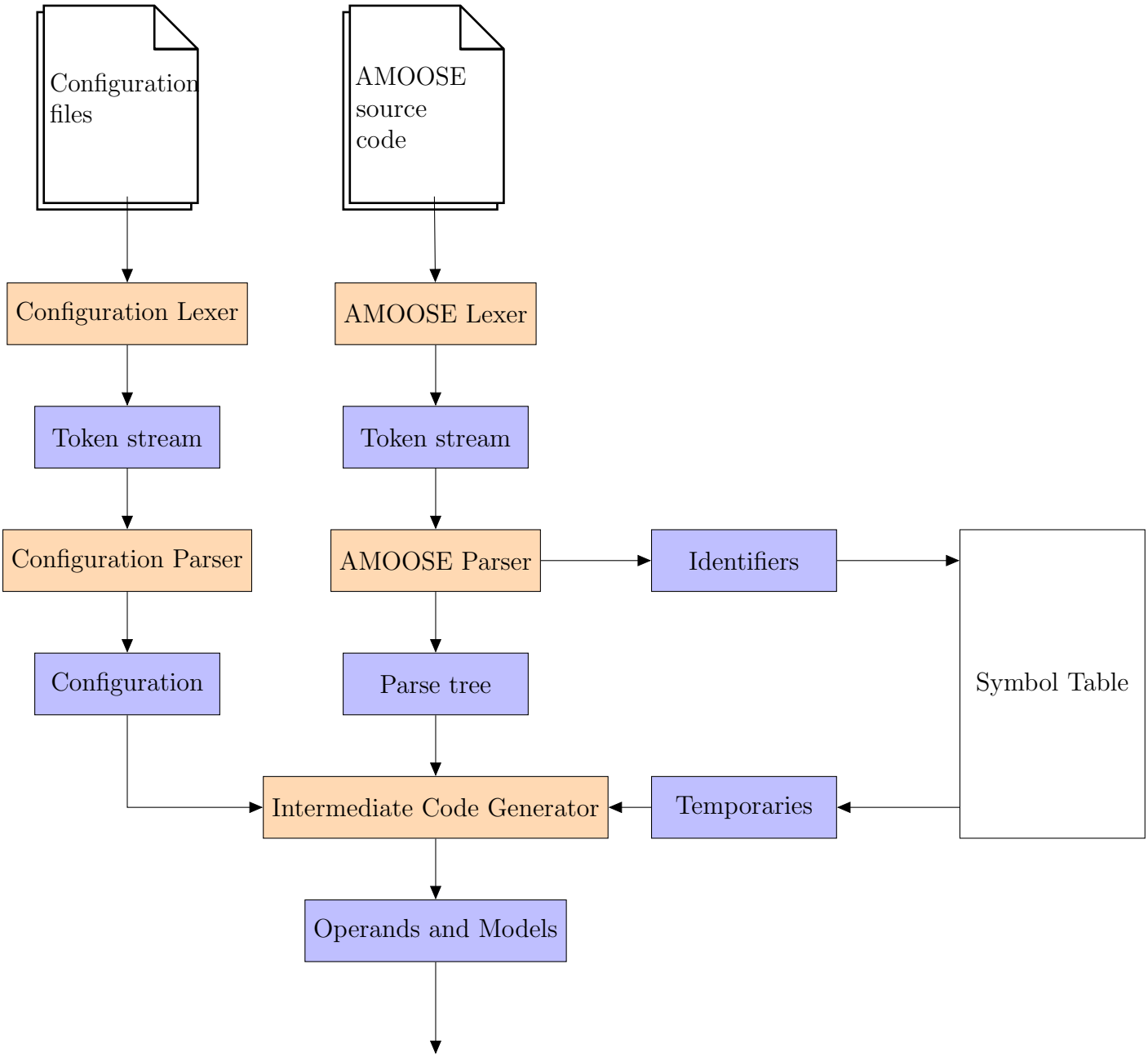

Figure 6: Front-end of ANTLER.

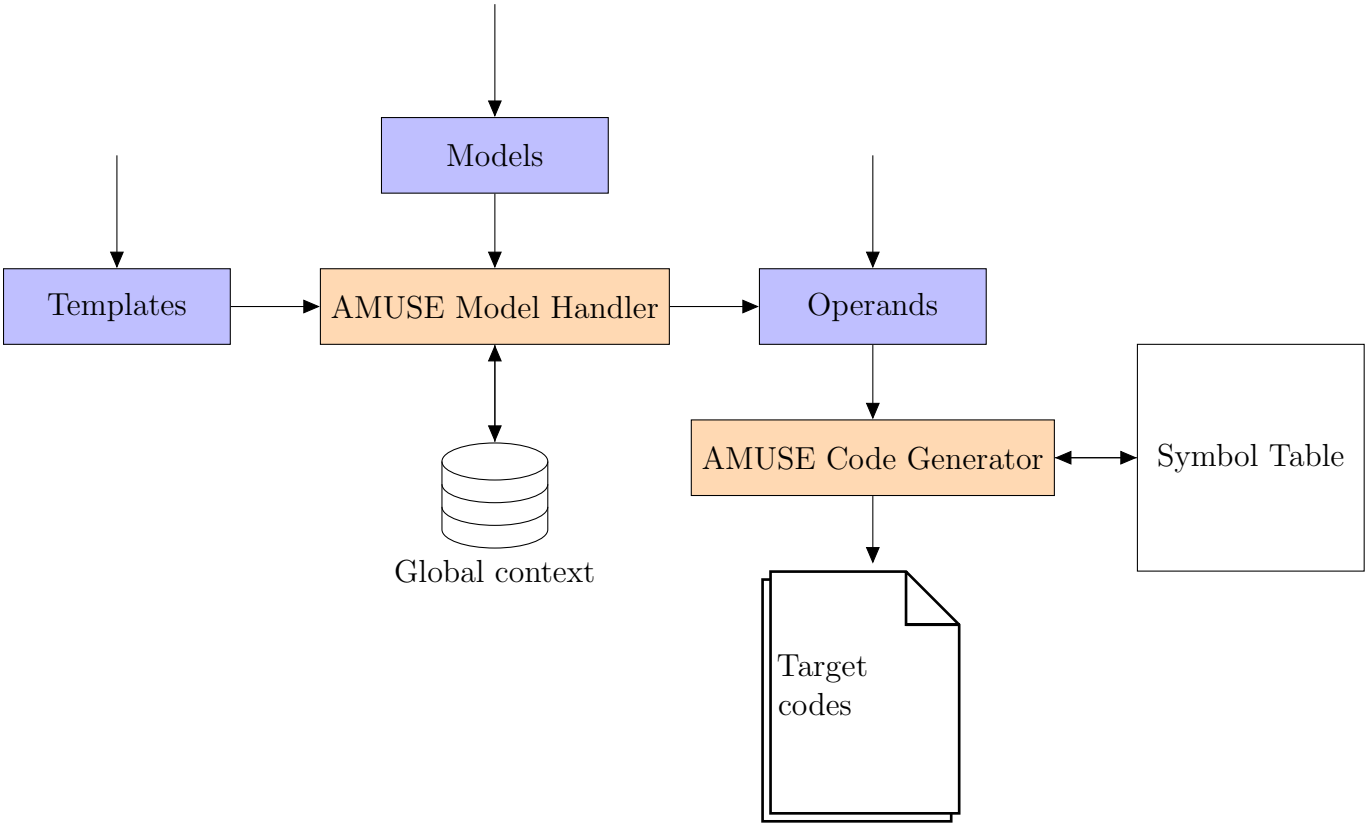

Figure 7: Back-end of ANTLER.

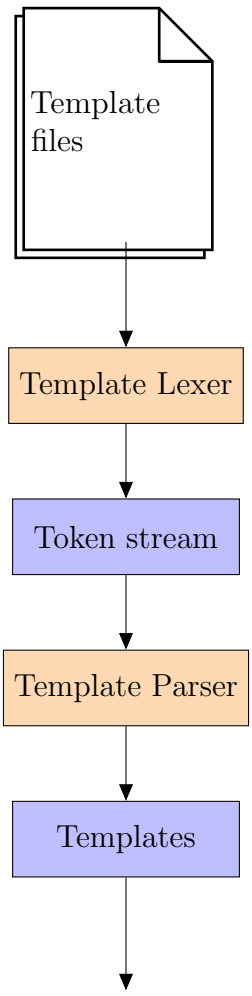

Figure 8: Template handling part of the front-end of ANTLER.

The operations and operands in our intermediate language should indicate no traces of AMUSE code, but should be general. So, "create a particle set" is interpreted as "Call the Particles function (which is the constructor for the particle set) with some arguments (the size of the set) and store the result in a variable". Since we do want to treat units and quantities as basic types, we assume these are present in the target language, so that we can use these types in the intermediate language.
The most important reason we want to generate intermediate code, is to facilitate a more effective walk through the representation for generating multiple target files. Whenever an operation is found that results in multiple target files, we interpret the remainder of the representation for multiple outcomes. This boils down to walking through a mostly linear representation in a backtracking style method.

5.2.3 Code Generator

Code generation (Figure 7) is performed on abstract models that hold a reference to a template specification and properties. The template reference retrieves a recipe for the correct usage of all the properties provided. The target code, generated from the abstract models, is concatenated with that generated from the operands, with respect to the original order.
A certain state in the compilation pipeline is called a branch, where we may branch off of a particular branch by creating a copy of its state. We do so to finish the compilation pipeline for sampled concrete configuration values, for the purpose of calibrating parameters values. For a given parameter, we branch off, generate multiple intermediate target codes, run an analysis on the results and return to the prior branch with the found value. The main branch of our compilation pipeline generates a final target code and branches are kept on a stack.

5.2.4 Invocation

The transpiler can be invoked in the following way:

```
amoose <AMOOSE file.moo>
    [-c <config1.cfg>[,<config2.cfg>]*]
    [-o <output.py>]
    [-v|-b]
```

First, the AMOOSE file is the source code file that the user wants us to compile; usually, we make use of the "`.moo`" extension. The "`-c`" flag is followed by one or more comma-separated filenames of configuration files. The "`-o`" flag is followed by a path to the output file, which the output target files will use as a basis for writing back the final target code. If the file already exists, we append the filename with a number that causes the filename to be unique in the indicated directory. The "`-v`" and "`-b`" flags stand for "verbose" and "brief" respectively, where "verbose" causes debug information to be shown to the user, "brief" causes no debug information to be shown.

Two JSON files are supplied in the data folder, where "codes.json" describes all the community codes and "models.json" describes all the templates used in the generation of the AMUSE code from the intermediate code for models. These JSON files are entry points for maintainers of AMUSE. As AMUSE accumulates more functionality and community codes, these pieces of information can be added to these JSON files, so that ANTLER can make use of the added AMUSE implementations.
The optional configuration files that come with an AMOOSE script need to be parsed into a structure that can be used during the compilation process. Furthermore, the templates found in the model's JSON are also parsed using a different parser, this is done before the compilation process. These parsed template descriptions are used in the last compilation stage, when we generate the intermediate and final target codes. Lastly, the AMOOSE code is parsed using its corresponding parser. We will describe these three parsers next.

5.3 Parser

The first stage of the compilation process starts with the lexical analyzer (Figure 6 and Figure 8), Flex. It tokenizes the raw input string, which is read from the input file. This token string is parsed by the parser generated by Bison, the parser generator we are using. The token string is now represented in an AST or other abstract representation, depending on the kind of resource. In an AST, we refer to nodes with children as "internal nodes", otherwise we call them "value nodes". The children of an internal node are arguments of the represented structure. For instance, the two children of an addition represent the left hand side and the right hand side of the addition.

5.3.1 Parsing configuration files

In Listing 14 we give an example of a configuration file. We specify parameters for two variables that may be present in our AMOOSE code. If the same parameter is set twice for the same variable identifier (either in the same file or in another configuration file that is specified), the user is warned about the collision. The "`eps_is_h_flag`" parameter is set to "`False`" for "`hydro`", the "`gas_epsilon`" is set to "`0.1 | units.au`". For "`hydro2`", "`integrate_entropy_flag`" is set to "`False`" and "`timestep`" is set to "`0.01 * Pinner`", where the user is responsible for defining "`Pinner`" in the code.
The tokens used in a configuration file (Figure 6) are either identifiers (CID) or parameters (CPARAM). CID consists of at least one letter, which includes underscores ("_"), and may then consist of a mix of letters and digits. A configuration program consists of a list of configurations, or may be empty. Each configuration starts with a CID we will call "`variable`", followed by a colon. The identifier is used to match a variable in the AMOOSE code by name. On the next line, another CID we will call "`property`" is found, followed by a space (" ") or a tab ("`\t`") and a character string we will call "`value`", which does not directly start with a colon and ends with a newline. The property is separated from the value and describes which parameter of the variable is set to the extracted value. Until we find another line containing an identifier followed by a colon, or until the end of the file, we read parameters for the last specified variable identifier.

Listing 14: Example configuration file.

```
hydro:
        eps_is_h_flag False
        gas_epsilon 0.1 | units.au

hydro2:
        integrate_entropy_flag False
        timestep 0.01 * Pinner
```

5.3.2 Parsing model files

The "models JSON file" contains two sections: macros and models that contain templates (Figure 8). Each model is given a name, by which it can be referenced in the AMOOSE code, as seen as for "`BinaryOrbitalElements`" in Listing 15. Each model has its own sections for: variables, template and include statements. The variables are a mapping from a variable name to its properties. A variable name may be prepended by a pound sign, making it a meta-variable, or meta-parameter. Meta-variables differ from ordinary variables as they represent information that is not supplied in the AMOOSE code explicitly by an attribute name. For example, a particle can be given a name in the AMOOSE and AMUSE code, which assigns a string representing the given name to an attribute called "`name`". This particle is stored in a variable, which has its own name. But "`name`" is already the way we would reference the attribute "`name`". To eliminate this collision however, a variable or attribute name can not include a pound sign in neither language. So in this situation, it is safe to use "`#name`" to reference the variable representing the particle, which happens to have an attribute called "`name`". We would like to store the name of the variable in "`#name`" (the meta-parameter), as we need it in the template to reference this variable.
The properties of the variable depend on the type of a variable, we have types "`variable`" which we will call "`pure`" for now, "`series`" and "`collection`". A pure variable just represents a single value. It must have a "`required`" attribute set to "`required`" to force a check to make sure that the variable is supplied by the user. (We do not perform this check yet, we will in the future, but more on this in the discussion). A "`default`" property is set to a default value for when the variable has not been set in the AMOOSE code. These can be combined using a "`default_include`" property, which specifies modules or packages that should be included in case the default value is used. A "`series`" represents an ordered list of elements, such as "`#particles`" representing a list of particles. Series also have a "`variables`" property, where the variables of the elements are specified. Furthermore, the "`variables`" of the "`series`" should include a "`#size`" property, so that we can make use of the size for iteration. "`collection`" represents a set of variables which can also be referenced directly, the collection only provides a wrapper for representing all contained variables at once. These variables can be of type "`kwarg`", in a collection called "`kwargs`", for the use of arguments to a function call for creating a particle set of some kind. Variables of type "`kwarg`" may be left out, if they are not, referencing the collection as a whole will result in a comma separated list of all present

variables.
A template string contains both constant sequences of characters (TMPLT_TEXT) and variable parts, which make the template parameterized. The template string contains "${" to open an expression (TMPLT_EXPR_OPEN), "}" closes the expression (TMPLT_EXPR_CLOSE), these markers always come in pairs and expressions can be nested. A counter is incremented when an expression is opened and decremented when one is closed. The counter being zero indicates we are reading template text text. If at the end of the template the counter is a non-zero value, an error is reported, as one or multiple the expressions were not closed. When a newline is found, the template starts a new statement, therefore, our representation will consist of statements that contain separate parts of the template statement, either template text or expressions. Template expressions can either be atoms or macro expressions. A template atom is either a primary, like numbers (TMPLT_NUM) or an identifier (TMPLT_ID), an access or an index. TMPLT_NUM consists of at least one digit, TMPLT_ID follows the same regular expression as CID, but can be prepended with a pound sign ("#"). This pound sign is what makes variables "meta-variables" or "meta-parameters". An access is depicted by a concretization of the general "`object.property`" and represents targeting a property of a specified object, by using the dot sign "`.`" (TMPLT_DOT). An index, depicted by "`object[n]`", takes the n^{th} element from an object, where the object is followed by a square-bracket enclosed "`[`" (TMPLT_SQ_OPEN), "`]`" (TMPLT_SQ_CLOSE) index. A macro expression / macro call, depicted by "`macro(arg_one, arg_two)`", calls a macro called "`macro`" with the comma ("`,`") (TMPLT_COMMA) separated arguments "`arg_one`" and "`arg_two`", enclosed in round brackets "`(`" (TMPLT_RB_OPEN), "`)`" (TMPLT_RB_CLOSE). These arguments can be of any amount, including zero. To repeat some sequence of statements for elements of a certain collection, we use a for-loop. The sequence of statements is enclosed in an opening statement and a closing statement. The opening statement, depicted as "`${for i : #particles.#size}`", opens the template expression. The user specifies the creation of a for-loop, using the keyword "`for`" (TMPLT_FOR), chooses a name for the cursor that will be used for indexing values "`i`", adds a colon (TMPLT_COLON) to seperate the iterator from the next expression "`#particles.#size`" and closes the expression. We iterate using the variable "`i`" from 0 up to (exclusively) the value of `#size` of `#particles`. So, if there were 3 elements in particles, we would see the values for "`i`" being 0, 1 and 2. The for-loop is ended by another isolated statement, depicted by "`${endfor}`", ending the for loop. For-loops can be nested, in which case each combination of iterator values is seen exactly once, regarding their upper-limits.
The "`template`" section is a list of strings that make up the whole template. This template is parsed as we have explained before. The "`includes`" section is used to specify any imports of modules and packages that should be performed to support the template code. The "`macros`" section is similar to the "`models`" section, an example can be seen in Listing 16 for the macro "`around`". It contains a map of names of user defined macros, paired to a template property, in the same manner as for models, and an "`argc`" property. The "`argc`" property maps to a number, which represents the number of variables used. The arguments used in the macro call are referenced by position, so if we reference variable "`0`" in the macro template, we use the first

Listing 15: JSON description of the model “BinaryOrbitalElements”.

```
"BinaryOrbitalElements": {
    "variables": {
        "#name": {"required": "required"},
        "#around": {"required": "required"},
        "mass": {"required": "required"},
        "semimajor_axis": {"required": "required"},
        "G": {
            "type": "kwarg",
            "required": "none",
            "default": "constants.G",
            "default_include": {
                "module": "constants",
                "package": "amuse.units"
            }
        },
        "kwargs": {
            "type": "collection",
            "eccentricity": {"required": "none", "type": "kwarg"},
            "true_anomaly": {"required": "none", "type": "kwarg"},
            "inclination": {"required": "none", "type": "kwarg"},
            "longitude_of_the_ascending_node": {"required": "none", "type": "
               kwarg"},
            "argument_of_periapsis": {"required": "none", "type": "kwarg"}
        },
        "#particles": {
            "type": "series",
            "variables": {
                "#name": {"required": "required"},
                "#size": {"required": "required"},
                "mass": {"required": "required"}
            }

        }
    },
    "includes": [
        {
            "module": "new_binary_from_orbital_elements",
            "package": "amuse.ext.orbital_elements"
        }
    ],
    "template": [
        "${#name} = new_binary_from_orbital_elements(${comma_seperated(#
           particles[0].mass, #particles[1].mass, semimajor_axis, G, kwargs)
           })",
        "${for i : #particles.#size}",
            "${#particles[i].#name} = ${#name}[${i}]",
            "${attributes(#particles[i].#name)}",
        "${endfor}",
        "${attributes(#name)}"
    ]

}
```

Listing 16: JSON description of the macro "around".

```
"around": {
    "argc": 2,
    "template": [
        "${0}.move_to_center()",
        "${0}.position += ${1}.position",
        "${0}.velocity += ${1}.velocity"
    ]
}
```

argument, for "1" we use the second, and so on.

5.3.3 Parsing AMOOSE files

Our compiler divides an AMOOSE script in the different types of statements from the AST, which we have described in the design of AMOOSE (Section 4). For this division, the parser is able to determine which lines an abstract model (such as a particle set creation) spans, when considering indentation. Recognizing indented blocks is accomplished in the lexical analyzer by adding states to the tokenization process. We memorize the indentation level of the previous line and we start in a state where the indentation level of the current line is counted. If the first token other than indentations is found on a higher indentation level than that of the previous line, then we place back the token and yield indentation markers instead. This is simple, as indentations only grow at most one level at a time. However, if we exit multiple indented blocks at once, we would have to output a corresponding number of dedentation markers at once. By placing back the symbol that is read and yielding dedentation markers instead, until the indentation level is compensated for, we can still achieve indentation. Indentation and dedentation markers are pairs and should be appropriately placed. As we want our indentation and dedentation markers to encapsulate the appropriate block, the last newline marker should come after the dedentation marker. However, we only detect dedentation after the newline is read. We store the newline marker for when we are done generating all dedentation markers.

5.4 Configuration

One of our insights is that the generated AMUSE code is machine dependent, whereas the original AMOOSE code can be machine independent. Different machines have different specifications, and so exhibit different execution times for the same Astrophysical Multipurpose Software Environment (AMUSE) code. We may want to tweak some parameters and loosen the accuracy criterion (for slower machines), in order to end up with a decent performance for different machines. We would like to make these modifications without without changing the AMOOSE code, therefore AMOOSE allows for the separation of parameter values into a configuration file (Figure 6). With

this feature, we end up with a machine independent AMOOSE code and a machine dependent configuration file.
While parsing the configuration file, we may create ConfigNodes with different ConfigNodeTypes (CONFIG_UNKNOWN, CONFIG_IDENT, CONFIG_PARAM, CONFIG_VALUE, CONFIG_PARAM_LIST, CONFIG_IDENTIFIERS). These ConfigNodes are stored in a ConfigTree.
We implement a Configuration class, that holds a mapping from the name of an identifier to a set of parameters, which is a map from the name of the parameter of the identifier, to a value supplied by the user. Since we allow for multiple configuration files, collisions may occur for parameter values on a parameter of an identifier. These collisions are reported, the first value remains.

5.5 Front-end Structure

Our AMOOSE parser converts the token stream supplied by the AMOOSE lexer into an AST. This tree consists of nodes that have information about the line number they originate from in the source code, the type of node they represent and the type they would return if used. With the node type, we can differentiate between the different operations and structures the nodes may represent. With the return type, we can differentiate between return types the nodes may have, such as representing numbers or a particle set. Nodes can be further classified as either being a ValueNode, which store a value in the form of a string (as we read strings directly from the source code), and InternalNodes. An InternalNode may contain any amount of children, usually it has at least one child.

5.5.1 Symbol Table

The symbol table we have built incorporates a feature where identifiers are stored in combination with the line numbers they are declared on. This information may need to be preserved in future steps where we want to abstract away from AST walks; a mapping is much easier to search through than an AST. Python allows for redeclaration of identifiers, which means that a variable may represent different types at different lines in the same scope. So when we need the return value of a certain variable, it is necessary to know at which point in the code the variable is referenced. A simple method occurs where given the line number of a node and the name of the identifier, we look up the list of line numbers the identifier is (re)declared on. From this list, we obtain the symbol information corresponding to the closest line number that is smaller or equal to the supplied line number. The symbol information consists of the identifier name, line number of declaration and the return type of the identifier. The symbol table consists of the mentioned hashmap from identifier name to a LineLookup tree and another hashmap that maps temporary variable names to resolved variable names. A LineLookup tree stores a binary search tree of LineLookupNodes, in which an inorder walk yields the line numbers in ascending order. We can make use of this information to quickly find any line number that is smaller or equal to the specified line number. The symbol table also has a mapping from the names of particle(set)s to the particle sets they are contained in, this is useful when copying channels.

5.6 Intermediate generator

In Figure 9 we show the generation of the output code from the source code regarding abstract models. The model template name is specified by the user when creating a model, the properties that are specified are filled in into the referenced template, this results in a snippet of output code, later merged with the output code generated from operands and other abstract models.
The intermediate code representation consists of IntermediateStatements. These statements consist of a type and can be further classified as either being PrimaryIntermediateStatements or AbstractIntermediateStatements. PrimaryIntermediateStatements describe singular operations with any number of arguments. These intermediates represent singular operations that directly correspond to operations found in the input code. Each of the arguments may either be a list of IntermediateStatements, a singular IntermediateStatement or a string. This structure allows for intermediates to be nested, an unnested intermediate language like three-address code will straighten out the operations, exactly one per line. AbstractIntermediateStatements represent abstract models, we do not process these into seperate statements yet, as this may only be done in the background. Another output language may want to decide to use other sequences of statements to configure such an abstract model, than the prescribed sequence we would have to generate in this step, otherwise. Thus, the definitions are given in the template JSON. AbstractIntermediateStatements hold information about the name of the model used and a set of arguments that form the model configurations. PrimaryIntermediateStatements are created by a PrimaryIntermediateStatementCreator, which holds helpers for all structures of IntermediateStatement types. This creator also has a counter that is used for creating unique temporary variable identifiers. The IntermediateCodeGenerator is a PrimaryIntermediateStatementCreator and can reference the Symbol Table. This generator processes the AST nodes in sequences of IntermediateStatements.

5.7 Abstract Model

Since the front-end (Figure 6) may not specify or make use of details that are limited to the back-end (Figure 7), we implement a few abstract classes that provide the front-end with target language independent procedures that construct descriptions of a model. By inheritance, we can elaborate on these procedures, so that we can define the behavior for a specific target language. A generator class written for the front-end would be extended in functionality for generating code in the appropriate target language by making use of inheritance. These classes are fundamental to the usage of templates for code generation, we must read templates from ordinary JSON files and structure the definitions in memory in some way, regarding to both the intermediate and target language.
We need a representation of a model, a representation of the template code and its parameters, a way of indexing these template parameters, a structure that contains the actual model arguments (the values filled in for the template parameters), and a way of passing on information from one model to the other. In Figure 9, in the middle we see a representation of a model "Binary". A more exact view is given in Listing 15.

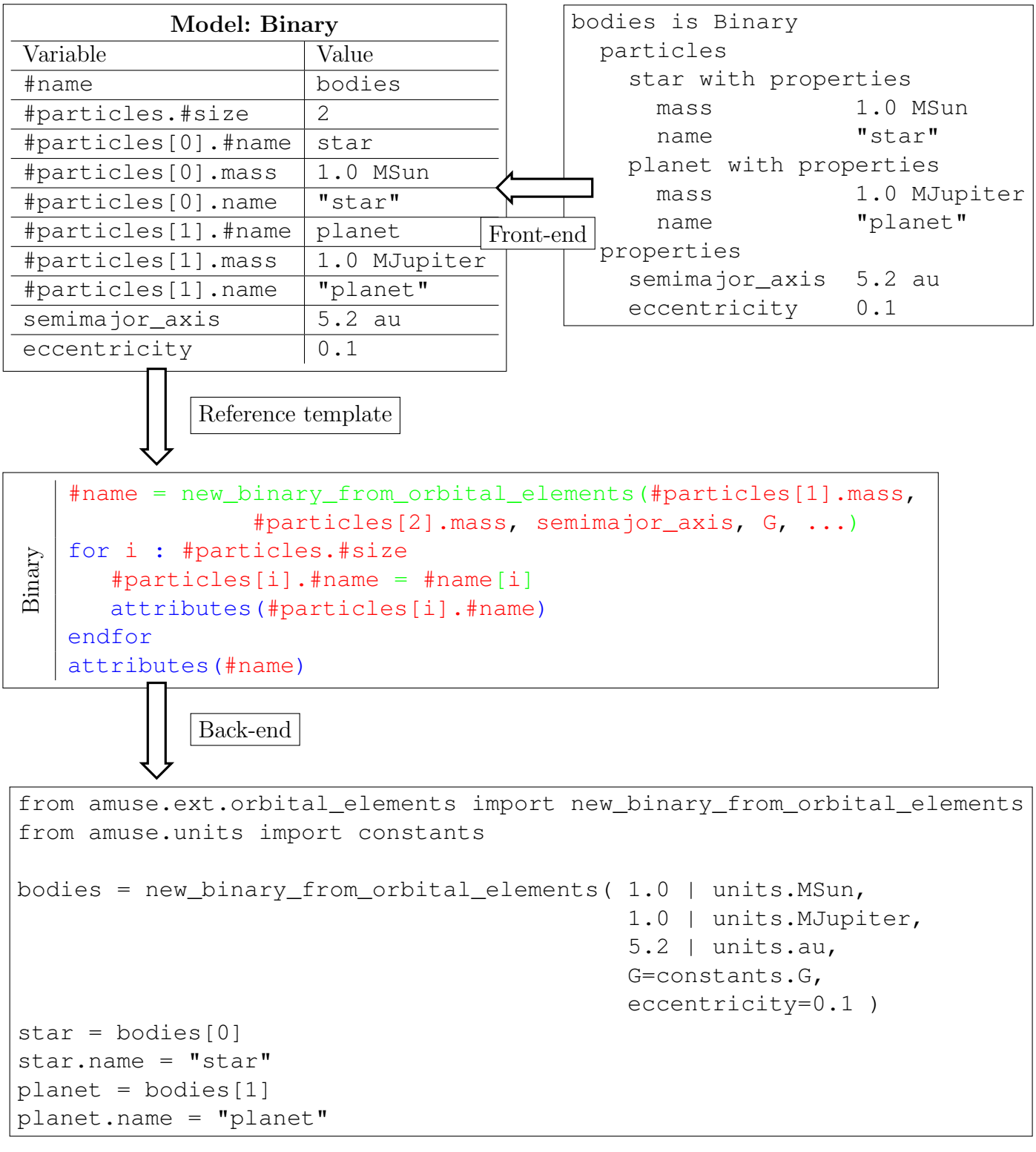

Figure 9: Generation from abstract models.

5.7.1 Models

We start at the Model class, which represents a single model from the model's JSON. Each Model is defined by a ParametrizedTemplate and TemplateParameters. The ParametrizedTemplate consists of a sequence of ParametrizedTemplateStatements. A ParametrizedTemplateStatement contains ParametrizedTemplateParts, making up the statement. A ParametrizedTemplateStatement may also be a ParametrizedTemplateStatementLoop, a statement representing a for-loop. This type of statement also contains the name of the iterating variable and an expression representing the set we iterate through. For each iteration of the set, using the iterator variable, we generate a sequence of statements, which are also contained in the ParametrizedTemplateStatementLoop class.
Each statement consists of ParametrizedTemplateParts, of which there are two types. First, there is the ParametrizedTemplateStringLiteral, which is a constant string of characters that make up the static parts of the template. Next, there are the ParametrizedTemplateExpressions, of which there are four types. These expressions make up the parameterised part of the template. The first expression type is the ParametrizedTemplateExpressionId, which represents an identifier corresponding to a model argument that may or may not be supplied. Next, there is the ParametrizedTemplateExpressionIndex class, which represents indexing some collection resulting from an expression, using the result of another expression. We can use the ParametrizedTemplateExpressionNum, which represents a number, the ParametrizedTemplateExpressionAccess class, which represents obtaining a property of some object, and lastly the ParametrizedTemplateExpressionMacro. The latter represents a macro, which can be called similar to a function, given the name of the macro and a set of arguments. These macros can be defined in the model's JSON in a section called "macros", but we also provide predefined macros, which we will explain in Section 5, when we describe the AMUSE generator. These predefined macros however, are important, as we have not implemented a way of giving full control over the model handler in the model's JSON. We intend to do so in the future, as well as supporting lists and operations to manipulate them, quite like we see in LISP. In LISP, we can take the first element (head) off of a list and recurse on the rest (tail). This helps users create powerful template definitions. Also, we want to add conditional statements to the templates and explore the possibility of making the template language Turing complete.
The TemplateParameters class contains a set of TemplateParameter objects. A TemplateParameter contains the name of the parameter, such as "`#particles`", a type, sub-parameters and properties from the JSON file. The three types of TemplateParameters are Variable, Collection and Series. A Variable TemplateParameter has no sub-parameters, it is only a variable with some properties. Such properties may be e.g. the default value of the argument when none is given. A Collection, such as "`kwargs`" is a set of variables, which are each individual parameters, but may be referenced as a group by the name of the Collection. This makes it easy to reference all arguments to a function in the template as one, to concatenate them, separated by commas. A Series TemplateParameter may have sub-parameters, such as a list of called "`#particles`", where each element (each particle) may have a "`#name`" and/or a "`mass`". In Figure 9, a model "Binary" is shown, containing a parameterized template. We see variables and meta-variables that are required from the model arguments, shown in the top left. All strings shown in red represent model

parameters (ParametrizedTemplateExpressions), these are filled in by using the model arguments. The parts shown in green are constant string literals (ParametrizedTemplateStringLiteral) that are copied straight to the output code snippet. The parts shown in blue make up either the for-loop statement or represent macro calls to the "`attributes`" macro, which generates the configuration of all unused attributes that were supplied by the user.

5.7.2 Model arguments

In order to index these TemplateParameters (and later the ModelArguments), we make use of the ModelArgumentKey type. This key consists of an ordered list of ModelArgumentKeyParts, which each contain a value and a type. For example, when we want to store a key "the mass of the first element in the '`#particles`' collection", we get a key consisting of the ordered values ["`#particles`", "0", "`mass`"]. The first and the last value are of the Identifier type, the second element is of the Index type. If we do not care about the type, we can specify this using type Any. This helps when looking for a specific argument, based on the values only. Since TemplateParameters do not care about indices (it only knows that the elements in "`#particles`" have properties with the name "`mass`"), we would leave out the second element in the key of the previous example. For example, in Figure 9, we see an object of model arguments for the "Binary" model, where the keys of the variables are shown in the left column, their corresponding values on the right. These keys and values were obtained from an abstract model from the source code, by the front-end.
The ModelArguments class contains an ordered list of arguments, these are key-value pairs of a ModelArgumentsKey and ModelArgumentValue. The ModelArgumentValue consists of a flag that tells whether the argument has been used and a parsed statement as the value of the argument. We do not store these lists of pairs as maps, as the order matters. Namely, when the user specifies a couple of argument values in a specific order, dependencies may arise which will be broken when the order of the argument values are stored in an unordered way. We are able to request the remainder of the arguments of the ModelArguments object, the elements which have not been used yet. These are likely to be additional arguments that may be supplied, regardless of the template definition.
The ModelHandler holds the model and macro definitions using the Model class. It reads the model JSON file and creates these Models (or any inherited class as we provide a polymorphic entry point). For this, it parses each template using the template parser. It also reads the JSON structure into TemplateParameters for each Model. Note that the parameters and arguments corresponding to a model describe a *local context* of the parameterized template. In other words, the values used in the parameterized template are expected to all be present, when disregarding the fact that we may reference values from another model, another local context. We need a *global context* to propagate a value from a prior model, as we are not able to access these values if not made public.

5.7.3 Local and Global context

The GlobalContext (Figure 7) class contains the configurations from the configuration files and global declarations. When a model has been handled, the parameters from the local context should be added to the global context. In the local context, the name of the identifier referencing the model is implicit. However, we need to add this name to all ModelArgumentKeys in the global context. We also need to resolve any other ambiguities, such as meta parameters and variable indices. All in all, we should be able to pinpoint the exact variable referenced, as we would do in the target code. The ModelArgument already has resolved indices for Series, it also contains the "`#name`" meta-parameter for most objects. This means that we can look up the value for "`#name`", as the name of a model or of an object that is part of it, complement the ModelArgumentKey used in the local context and add it to our global context.

5.8 AMUSE Generator

The aim of the AMUSECodeGenerator is to convert the intermediate representation into the final target code, while possibly invoking the calibration of parameter values. The AMUSECodeGenerator implements simple functions for the generation of PrimaryIntermediateStatements. The generation of models is more sophisticated, as it will use another class, the AMUSEModelHandler. The AMUSECodeGenerator class has access to the AMUSECodeLookup class for looking up community codes in the JSON file. It also has access to the SymbolTable for accessing variables that have been set earlier, and for determining resolved names for temporary variable names without having a collision. Whenever a temporary name is found, we immediately resolve it, the mapping from the temporary name to the resolved name is stored in the symbol table for later reference. Given that all reserved symbols have been registered in the symbol table in earlier passes, we are able to come up with unique names that are not used anywhere else in the code. The code generator is also given an object of AMUSEImports, for the imports that have been provided in literal code sections. Furthermore, it has access to a stack of CodeGenerationStates, that represent branches.

5.8.1 Imports

Imports in Python are either written as whole module imports, which may be renamed by the programmer:

```
import time
import numpy as np
```

Modules can also be imported from packages, where the module can also be renamed by the programmer:

```
from amuse.units import constants
from matplotlib import pyplot as plt
```

We can also use the asterisk as a wildcard to import everything from a package, so that we can directly reference the package's exposed modules by name.

```
from amuse.ext.composition_methods import *
```

We eventually structure imports coming from literal sections in the source code, from the intermediate code generator and from the AMUSECodeGenerator in the AMUSEImports class. The reason for structuring is to create a more readable output, which does not contain duplicate imports. The AMUSEImports class has a list of unique pairs, consisting of the imported module name and an optional user defined custom name. As soon as a module is given a custom name, however, this does not make the import redundant, as it may be referenced by its custom name instead of its original. We also structure package imports in a mapping from the module to a set of package imports. We keep a Boolean to keep track of whether the "$*$" wildcard is used, in which case, all package imports that do not have a custom-name are obsolete as we would include them anyway, as they are referenced originally. We can easily merge these AMUSEImports objects and add module and package imports.

5.8.2 Community code lookup

At the start of the program, community codes are read from a JSON file, which contains the set of community codes, structured and grouped by their categories. In branching, we perform the same search for every variable outcome. The AMUSECodeLookup class is able to find a community code by name to generate the appropriate import. It can also provide all community codes in a given category, for when the user wants to generate the target code for different community codes. When "`category`" is used, the codes are picked by taking the first code of every single subcategory of the category supplied as the argument by the user. So, for example, if there are 5 different subcategories for "`gravitational_dynamics`" ("`pure`", "`direct`", "`approximate`", "`dedicated`" and "`limited`"), then we take the codes "`aarsethzare`", "`nbody6xx`", "`bhtree`", "`kepler`" and "`etics`", which each correspond to their respective subcategory of "`gravitational_dynamics`". We pick codes this way, because the user is now able to generate 5 target codes at once, one for each category, and get an understanding of which category fits best for the problem. However, we do assume that the single codes that are picked are representative of their categories. Now, generating target codes for all 26 codes for "`gravitational_dynamics`" is simply achieved by writing "`category(gravitational_dynamics.all)`". Getting all codes from the last subcategory of "`pure`" from "`gravitational_dynamics`", follows the same description, by diverging on every code it contains. This means that writing "`category(gravitational_dynamics.pure)`" yields all 7 codes that are in the "`gravitational_dynamics.pure`" category.

5.8.3 Variational code parts

A segment of AMUSE target code, consisting of zero or more statements, is described by a string that contains the raw target code, and an AMUSEImport object that contains all the imports. The imports are stored separately, so that we can merge the AMUSECodeParts, while keeping the imports at the heading of the generated target code. When the user wants to generate multiple target codes for, say, each community code in a specified category, we want to generate different variations of the same code. These variations are maintained in a string-like structure (multi-string) that represents multiple strings at once. When multiple variational code parts are concatenated (in case the user wants to generate multiple target codes, for multiple community code categories), we want to take the Cartesian product of the multi-strings. We have implemented the AMUSEVariationalCodePart, which holds a vector of AMUSECodeParts. This structure manages its concatenation with strings, AMUSECodeParts and AMUSEVariationalCodeParts. Storing the full target code for every variation is a little inefficient, so we should create a linear graph in the future, where each node represent a part of the final code. The graph is segmented in layers, where multiple nodes may exist on the same layer. For all nodes on a single layer, we concatenate the accumulated codes with the code stored in the nodes, by means of a Cartesian product. This makes storing the target codes more space efficient.

5.8.4 Code Generator

Now that we have described the format in which the target code is represented right before the final generation, we describe how the generation actually works. First we need to mention that during the code generation process, we need to consider our calibration feature, where we generate intermediate target codes during the regular compilation process of the final target code. We will focus on the calibration later, but we already need to consider it.
The AMUSECodeGenerator (Figure 9) makes use of the AMUSEModelHandler when a model is found. This model handler has access to the generator for the creation of new branches, that are maintained on a stack. If we want to create an intermediate target code, without adding information to the current environment, we need a copy of the current environment that serves as a sandbox. For example, if we continue our generation with only a single environment, definitions of models and variables will be added. When we return to the line that we calibrated a value for, we will have information about future lines. We may now already find declarations that should be missing, or find updates on existing declarations, which would only be known further up in the compilation process. If the AMUSECodeGenerator detects that a branch has finished generating it is taken off of the stack. The intermediate target codes are calibrated by the AMUSEModelHandler, which adds newly found information to the global context to the current branch. The AMUSECodeGenerator then proceeds working on this branch. Eventually the main branch is fully processed, in which case the final target codes are written to the output files. A more elaborate implementation would be able to process statements that are independent of the calibration only once, as this generation would otherwise be performed for each branch separately. It is not too inefficient, but can certainly be improved in the future.

The CodeGenerationState is an element on a stack of states residing in the AMUSECodeGenerator, each element representing a single branch of the compilation process. Except for the main one, branches lead to the generation of intermediate target codes and perform the calibration from the results of these samples. When the calibration has been performed, the found value is inserted in the GlobalContext of the underlying branch. The CodeGenerationState therefore contains the target codes that have been generated up until now in the current branch, the location in the intermediate code, as given by the index of the next statement, and the AMUSEGlobalContext at the current point in the generation process. When a new branch is pushed to the stack, all values are copied to the new branch. The rule of thumb here is that the bottom branch on the stack is the main branch, from which we may only generate the final target code. All other branches may generate intermediate target codes.
The AMUSEModel extends the Model class with the adaptation of AMUSEImports that represent the imports that are specified in the model template. The AMUSEModelHandler generates target code from models by iterating over all statements (in the case of loop statements, it performs several generations over its statements), and all template parts within these statements. We generate the appropriate target code by resolving variables from either the local or global context, and generate code from predefined or custom macros. At the end of a generation of a model, the global context is updated with model arguments, for which in the keys aliases are used when we could specify variables in different ways: when having a bridge "`gravhydro`" with two community codes "gravity" and "hydro": "Attribute X of the second community code of '`gravhydro`'" or "attribute X of hydro". The predefined macros either create comma separated lists, indent some code, generate leftover attribute configuration statements, generate statements from the configurations of a configuration file for a given variable, register or copy channels, register and use the converter and start the calibration process. When using the latter, like mentioned before, some code is generated to create a sample value of the parameter we are calibrating. Next we also add some code that lets us observe the results. We push the code variations onto a new branch and after the whole process, update the next branch's global context with the value that was determined.
The model handler uses a AMUSEConverterManager, AMUSEChannelManager and an AMUSE-CalibrationManager. These are stored in a global context however. Whenever we generate a model, we only hold a local view of its arguments through the model arguments. However, the premise of our reduction is given by reusing information that was already supplied in the source code, in an earlier model. To allow for the interaction between our models, we store the model arguments as they should be found as global arguments present in the current scope at some given line.
The AMUSEGlobalContext contains the converter manager, channel manager, calibration manager in addition to the general GlobalContext. Their states are bound to the moment of the target code we are currently generating. If we would branch off while using the same Managers, they may pick up information about future code generation. If this happens, then variables are registered as declared, while they are not upon returning to the original branch and the declarations will be ignored, while being required instead.
The AMUSEConverterManager contains the name of the converter and supplies the generation

process with it whenever needed. It also generates code for the declaration of the converter in the target code, together with the imports that it requires.
The AMUSEChannelManager contains the name of the channel object, together with a list of all defined "to-channels" and "from-channels", where the "from-channels" copy the values of particle(set) properties from the particle set to a certain code's working vector and vice versa for the "from-channels". It provides code for the declaration of the channels object, for the registration of the from- and to-channels, which are added to the channels object in the target code, and for copying the values in a bridge, using the from- and to-channels. This function may later be expanded by only supplying the target code that corresponds to the actual community codes being used in the bridge, but for now we use only one bridge at a time, so every community code is relevant.

5.8.5 Calibration

We want to automatically pick appropriate parameter values or even community codes, AMOOSE makes this possible during the code generation stage. This is done by first generating different scripts, using sampled values of different available options. We then execute these intermediate target codes for a portion of the full model simulation time. We collect information about the execution time and simulation error, which are used for the analysis for the selection of the appropriate configuration.
During the code generation stage, we detect whether a parameter should be calibrated by checking whether the local or global context contain a value for the required parameter. If not then the calibration is performed, otherwise the found value is used. The calibration of a parameter is performed by generating multiple target codes with different concrete values for the parameter. Also, additional code is present for measuring the execution time and error, and writing both values to a file. Next, as these scripts are executed using `fork()`, they provide their output files, which can then be analyzed in a Python script. This Python script then creates a final output file, containing the value for the calibrated parameter. This value is then read when finalizing the calibration, by adding the value to the global context of the branch we fall back on, the branch will not go into calibration mode, but will instead use the value as mentioned in the start.
In this thesis, we consider the specific case of the calibration of the time-step parameter. We know that the absolute difference in the energy of the system (error) increases with the size of the time-step. This is because larger time-steps make more inaccurate approximations of the system state, including the variables that determine the energy. Because this is a given, we fix the constraint of the time-step regarding the error to be preferably as small as possible. For the execution time of the model, taking a time-step that is too small will yield a long execution time. This is evident, as a time-step of 0 would result in an infinite amount of steps to finish the model. A time-step that is too large results in a very high error, while it only gives an insignificant improvement on the execution time. A time-step that is infinitely large will result in an execution time equal to the overhead.
We take inspiration from the paper "Auto-Scaling Cloud-Based Memory-Intensive Applications" by Novak et al. [17]. In this paper, the authors describe a method to determine natural thresholds

on an independent variable, given a relation between the dependent and independent variables that is represented by a hyperbola. They do so by finding the intersection of the hyperbola with its Latus Rectum (LR). This value can easily be found when having an equation of the hyperbola in standard form. However, we make use of a multiplicative inverse, for which finding the LR is not as trivial.

In Figure 10 we show how we find the intersections of the hyperbola with its LR, corresponding to the experimentational setup we have described. First, we determine the hyperbola's equation $y = \frac{a}{x+b} + c$, where a controls the curvature of the function, b controls the Vertical Asymptote (VA) of the hyperbola ($VA = -b$) and c controls the Horizontal Asymptote (HA) of the hyperbola ($HA = c$). An initial guess for b is $b = 0$, as this theoretically defines an infinite execution time, like we mentioned. An initial guess for c is the minimal y-value of the points, as it is closest to the value of a constant overhead of the execution time. When filling in b, c, x and y for each point, we can calculate a for each point and take the average as our initial guess. We let the SciPy [28] `curve_fit` optimizer find a best function fit by configuring a, b and c, using non-linear least squares, from our initial guess values. The multiplicative inverse is equivalent to a rectangular hyperbola, the HA and VA are orthogonal. The angle of the Transverse Axis (TA) lays halfway in between these linear functions and cuts through the intersection of the HA and VA at $(-b; c)$. From this information we can determine that the TA is a function of the form $y = a'x + b'$, where $a' = 1$, since this results in a 45 degree angle, and $b' = b + c$. The Knees (or Vertices) of the function are the intersections of the TA with the hyperbola, we are only interested in the knee where x is larger, so at coordinate $(\sqrt{a} - b; \sqrt{a} + c)$. The semi-major axis is the line segment between the intersection of HA and VA and the knee. The Linear Eccentricity is the hypotenuse of the triangle formed by the semi-major axis and the line segment orthogonal to the TA from the knee, with a length of $2\sqrt{a}$. If we take this length from the intersection of the HA and VA along the TA, then we end up at the focus of the hyperbola, at coordinate $(\sqrt{2}\sqrt{a} - b; \sqrt{2}\sqrt{a} + c)$. The Latus Rectum is a linear function ($y = a'x + b'$) orthogonal to the TA (so $a' = -1$, passing through the focus, so $y = -x + 2\sqrt{2}\sqrt{a} - b + c$. The intersections of the LR with the hyperbola are $((\sqrt{2} - 1)\sqrt{a} - b; (\sqrt{2} + 1)\sqrt{a} + c)$ and $((\sqrt{2} + 1)\sqrt{a} - b; (\sqrt{2} - 1)\sqrt{a} + c)$ for the left and right coordinates respectively.

If we set the time-step to the x-value of the right coordinate, then larger time-steps do not significantly improve the execution time. If we set the time-step to the x-value of the left coordinate, then a smaller time-step will significantly decrease the execution time. Thus, if we want to minimize error then the left coordinate x-value for time-step would do so while resulting in an execution time that is manageable. If we prefer a faster program, where we do not care as much about the accuracy, then the x-value of the right coordinate should be chosen for the time-step.

The disadvantage of this method is that when we double the model time, then the y-axis is multiplied by 2. This changes the curvature of the function and will thus result in different thresholds. We may eliminate this problem by using a standard model time as reference point and dividing the y-value by the fraction of the actual model time divided by the standard model time. However, we would have to look into this further and look into this for different systems.

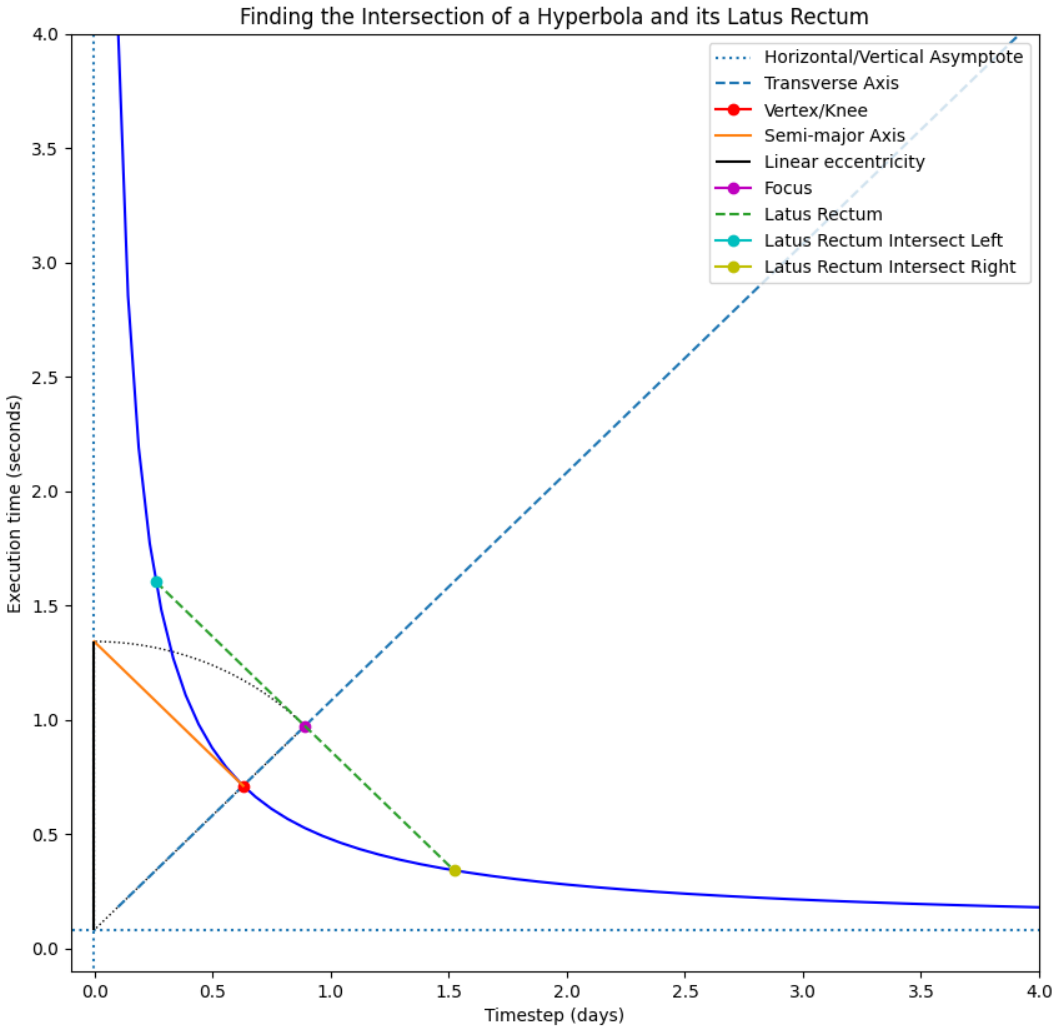

Figure 10: **Finding the intersection of a hyperbola and its latus rectum:** The x-axis represents the time-step in days, the y-axis represents the execution time of the bridge evolution in seconds. The legend contains the names of all markers that are referenced in our method of finding the intersection of the hyperbola with its latus rectum.

6 Results / Evaluation

In this section we show the calibration of the time-step variable as performed during the code generation compilation stage, as explained in Section 5.8.5. Furthermore, we show the reduction in the amount of tokens for one of the tutorial scripts. We believe the picked tutorial script is representative of any simple AMUSE script, as it includes the creation of multiple particle sets, the creation of a bridge and its application: a complete and realistic simulation.

6.1 Calibration

We want to show that a calibration can be performed for the tutorial "AMUSE tutorial on high-order bridge" [21]. We do so by not specifying a time-step parameter in our AMOOSE code, eventually invoking our calibration module.
In Figure 11 we can see that, corresponding to the theory, the error of a system generally increases with the size of the bridge's time-step parameter. We also see that outliers are not uncommon and that these variables do not always relate in a linear fashion. The code executed for this experiment is the intermediate target code that is derived when rewriting the code provided in "AMUSE tutorial on high-order bridge" [21] as an AMOOSE program. We have set the random state of the program to a constant value, to take away the random effect on the error. This data was generated for a bridge that is run for a total model time of 100 simulated days. We determined 10 time-steps by dividing the total model time by powers of two, starting with 2^1. We get the timesteps (in days) $[50, 25, 12.5, 6.25, 3.125, 1.5625, 0.78125, 0.390625, 0.1953125, 0.09765625]$, for which we generate a separate intermediate target code. We run 20 separate repetitions of each intermediate target code, of which we plot the means (as crosses) and standard deviation (using vertical lines). The standard deviation can not be seen in this graph, as all errors were the same due to the fixed random state. This experimental setup also applies for determining the graph of the execution time of the bridge as a function of the time-step. The execution time of the bridge starts being measured just before we enter the bridge loop, we stop right after exiting the bridge loop. We can see the relation between the execution time and time-step in Figure 12. We see that a multiplicative inverse $y = \frac{a}{x+b} + c$, equivalent to a rectangular hyperbola, can be fit well on the mean execution time. From this point, we were able to easily compute the optimal time-step to be about 0.259 simulated days, corresponding to an execution time of 1.604 seconds. Our error of 0.000126 is expected to be very low, since this was our first criterion. Our experiment does not generalize in any quantitative results, however we can conclude that, qualitatively, the manual search is eliminated by automatically determining the non-trivial parameter value.

6.2 Reduction in amount of tokens

We have tested our proof of concept with one of the AMUSE tutorial programs [20] that makes use of a bridge. When looking at the code corresponding to the supported structures in AMOOSE, the required number of lines reduced from 93 in AMUSE to 44 in AMOOSE, the number of characters from 2423 to 1085 and the number of tokens reduced from 1215 to 432, a considerable

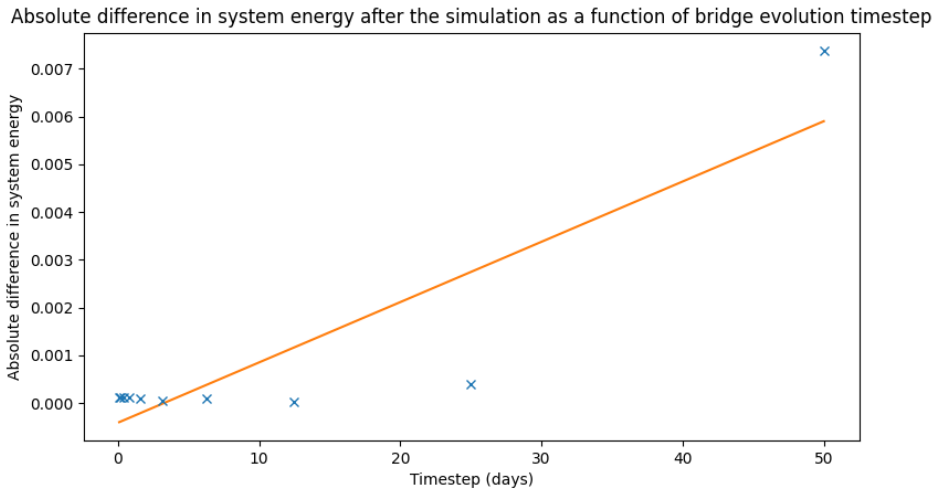

Figure 11: **Absolute difference in system energy after the simulation as a function of the bridge evolution timestep:** On the x-axis, we see the time-step in simulated days, on the y-axis we see the error of the system. The orange line is a linear function that is determined via function fitting using non-linear least squares from the SciPy optimize package [28].

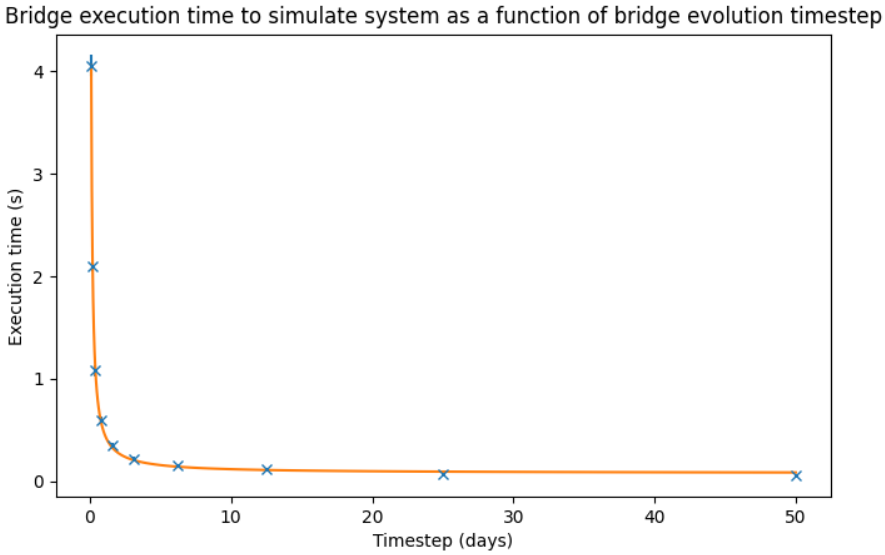

Figure 12: **Bridge execution time to simulate the system as a function of the bridge evolution time-step:** The x-axis shows the time-step in days, the y-axis shows the execution time in seconds. The orange curve is a multiplicative inverse function that is determined via function fitting using non-linear least squares from the SciPy optimize package [28].

	With literal code		Without literal code		Configuration
	AMUSE	AMOOSE	AMUSE	AMOOSE	AMOOSE
Strings	18	10	12	4	0
Words	1092	694	796	332	41
Numbers	53	39	39	25	5
.	145	46	111	9	2
**	6	6	3	3	0
+=	5	2	5	2	0
−=	2	0	2	0	0
,	40	20	18	0	0
*	18	16	14	12	1
(	85	47	50	12	0
)	85	47	50	12	0
|	11	1	10	0	2
=	65	20	53	8	0
[	15	7	8	0	0
]	15	7	8	0	0
{	7	0	7	0	0
}	7	0	7	0	0
/	13	13	9	9	0
:	11	4	7	0	1
`−	0	9	0	0	0
−`	0	9	0	0	0
−	6	4	5	3	0
+	3	3	0	0	0
<	1	0	1	0	0
;	0	1	0	1	0
Space	293	166	232	101	16
Tab	18	68	5	51	10
Newline	107	92	79	43	11
Total non-whitespace	1703	1005	1215	432	52
Total including whitespace	2121	1331	1531	627	89

Table 1: An overview of all token (left column) counts for an equivalent program of an AMUSE tutorial script [20]. The "With literal code" columns include the token counts originating from the code included in the literal code sections. We have not included any optimizations of these statements in the current version of AMOOSE, so the effective counts are shown in the "Without literal code" columns scripts. We only counted the tokens when removing these literal code sections, both from the AMOOSE script and the corresponding lines in the AMUSE scripts. "Configuration" shows the token counts of all tokens that were used in the configuration file that accompanies the AMOOSE script. The bottom two rows show the total token counts when excluding and including whitespace respectively.

reduction. The full overview of the token counts can be found in Table 1. This reduction does not necessarily mean that the language is better understandable in the general case. However, we argue that AMOOSE is easier to understand, as we did not add required specifications, but we rather only removed tokens that were perceived as redundant, then restructured the essential information. In other words, we have raised the density of information while coming up with a grammatically feasible language. For this single AMOOSE program, we could successfully generate an arbitrary amount of variants of the target code, each of which is a valid script using a different community code. By calibration, we can predict an optimal time-step parameter, where each final code yields a different performance and accuracy. This feature simplifies the use of AMUSE for novice users, as non-trivial parameter values can be automatically determined. For experienced users, this shortens the time needed to determine this parameter value using a similar analysis. The automatic determination of non-trivial parameter values does not follow from the token counts, as this is only an indicator showing improvements in the syntax of the language.

7 Discussion / Future Work

We believe AMOOSE serves as an enabler for more productive use and the introduction of new features in the AMUSE framework, as we have partially described. One of those features is the automatic parameter calibration. Besides this, there are several other avenues for future work to aid in performance tuning and optimization, and in automating the generation of trivial statements. The performance and accuracy depend on the chosen community codes and their configuration. Different community codes may exist for a category, such as for gravitational dynamics. Some codes are more suitable than others for certain modelled systems and platform specifications. In the future, we will be able to perform a thorough but quick search for the best community code and configuration of parameter values for a particular astrophysical problem and target architecture. This will be achieved by quantifying the modelled system and performing a dimensionality reduction on the independent variables. As multiple users will be exploring the parameter space in a distributed manner, we foresee the creation of a central server where calibration results can be shared to help other users find suitable configurations in less time.

As the structure of interactions between particles may change during execution, such as a moon leaving the orbit of a planet due to an external force, configurations need to react to these changes at runtime to remain accurate. While difficult to accomplish in Python code, AMOOSE can transparently support continuous code generation by periodically re-compiling and re-calibrating the original AMOOSE program.

AMUSE comes with a couple of pitfalls and inconsistencies, which experienced users know how to deal with. We can incorporate their experience in a knowledge base in the form of rules to generate better code. This also helps dealing with the heterogeneous nature of the different community codes, and adds more predefinitions and shorthands. Knowledge can also be automatically generated, as we can test whether generated snippets of code result in an error when executed. This reveals information about, for example, unit conversions.

Finally, AMOOSE may also be generalized to support simulation of systems in different research fields. The nature of simulation in general allows for this opportunity, as simulations mainly follow two steps: the definition of a static model followed by a description of the dynamics.

In our implementation, we have made a couple of simplifications in our interest. For example, the most obvious improvement we can make is to eliminate literal code sections, supporting all statements that may occur in a Python script as AMOOSE statements. Furthermore, we could implement a more generic manner of configuring the front-end, possibly using more configuration files, to configure the compiler. In this manner it would even be possible to create a pure compiler generator, as opposed to regular parser generators.

8 Conclusion

In this thesis, we have introduced AMOOSE, a DSL for the AMUSE framework and the accompanied compiler ANTLER. AMUSE is a framework that helps solving computational astrophysics problems, defining simulations is performed using the features AMUSE offers. Units, quantities and constants are provided as they are fundamental to physics. Users can define particle sets of objects that are contained in the system they are defining. Community codes are used for defining dynamics on these static objects, which simulate a time-step based evolution, where bridges allow for the simultaneous use of multiple community codes, with different time-step values.
As AMUSE has its caveats for both novice and experienced users, we tried solving this problem by introducing a new DSL. We first identified the issues by interviewing astrophysics students, collecting information on the framework ourselves and we defined some additional requirements. The new DSL, AMOOSE, needs to be more efficient to work with, so we increase the information density. We structure the way systems are defined in a hierarchical manner, using indentation, and make sure the grammar is feasible. The source code can be split up into machine independent AMOOSE code and machine dependent configuration files. We gave an overview of all statements that are supported by AMOOSE. Furthermore, a calibration should be performed for values that are non-trivial for users to pick. This turns the prior imperative language into a new declarative domain specific language.
To implement AMOOSE, we develop a compiler, ANTLER. We explain how we used the traditional components found in compilers, but also the differences. As we compile to another source code instead of machine code, our intermediate language and back-end were developed likewise. In particular, the intermediate language is designed to allow for the generation of well-readable target code. Our back-end handles operations separately from abstract models, which are used for the definition of more complex structures, such as particle sets and bridges, or the application of a bridge. To recall attributes defined in prior models, we implemented a global context, which stores all information about handled models, prior to a certain current model. Furthermore, the calibration of the time-step variable of the bridge is performed by branching off into multiple samples, processed by an analysis module. To make modifications to ANTLER easy, we expose entry points to the code generation, configuration of available community codes and calibrator analysis module.
AMOOSE uses fewer tokens to create a model configuration that is equivalent to a target code in the AMUSE framework. The essence of introducing AMOOSE to students in the astrophysical research field is that they will be able to focus better on the subject of modeling. We show an evaluation of our calibration method on an instance of an AMOOSE program. AMOOSE made calibration possible due to being compiled by our transpiler ANTLER, an enabler that can perform interesting analyses to automatically determine appropriate configurations. We performed an experiment where we could find a natural value for the non-trivial time-step parameter from an analysis of different sampled intermediate target codes. Our initial results show that the AMOOSE approach works, leaving many more possibilities to follow in the future. The descriptive nature of the language eliminates redundancy that imperative languages cause. Our results show a

65% reduction in the number of tokens for defining an equivalent program in AMOOSE, for a representative AMUSE tutorial script. Additionally, AMOOSE creates many opportunities to further improve the usability of AMUSE and to introduce new features, such as collaborative online parameter space exploration, continuous code generation and the addition of a more elaborate knowledge base.

References

[1] Python - full grammar specification. https://docs.python.org/3/reference/grammar.html. Accessed: 2022-05-21.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[3] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. 2019. doi: 10.48550/ARXIV.1903.01855. URL https://arxiv.org/abs/1903.01855.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. ISBN 0321486811.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi: 10.1145/2228360.2228584.

[6] Robert A. Van Engelen. ATMOL: A domain-specific language for atmospheric modeling. *Journal of Computing and Information Technology*, 9(4):289, 2001. doi: 10.2498/cit.2001.04.02. URL https://doi.org/10.2498/cit.2001.04.02.

[7] Roland Ewald and Adelinde M. Uhrmacher. Sessl: A domain-specific language for simulation experiments. *ACM Trans. Model. Comput. Simul.*, 24(2), feb 2014. ISSN 1049-3301. doi: 10.1145/2567895. URL https://doi.org/10.1145/2567895.

[8] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of understandability. volume 29, pages 353–366, 01 2009. ISBN 978-3-642-01861-9. doi: 10.1007/978-3-642-01862-6_29.

[9] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN 9780131392809. URL https://books.google.nl/books?id=ri1muolw_YwC.

[10] Michiko Fujii, Masaki Iwasawa, Yoko Funato, and Junichiro Makino. BRIDGE: A Direct-Tree Hybrid N-Body Algorithm for Fully Self-Consistent Simulations of Star Clusters and Their Parent Galaxies. , 59:1095, December 2007. doi: 10.1093/pasj/59.6.1095.

[11] D. Grune, H.E. Bal, H.E. Bal, C.J.H. Jacobs, and K.G. Langendoen. *Modern Compiler Design*. Worldwide Series in Computer Science. Wiley, 2000. ISBN 9780471976974.

[12] M. H. Hénon. The Monte Carlo Method (Papers appear in the Proceedings of IAU Colloquium No. 10 Gravitational N-Body Problem (ed. by Myron Lecar), R. Reidel Publ. Co. , Dordrecht-Holland.). , 14(1):151–167, November 1971. doi: 10.1007/BF00649201.

[13] John Levine and Levine John. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596155972.

[14] Niels Lohmann. JSON for Modern C++. https://json.nlohmann.me. Accessed: 2022-05-22.

[15] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892. URL https://doi.org/10.1145/1118890.1118892.

[16] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, March 1956. URL http://www.musanim.com/miller1956/.

[17] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. Auto-scaling cloud-based memory-intensive applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 229–237, 2020. doi: 10.1109/CLOUD49709.2020.00042.

[18] Bjørn Myhrhaug Ole-Johan Dahl and Kirsten Nygaard. Simula common base language. 1970.

[19] Simon Portegies Zwart. Amuse-tutorial, spzwart. https://github.com/spzwart/AMUSE-Tutorial, . Accessed: 2022-05-22.

[20] Simon Portegies Zwart. Amuse tutorial on high-order bridge. https://github.com/spzwart/AMUSE-Tutorial/blob/master/bridge_gravity_with_hydro.ipynb, . Accessed: 2022-05-22.

[21] Simon Portegies Zwart. Amuse tutorial on high-order bridge. https://github.com/spzwart/AMUSE-Tutorial/blob/master/high_order_bridge.ipynb, . Accessed: 2022-05-22.

[22] Simon Portegies Zwart. Amuse tutorial on particle sets. https://github.com/spzwart/AMUSE-Tutorial/blob/master/particles.ipynb, . Accessed: 2022-05-22.

[23] Simon Portegies Zwart and Steve McMillan. *Astrophysical Recipes*. 2514-3433. IOP Publishing, 2018. ISBN 978-0-7503-1320-9. doi: 10.1088/978-0-7503-1320-9. URL https://dx.doi.org/10.1088/978-0-7503-1320-9.

[24] Simon Portegies Zwart, Inti Pelupessy, Carmen Martínez-Barbosa, Arjen van Elteren, and Steve McMillan. Non-intrusive hierarchical coupling strategies for multi-scale simulations in gravitational dynamics. *Communications in Nonlinear Science and Numerical Simulations*, 85:105240, June 2020. doi: 10.1016/j.cnsns.2020.105240.

[25] Reinhardt M. Rosenberg. *Analytical Dynamics od Discrete Systems*. Mathematical Concepts and Methods in Science and Engineering. Plenum Press, 1977. ISBN 030631014-7.

[26] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988. doi: https://doi.org/10.1207/s15516709cog1202_4. URL https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1202_4.

[27] Robert van Engelen, Lex Wolters, and Gerard Cats. CTADEL. In *Proceedings of the 10th international conference on Supercomputing - ICS '96*. ACM Press, 1996. doi: 10.1145/237578.237589. URL https://doi.org/10.1145/237578.237589.

[28] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[29] J. von Neumann. First draft of a report on the EDVAC. 1945.

[30] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2 (OOPSLA), oct 2018. doi: 10.1145/3276491. URL https://doi.org/10.1145/3276491.

A AMUSE Units

Base quantity name	**Base unit name**	**Symbol**
Length	meter	m
Mass	kilogram	kg
Time	second	s
Electric current	ampere	A
Thermodynamic temperature	kelvin	K
Amount of substance	mole	mol
Luminous intensity	candela	cd

Table 2: SI 7 Basic Units of Measurement.

Unit name	**Symbol**	**Value**
Hertz	Hz	s^{-1}
Megahertz	MHz	$1 \times 10^6\,\mathrm{Hz}$
Radian	rad	$\mathrm{m\,m}^{-1}$
Steradian	sr	$\mathrm{m}^2\,\mathrm{m}^{-2}$
Newton	N	$\mathrm{kg\,m\,s}^{-2}$
Pascal	Pa	$\mathrm{N\,m}^{-2}$
Joule	J	$\mathrm{kg\,m}^2\,\mathrm{s}^{-2}$
Watt	W	$\mathrm{J\,s}^{-1}$
Farad	F	$\mathrm{s}^4\,\mathrm{A}^2\,\mathrm{m}^{-2}\,\mathrm{kg}^{-1}$
Coulomb	C	A s
Volt	V	$\mathrm{J\,C}^{-1}$
Tesla	T	$\mathrm{kg\,A}^{-1}\,\mathrm{s}^{-2}$
Ohm	ohm	$\mathrm{V\,A}^{-1}$
Siemens	S	$\mathrm{A\,V}^{-1}$
Weber	Wb	V s

Table 3: AMUSE Derived SI Units.

Unit name	**Symbol**	**Value**
Angstrom	angstrom	$1 \times 10^{-10}\,\mathrm{m}$
Astronomical unit	AU	$149\,597\,870\,691.0\,\mathrm{m}$
Astronomical unit	au	$149\,597\,870\,691.0\,\mathrm{m}$
AU per day	AUd	$149\,597\,870\,691.0\,\mathrm{m\,d^{-1}}$
Parsec	parsec	$\mathrm{AU}^{-1}\tan(\pi/(180\cdot 60\cdot 60))$
Kilo parsec	kpc	1×10^{3} parsec
Mega parsec	mpc	1×10^{6} parsec
Giga parsec	gpc	1×10^{9} parsec
Light year	ly	$9\,460\,730\,472\,580.8\,\mathrm{km}$
Solar luminosity	LSun	$3.839 \times 10^{26}\,\mathrm{W}$
Solar mass	MSun	$1.988\,92 \times 10^{30}\,\mathrm{kg}$
Jupiter mass	MJupiter	$1.8987 \times 10^{27}\,\mathrm{kg}$
Earth mass	MEarth	$5.9722 \times 10^{24}\,\mathrm{kg}$
Kilo year	kyr	$1 \times 10^{3}\,\mathrm{yr}$
Million year	Myr	$1 \times 10^{6}\,\mathrm{yr}$
Giga (billion) year	Gyr	$1 \times 10^{9}\,\mathrm{yr}$

Table 4: AMUSE Astronomical Units.

Unit name	**Symbol**	**Value**
Electron charge	e	$1.602\,176\,487 \times 10^{-19}\,\mathrm{C}$
Electron volt	eV	$e\,\mathrm{V}$
Mega electron volt	MeV	$1 \times 10^{6}\,e\,\mathrm{V}$
Giga electron volt	GeV	$1 \times 10^{9}\,e\,\mathrm{V}$
Hartree energy	E_h	$4.359\,743\,94 \times 10^{-18}\,\mathrm{J}$
Atomic mass unit	amu	$1.660\,538\,782 \times 10^{-27}\,\mathrm{kg}$
Rydberg unit	Ry	$10\,973\,731.5685\,\mathrm{m^{-1}}$

Table 5: AMUSE Units Based on Measured Quantities.

B Syntax Tables

Below, we provide the overviews of the regular expressions used in the lexical analyzers for the AMOOSE parser, configuration file parser and template parser respectively. On the left we show the exact regular expression, sometimes starting with angle brackets to indicate the current state of the lexical analyzer. Whenever the analyzer is in a certain state, it may only match regular expressions that correspond to that state. This comes in to play when having a grammar that depends on indentation or parses differently for different parts of an input file. On the right side we see either shorthands, tokens or skip. A shorthand indicates a regular expression shorthand, which may be used in other regular expressions. A token specifies the token that is given to the parser, we will see these back in the grammars in Appendix C. Skip indicates taking no action, this is often the case when changing states or reading garbage characters such as whitespace and comments. Sometimes multiple actions are shown, in which case the actual logic is given in the source code files and is too complex to summarize in this appendix.

B.1 AMOOSE code files

`[a-zA-Z_]`	shorthand `LETTER`
`[0-9]`	shorthand `DIGIT`
`{DIGIT}{DIGIT}*`	shorthand `DIGITS`
`("."({DIGITS})?)?`	shorthand `DECIMAL`
`((("E"|"e")("+"|"-")?{DIGITS}))?`	shorthand `EXPONENT`
`{DIGITS}{DECIMAL}{EXPONENT}`	shorthand `NUM`
`{LETTER}({LETTER}|{DIGIT})*`	shorthand `ID`
`\"(\\.|[^\\\"])*\"`	shorthand `STRING_LITERAL`
`<indentation><<EOF>>`	token `DEDENT`, token `ENDL`, skip
`<indentation>^[ \t]*#[^\n]*$`	skip
`<indentation>\t`	skip
`<indentation>'-[\t ]*$`	skip
`<indentation>\n`	token `ENDL`, token `DEDENT`, skip
`<indentation>.`	token `DEDENT`, token `ENDL`, skip
`<parse><<EOF>>`	skip
`<parse>\n`	skip
`<parse>{STRING_LITERAL}`	token `STRING_LITERAL`
`<parse>{NUM}`	token `NUM`
`<parse>particles?`	token `PARTICLES`
`<parse>[Bb]ridge`	token `BRIDGE`
`<parse>code`	token `CODE`
`<parse>codes`	token `CODES`
`<parse>read`	token `READ`

<parse>from	token FROM
<parse>are	token ARE
<parse>is	token IS
<parse>as	token AS
<parse>on	token ON
<parse>error	token ERROR
<parse>copy	token COPY
<parse>evolve	token EVOLVE
<parse>with	token WITH
<parse>center	token CENTER
<parse>around	token AROUND
<parse>category	token CATEGORY
<parse>apply	token APPLY
<parse>for	token FOR
<parse>properties	token PROPERTIES
<parse>to	token TO
<parse>write	token WRITE
<parse>{ID}	token ID
<parse>\(	token ROUND_BRACKET_OPEN
<parse>\)	token ROUND_BRACKET_CLOSE
<parse>,	token COMMA
<parse>\.	token DOT
<parse>;	token SEMICOLON
<parse>**	token DOUBLE_ASTERISK
<parse>*	token ASTERISK
<parse>\+	token PLUS
<parse>-	token MINUS
<parse>~	token TILDE
<parse>\/\/	token DOUBLE_FORWARD_SLASH
<parse>\/	token FORWARD_SLASH
<parse>\%	token PERCENT
<parse>@	token AT_SIGN
<parse>\= +	token PLUS_ASSIGNMENT
<parse>-=	token MINUS_ASSIGNMENT
<parse>**=	token DOUBLE_ASTERISK_ASSIGNMENT
<parse>*=	token ASTERISK_ASSIGNMENT
<parse>@=	token AT_SIGN_ASSIGNMENT

<parse>\/\/=	token DOUBLE_FORWARD_SLASH_ASSIGNMENT
<parse>\/=	token FORWARD_SLASH_ASSIGNMENT
<parse>\%=	token PERCENT_ASSIGNMENT
<parse>&=	token AMPERSAND_ASSIGNMENT
<parse>|=	token PIPE_ASSIGNMENT
<parse>\^=	token CARET_ASSIGNMENT
<parse>\<\<=	token DOUBLE_ANGLE_BRACKET_OPEN_ASSIGNMENT
<parse>\>\>=	token DOUBLE_ANGLE_BRACKET_CLOSE_ASSIGNMENT
<parse>=	token ASSIGNMENT
<parse>[\t]+	skip
<parse>#[^\n]*	skip
<literal_code_section>^[\t]*-`[\t]*$	token LITERAL_CODE
<literal_code_section>^(from [\t]+[A-z0-9_.]+[\t]+)? import[\t]+\(?[\t]* (*|(([A-z0-9_.]+([\t]+ as[\t]+[A-z0-9_]+)? ([\t]*,[\t]*[A-z0-9_.]+([\t]+ as[\t]+[A-z0-9_]+)?)*))[\t]*\)?[\t]*$	token LITERAL_CODE_IMPORT
<literal_code_section>\n	skip
<literal_code_section>.*	skip
<literal_code_section_endl>.	token ENDL
<literal_code_section_endl>\n	token ENDL

B.2 Configuration files

[a-zA-Z_]	shorthand CLETTER
[0-9]	shorthand CDIGIT
{CLETTER}({CLETTER}|{CDIGIT})*	shorthand CID
^[\t]*{CID}[\t]*:[\t]*$	token CID
^[\t]*{CID}[\t]+[^ \t:\n][^\n]*[\t]*$	token CPARAM
[\t]*\n	skip
^[\t]+	skip

B.3 Templates

<tmplt_text>\n	token TMPLT_LINE_END
<tmplt_text>\$\{	skip
<tmplt_text>([^\$\{\n]*\$[^\{\n]|[^\$\n]\{|^\{|\$$)*	token TMPLT_TEXT
<tmplt_frmt>\}	token TMPLT_EXPR_CLOSE, skip
<tmplt_frmt>for	token TMPLT_FOR
<tmplt_frmt>endfor	token TMPLT_ENDFOR
<tmplt_frmt>:	token TMPLT_COLON
<tmplt_frmt>[0-9]+	token TMPLT_NUM
<tmplt_frmt>#?[A-Za-z_][A-Za-z0-9_]*	token TMPLT_ID
<tmplt_frmt>\.	token TMPLT_DOT
<tmplt_frmt>\[	token TMPLT_SQ_OPEN
<tmplt_frmt>\]	token TMPLT_SQ_CLOSE
<tmplt_frmt>\(	token TMPLT_RB_OPEN
<tmplt_frmt>\)	token TMPLT_RB_CLOSE
<tmplt_frmt>,	token TMPLT_COMMA
<tmplt_frmt>\$\{	token TMPLT_EXPR_OPEN

C Grammars

Here, we show the grammars for parsing AMOOSE code files, configuration files and templates respectively. Empty strings are indicated by λ, non-terminals are encapsulated in angle brackets, constants are not. Use the overview from Appendix B for the corresponding parser type to find the regular expressions that generate these constants.

C.1 AMOOSE code files

⟨*program*⟩ ::= ⟨*statement_list*⟩
  | λ

⟨*statement_list*⟩ ::= ⟨*statement*⟩ ENDL
  | ⟨*statement_list*⟩ ⟨*statement*⟩ ENDL

⟨*statement*⟩ ::= ⟨*read_file*⟩
  | ⟨*write_file*⟩
  | ⟨*particles_creation*⟩
  | ⟨*bridge_creation*⟩
  | ⟨*literal_code_block*⟩
  | ⟨*literal_code_import*⟩
  | ⟨*assignment*⟩
  | ⟨*action*⟩
  | ⟨*bridge_application*⟩
  | ENDL

⟨*bridge_application*⟩ ::= APPLY ⟨*name*⟩ FOR ⟨*quantity*⟩ WITH ERROR ⟨*name*⟩
    ENDL INDENT ⟨*optional_literal_code_block_endl*⟩ EVOLVE ENDL
    ⟨*optional_literal_code_block_endl*⟩ COPY
    ⟨*optional_literal_code_block_dedent*⟩

⟨*optional_literal_code_block_dedent*⟩ ::= ENDL ⟨*literal_code_block*⟩ DEDENT
  | DEDENT

⟨*optional_literal_code_block_endl*⟩ ::= ⟨*literal_code_block*⟩ ENDL
  | λ

⟨*action*⟩ ::= CENTER ⟨*name*⟩

⟨*assignment*⟩ ::= ⟨*single_target*⟩ ⟨*assignment_op*⟩ ⟨*quantity*⟩

⟨*single_target*⟩ ::= ⟨*single_subscript_attribute_target*⟩
  | ⟨*name*⟩
  | ROUND_BRACKET_OPEN ⟨*single_target*⟩ ROUND_BRACKET_CLOSE

⟨*single_subscript_attribute_target*⟩ ::= ⟨*t_primary*⟩ DOT ⟨*name*⟩

⟨*t_primary*⟩ ::= ⟨*t_primary*⟩ DOT ⟨*name*⟩
  | ⟨*name*⟩

⟨*assignment_op*⟩ ::= PLUS_ASSIGNMENT
  | MINUS_ASSIGNMENT
  | DOUBLE_ASTERISK_ASSIGNMENT
  | ASTERISK_ASSIGNMENT
  | AT_SIGN_ASSIGNMENT
  | DOUBLE_FORWARD_SLASH_ASSIGNMENT
  | FORWARD_SLASH_ASSIGNMENT
  | PERCENT_ASSIGNMENT
  | AMPERSAND_ASSIGNMENT
  | PIPE_ASSIGNMENT
  | CARET_ASSIGNMENT
  | DOUBLE_ANGLE_BRACKET_OPEN_ASSIGNMENT
  | DOUBLE_ANGLE_BRACKET_CLOSE_ASSIGNMENT
  | ASSIGNMENT

⟨*literal_code_block*⟩ ::= LITERAL_CODE

⟨*literal_code_import*⟩ ::= LITERAL_CODE_IMPORT

⟨*read_file*⟩ ::= READ ⟨*name*⟩ FROM ⟨*string*⟩

⟨*write_file*⟩ ::= WRITE ⟨*name*⟩ TO ⟨*string*⟩

⟨*particles_creation*⟩ ::= ⟨*name*⟩ ⟨*is_or_are*⟩ ⟨*name*⟩ ⟨*optional_particle_set_size*⟩
    ⟨*optional_particles_creation_specifiers*⟩

⟨*optional_particles_creation_specifiers*⟩ ::= WITH
    ⟨*semicolon_seperated_particles_creation_specifiers*⟩
  | WITH ENDL INDENT ⟨*multiline_particles_creation_specifiers*⟩ DEDENT
  | λ

⟨*semicolon_seperated_particles_creation_specifiers*⟩ ::= ⟨*particles_creation_specifier*⟩
  | ⟨*semicolon_seperated_particles_creation_specifiers*⟩ SEMICOLON
    ⟨*particles_creation_specifier*⟩

⟨*multiline_particles_creation_specifiers*⟩ ::= ⟨*particles_creation_specifier*⟩
  | ⟨*multiline_particles_creation_specifiers*⟩ ENDL ⟨*particles_creation_specifier*⟩

⟨*optional_particle_set_size*⟩ ::= ROUND_BRACKET_OPEN ⟨*expression*⟩
    ROUND_BRACKET_CLOSE
  | λ

⟨*particles_creation_specifier*⟩ ::= ⟨*particle_creation*⟩
  | ⟨*initializer_list*⟩
  | CENTER
  | AROUND ⟨*name*⟩

⟨*particle_creation*⟩ ::= PARTICLES ⟨*comma_seperated_particle_creation*⟩
  | PARTICLES ENDL INDENT ⟨*multi_line_particle_creation*⟩ DEDENT

⟨*comma_seperated_particle_creation*⟩ ::= ⟨*name*⟩
  | ⟨*comma_seperated_particle_creation*⟩ COMMA ⟨*name*⟩

⟨*multi_line_particle_creation*⟩ ::= ⟨*name*⟩ ⟨*optional_initializer_list*⟩
  | ⟨*multi_line_particle_creation*⟩ ENDL ⟨*name*⟩ ⟨*optional_initializer_list*⟩

⟨*optional_initializer_list*⟩ ::= WITH ⟨*initializer_list*⟩
  | λ

⟨*initializer_list*⟩ ::= PROPERTIES ⟨*comma_seperated_initializer_list*⟩
  | PROPERTIES ENDL INDENT ⟨*multi_line_initializer_list*⟩ DEDENT

⟨*comma_seperated_initializer_list*⟩ ::= ⟨*initializer*⟩
  | ⟨*comma_seperated_initializer_list*⟩ COMMA ⟨*initializer*⟩

⟨*multi_line_initializer_list*⟩ ::= ⟨*initializer*⟩
  | ⟨*multi_line_initializer_list*⟩ ENDL ⟨*initializer*⟩

⟨*initializer*⟩ ::= ⟨*name*⟩ ⟨*quantity*⟩
  | ⟨*name*⟩ ⟨*quantity*⟩ AS ⟨*name*⟩

⟨*bridge_creation*⟩ ::= ⟨*name*⟩ ⟨*is_or_are*⟩ BRIDGE ⟨*optional_bridge_creation_specifiers*⟩

⟨*optional_bridge_creation_specifiers*⟩ ::= WITH ENDL INDENT
    ⟨*bridge_creation_specifiers*⟩ DEDENT
  | λ

⟨*bridge_creation_specifiers*⟩ ::= ⟨*bridge_creation_specifier*⟩ ENDL
    ⟨*bridge_creation_specifiers*⟩
  | ⟨*bridge_creation_specifier*⟩

⟨*bridge_creation_specifier*⟩ ::= ⟨*initializer_list*⟩
  | ⟨*code_application_list*⟩

⟨*code_application_list*⟩ ::= CODES ENDL INDENT
    ⟨*multiline_seperated_code_application_list*⟩ DEDENT

⟨*multiline_seperated_code_application_list*⟩ ::= ⟨*multiline_code_application*⟩
  | ⟨*multiline_code_application*⟩ ENDL ⟨*multiline_seperated_code_application_list*⟩

⟨*multiline_code_application*⟩ ::= ⟨*optionally_renamed_code*⟩ ON ⟨*particle_group_list*⟩

⟨*particle_group_list*⟩ ::= ENDL INDENT ⟨*multiline_particle_group_list*⟩ DEDENT
  | ⟨*semicolon_seperated_particle_group_list*⟩

⟨*particle_group_specifier*⟩ ::= ⟨*name*⟩
  | PARTICLES

⟨*multiline_particle_group_list*⟩ ::= ⟨*particle_group_specifier*⟩ ⟨*particle_expression_list*⟩
    ENDL ⟨*multiline_particle_group_list*⟩
  | ⟨*particle_group_specifier*⟩ ⟨*particle_expression_list*⟩

⟨*particle_expression_list*⟩ ::= ENDL INDENT ⟨*multiline_particle_expression_list*⟩ DEDENT
  | ⟨*comma_seperated_particle_expression_list*⟩

⟨*multiline_particle_expression_list*⟩ ::= ⟨*particle_expression*⟩ ENDL
    ⟨*multiline_particle_expression_list*⟩
  | ⟨*particle_expression*⟩

⟨*semicolon_seperated_particle_group_list*⟩ ::= ⟨*particle_group_specifier*⟩
    ⟨*comma_seperated_particle_expression_list*⟩ SEMICOLON
    ⟨*semicolon_seperated_particle_group_list*⟩
  | ⟨*particle_group_specifier*⟩ ⟨*comma_seperated_particle_expression_list*⟩

⟨*comma_seperated_particle_expression_list*⟩ ::= ⟨*particle_expression*⟩
  | ⟨*comma_seperated_particle_expression_list*⟩ COMMA ⟨*particle_expression*⟩

⟨*particle_expression*⟩ ::= ⟨*name*⟩
  | ⟨*particle_expression*⟩ MINUS ⟨*name*⟩
  | ⟨*particle_expression*⟩ PLUS ⟨*name*⟩

⟨*optionally_renamed_code*⟩ ::= ⟨*code*⟩
  | ⟨*code*⟩ AS ⟨*name*⟩

⟨*code*⟩ ::= CODE ⟨*name*⟩
  | CATEGORY ⟨*code_categorisation*⟩

⟨*code_categorisation*⟩ ::= ⟨*name*⟩
  | ⟨*code_categorisation*⟩ DOT ⟨*name*⟩

⟨*group*⟩ ::= ROUND_BRACKET_OPEN ⟨*expression*⟩ ROUND_BRACKET_CLOSE

⟨*primary*⟩ ::= ⟨*primary*⟩ DOT ⟨*name*⟩
  | ⟨*atom*⟩

⟨*expression*⟩ ::= ⟨*sum*⟩

⟨*power*⟩ ::= ⟨*primary*⟩ DOUBLE_ASTERISK ⟨*factor*⟩
  | ⟨*primary*⟩

⟨*factor*⟩ ::= PLUS ⟨*factor*⟩
  | MINUS ⟨*factor*⟩
  | TILDE ⟨*factor*⟩
  | ⟨*power*⟩

⟨*term*⟩ ::= ⟨*term*⟩ ASTERISK ⟨*factor*⟩
  | ⟨*term*⟩ FORWARD_SLASH ⟨*factor*⟩
  | ⟨*term*⟩ DOUBLE_FORWARD_SLASH ⟨*factor*⟩
  | ⟨*term*⟩ PERCENT ⟨*factor*⟩
  | ⟨*term*⟩ AT_SIGN ⟨*factor*⟩
  | ⟨*factor*⟩

⟨*sum*⟩ ::= ⟨*sum*⟩ PLUS ⟨*term*⟩
  | ⟨*sum*⟩ MINUS ⟨*term*⟩
  | ⟨*term*⟩

⟨*atom*⟩ ::= ⟨*name*⟩
  | TRUE
  | FALSE
  | ⟨*string*⟩
  | ⟨*number*⟩
  | ⟨*tuple*⟩
  | ⟨*group*⟩

⟨*quantity*⟩ ::= ⟨*expression*⟩ ⟨*unit*⟩
  | ⟨*expression*⟩

⟨*unit*⟩ ::= ⟨*name*⟩

⟨*tuple*⟩ ::= ROUND_BRACKET_OPEN ROUND_BRACKET_CLOSE
  | ROUND_BRACKET_OPEN ⟨*number*⟩ COMMA ⟨*optional_numbers*⟩
    ROUND_BRACKET_CLOSE

⟨*optional_numbers*⟩ ::= ⟨*numbers*⟩
  | λ

⟨*numbers*⟩ ::= ⟨*number*⟩ COMMA ⟨*numbers*⟩
  | ⟨*number*⟩

⟨*number*⟩ ::= NUM

⟨*string*⟩ ::= STRING_LITERAL

⟨*name*⟩ ::= ID

⟨*is_or_are*⟩ ::= IS
  | ARE

C.2 Configuration files

⟨*cprogram*⟩ ::= ⟨*cconfig_list*⟩
  | λ

⟨*cconfig_list*⟩ ::= ⟨*cconfig*⟩
  | ⟨*cconfig*⟩ ⟨*cconfig_list*⟩

⟨*cconfig*⟩ ::= ⟨*cidentifier*⟩
  | ⟨*cidentifier*⟩ ⟨*cparameter_list*⟩

⟨*cparameter_list*⟩ ::= ⟨*cparameter*⟩
  | ⟨*cparameter*⟩ ⟨*cparameter_list*⟩

⟨*cidentifier*⟩ ::= CID

⟨*cparameter*⟩ ::= CPARAM

C.3 Templates

⟨*tprogram*⟩ ::= ⟨*tmplt_statements*⟩ TMPLT_LINE_END
  | λ

⟨*tmplt_statements*⟩ ::= ⟨*tmplt_statements*⟩ TMPLT_LINE_END ⟨*tmplt_statement_base*⟩
  | ⟨*tmplt_statement_base*⟩

⟨*tmplt_statement_base*⟩ ::= ⟨*tmplt_statement*⟩
  | ⟨*tmplt_for_statement*⟩

⟨*tmplt_statement*⟩ ::= ⟨*tmplt_statement*⟩ ⟨*tmplt_statement_part*⟩
  | ⟨*tmplt_statement_part*⟩

⟨*tmplt_statement_part*⟩ ::= TMPLT_TEXT
  | ⟨*tmplt_expression*⟩

⟨*tmplt_expression*⟩ ::= ⟨*tmplt_atom*⟩
  | ⟨*tmplt_macro_expression*⟩

⟨*tmplt_atom*⟩ ::= ⟨*tmplt_primary*⟩
  | ⟨*tmplt_atom*⟩ TMPLT_DOT ⟨*tmplt_name*⟩
  | ⟨*tmplt_atom*⟩ TMPLT_SQ_OPEN ⟨*tmplt_primary*⟩ TMPLT_SQ_CLOSE

⟨*tmplt_primary*⟩ ::= TMPLT_NUM
  | ⟨*tmplt_name*⟩

⟨*tmplt_name*⟩ ::= TMPLT_ID

⟨*tmplt_macro_expression*⟩ ::= ⟨*tmplt_name*⟩ TMPLT_RB_OPEN
    ⟨*tmplt_optional_comma_sep_expressions*⟩ TMPLT_RB_CLOSE

⟨*tmplt_optional_comma_sep_expressions*⟩ ::= λ
  | ⟨*tmplt_comma_sep_expressions*⟩

⟨*tmplt_comma_sep_expressions*⟩ ::= ⟨*tmplt_expression*⟩
  | ⟨*tmplt_comma_sep_expressions*⟩ TMPLT_COMMA ⟨*tmplt_expression*⟩

⟨*tmplt_for_statement*⟩ ::= ⟨*tmplt_for_statement_open*⟩ TMPLT_LINE_END
    ⟨*tmplt_statements*⟩ TMPLT_LINE_END ⟨*tmplt_for_statement_close*⟩
  | ⟨*tmplt_for_statement_open*⟩ TMPLT_LINE_END ⟨*tmplt_for_statement_close*⟩

⟨*tmplt_for_statement_open*⟩ ::= TMPLT_FOR ⟨*tmplt_name*⟩ TMPLT_COLON
    ⟨*tmplt_atom*⟩

⟨*tmplt_for_statement_close*⟩ ::= TMPLT_ENDFOR

Glossary

abstract model Abstract models are structures that contain a reference to a model definition and contain concrete arguments to this referenced model. A coherent block of source code that contains information for a particular model is compiled into an abstract model, which is then compiled into target code after looking up the model description. 21, 41, 45, 50, 52, 53, 55, 68

actual type The actual type of a variable is the type a variable truly has, disregarding the context the variable is used in. The actual type of a variable may be converted into the expected type, if such a type conversion is defined. 16, 18

ambiguity A grammar is ambiguous when there exists more than a single derivation of some text, such that there may be different resulting parse trees. 18, 20

compiled language A programming language that is compiled into machine code before execution is possible. 14

declarative A programming language is declarative when the user specifies what computation is performed by the machine, rather than the exact manner in which it is performed. 4, 11, 14, 28, 29, 68

expected type The expected type of a variable is the type a variable is expected to be, derived from the context it is used in. A variable's actual type may be converted into the expected type, if such a type conversion is defined. 16

final target code The final target code of the compilation pipeline is the actual output of our compiler. 33, 40, 41, 45, 46, 56, 58, 59

global context Abstract models may need to recall information from prior models. In the global context, model argument values are stored for later use, such that they can be indexed regarding the abstract model they were once local to. 5, 41, 55, 56, 58–60, 68

imperative A programming language is imperative when the user specifies exact manner in which a computation it is performed, through a detailed description of the consecutive operations contained in the procedure. 4, 7, 11, 14, 21, 29, 68

intermediate target code The intermediate target code of the compilation pipeline is an intermediate output of our compiler, a medium to be used by the pipeline for experimentation and analysis. 40, 45, 58–60, 63

interpreted language A programming language that is interpreted by a certain engine to be executed, this can be done without any compilation steps, straight from the source code. 14

keyword A keyword is a predefined token reserved by the programming language and may apply additional semantics. 30, 33, 34, 38, 48

local context Abstract models contain a set of model arguments, to be used while handling the current model. These variables are local to the current model and form the local context of the abstract model. 55, 56

machine dependent Dependent of the machine the software is executed on. 41, 50, 51, 68

machine independent Independent of the machine the software is executed on. 41, 50, 51, 68

precondition The full description of both the static and dynamics of a system model. 23, 27, 34

source code The source code is the input of a compiler or interpreter, containing the developer's written program description. 3, 15, 18, 22, 25, 28, 40–42, 45, 51, 52, 55, 57, 59, 68

strong typed A programming language is strong typed when type checks are performed. 16

weakly typed A programming language is weakly typed when type checks are not performed. 16

Acronyms

ALU Arithmetic/Logic Unit. 13

AMUSE Astrophysical Multipurpose Software Environment. 3, 7, 8, 13, 23–30, 32–35, 37, 38, 40, 45–47, 50, 54, 58, 63, 66–69, 73, 74

AST Abstract Syntax Tree. 18, 19, 21, 41, 46, 50–52

ATMOL ATmospheric MOdeling Language. 11, 12

COBOL COmmon Business Oriented Language. 9

CPU Central Processing Unit. 13, 15

CTADEL Code-generation Tool for Applications based on Differential Equations using high-level Language specifications. 11, 12

CU Control Unit. 13

DSL Domain Specific Language. 3, 4, 7–14, 22, 68

GPL General Purpose (Programming) Language. 9

OOP Object Oriented Programming. 10, 11

SESSL Simulation Experiment Specification via a Scala Layer. 4, 10

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. 10