OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Building Domain Specific Languages and Polyglot Applications with GraalVM

Presented to the Faculty of Computer Science and Mathematics
University of Applied Sciences Regensburg
Study Programme:
Master Computer Science

Master Thesis

In Partial Fulfillment of the Requirements for the Degree of
Master of Science (M.Sc.)

Presented by:	Christian Paling
Student Number:	3213285
Primary Supervising Professor:	Prof. Dr. Michael Bulenda
Secondary Supervising Professor:	Prof. Dr. Carsten Kern
Submission Date:	April 14, 2021

Thesis Declaration

Abstract

Table of Contents

1 Introduction

In May 2019, Oracle published an announcement [9] which advertises the release of a new technology called *GraalVM*. In the announcement, Scott Lynn describes GraalVM as a virtual machine which delivers benefits such as higher efficiency and greater agility for companies working in the cloud environment. Claims are being made which pledge up to three times better performance of applications, hundred times faster application startup, and five times lower memory usage.

Among the highlighted core features of this new virtual machine is a language implementation framework which, according to the announcement, enables developers to implement any language for the GraalVM environment. Furthermore, GraalVM provides polyglot capabilities to allow the combination of multiple languages in a single program. The high-level architecture of these capabilities are depicted in figure 1. It illustrates that GraalVM is able to take code of a wide variety of languages as input to produce executable code. Interestingly, GraalVM does not only produce executable code for a single runtime, e.g. for the JRE, but supports multiple runtimes like Node.js or even a (Oracle) database.

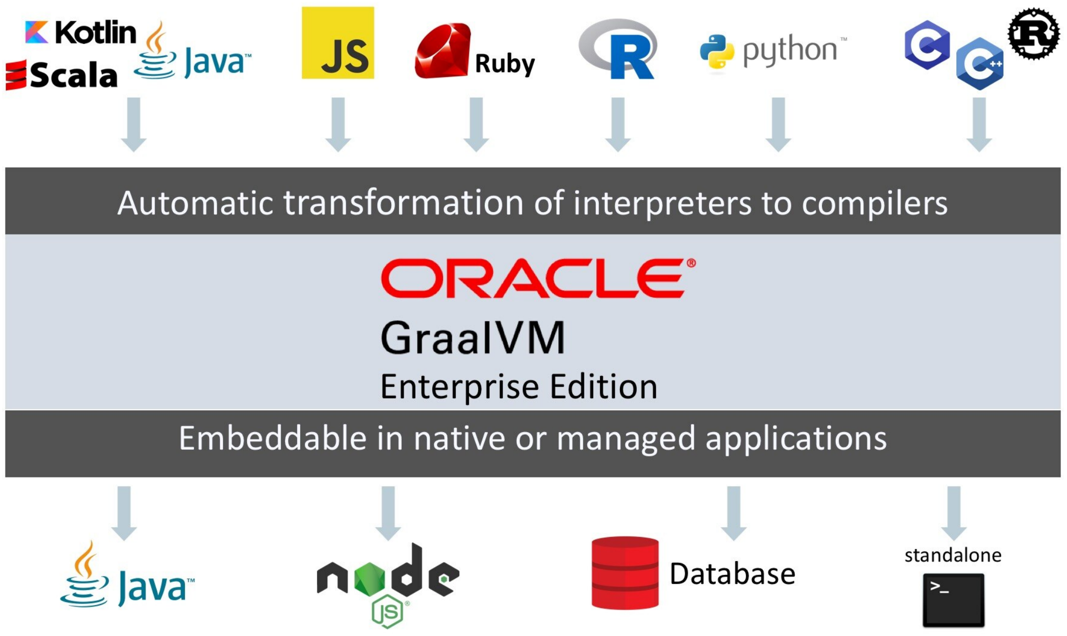

Kotlin
Scala
Java
JS
Ruby
R
python
Automatic transformation of interpreters to compilers
ORACLE
GraalVM
Enterprise Edition
Embeddable in native or managed applications
Java
node JS
Database
standalone
>_

Figure 1: High-level architecture of GraalVM, taken from [9].

These capacities of GraalVM offer various new possibilities. Languages could be reimplemented on top of GraalVM and could potentially reap performance improvements. Indeed, this has already been put into practice with *TruffleRuby*[1] which is an implementation of the Ruby language on top of GraalVM performing about seven times better than the standard Ruby implementation (CRuby) in some benchmarks [13].

Another possibility could be the reutilisation of popular libraries of a language in other languages. It is common for certain languages to excel in specific problem domains. As an example, the Python community were able to establish the Python language as a very

[1]`https://github.com/oracle/truffleruby`

popular choice regarding machine language problems by offering high quality libraries and frameworks for artifical intelligence. GraalVM could enable developers to reuse the capabilities of these libraries in a different language, say Java, through its polyglot capabilities.

However, in this thesis, a different potential opportunity of GraalVM will be examined. Domain Specific Languages (DSLs), i.e. small programming languages tailored to a specific domain, have been around for a very long time and are very popular to this day [3]. The widespread web framework *Ruby on Rails*[2] which offers a variety of DSLs, e.g. for database migrations, routing, or testing, is a good example for the modern practical application of DSLs. The newly released GraalVM could offer potential benefits for the development of DSLs. Since GraalVM offers functionalities to implement languages, i.e. languages with an arbitrary syntax, the DSLs which a developer could build using these provided frameworks would be as expressive as desired. Additionally, through the polyglot features of GraalVM, it could also be fairly easy to interact with the DSL, as well as having the DSL available by default for every language which is supported by GraalVM. Getting a good performance through the virtual machine would be an additional incentive to use GraalVM, though in the case of DSLs this only plays a minor role.

This thesis tries to explore this idea by showcasing and evaluating the implementation of a DSL for mathematical expressions called *A Mathematical Language (AML)*. The value proposition of this DSL is to enable developers to write mathematical code as similar as possible to the way it would be mathematically expressed. Especially for the implementation of algorithms, some languages are only rudimentarily suitable to express the algorithm in a clear and concise manner. These aspects of a software system could be outsourced by writing its logic in AML. In order to achieve this aim, the DSL will offer the possibility to define functions which can be written with accurate mathematical operations with *unicode* symbols. These functions can then be executed using the main programming language of the application.

The aim of this thesis is therefore threefold. First, it tries to compile and present information regarding the implementation of languages on top of GraalVM which at the time of writing is still quite sparse. Secondly, it presents the new DSL, AML, which should not only act as a means to examine GraalVM but to be actually able to be employed in practice as well. Lastly, through the implementation of the DSL, the thesis tries to evaluate the maturity and effects of building DSLs on top of GraalVM. Although this work focuses on the topic of DSLs, to a lesser extend the results of this thesis are also applicable for building programming languages and utilizing the polyglot features of GraalVM in general.

The thesis is structured as follows. First, DSLs are introduced and categorized, as well as processes and techniques for building them illustrated. Secondly, GraalVM is explained more in depth including the core features of this technology and their practical relevance. Afterwards, the implementation of AML with GraalVM is presented in detail followed by the integration of the DSL into programs using the available polyglot functionalities. Finally, the thesis concludes with a summarising chapter and highlights future potential work and prospects.

[2]`https://rubyonrails.org`

2 Domain Specific Languages

Before diving into the technical details and the implementation of DSLs on top of GraalVM, some background information is necessary to lay a foundation for the upcoming chapters of this thesis. First, the term *domain specific language* is properly defined and a distinction between different types of DSLs is made. Afterwards, benefits as well as problems of DSLs and the usage thereof are discussed. Lastly, development processes and implementation techniques to build the different types of DSLs are introduced.

2.1 Definition of Domain Specific Languages

DSLs are not by any means a modern concept. Different names have been used for DSLs such as *special purpose*, *application-oriented*, *specialized*, or *task-specific* languages [10] and examples for DSLs range back to at least 1957 where a DSL called *APT* to program numerically-controlled machines was developed [10]. Eventhough there is no standardized definition of the term *domain specific language*, the definitions presented by different authors are very similar:

- Martin Fowler [3] defines DSLs as follows: "a computer programming language of limited expressiveness focused on a particular domain."
- For Markus Voelter et al. [20], a DSL is "simply a language that is optimized for a given class of problems, called a domain" and is "based on abstractions that are closely aligned with the domain for which the language is built."
- Eelco Visser [19] describes a DSL as "a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain."

To consolidate these definitions, a DSL can be first of all characterized by begin a *language*, or more specifically a *computer programming language*. Its primary usage is to allow humans to instruct a computer to perform a certain action. However, contrary to a *general-purpose language* like Java or Ruby, a DSL only has a *limited expressiveness* and provides abstractions for a *particular domain*. In other words, a DSL only supports a small amount of features and syntax which are tailored to the domain where it should be employed.

Similar to the different definitions for DSLs, different approaches to classify DSLs exist, too. This thesis is based on the terms and classification of Martin Fowler [3] who distinguishes DSLs into three categories:

- **External DSLs** are separate from the main language of the application and usually have a custom syntax. They therefore have to be parsed by the host application in order to execute them.
- **Internal DSLs** (often also referred to as domain specific embedded languages [10]) use capabilities of the general-purpose language of the application to try to offer the feeling of a custom language. The code of the DSL is valid code in its general-purpose language as well, so no additional parsing is necessary.

- **Language Workbenches** offer environments for defining and building DSLs as well as writing scripts for the DSLs. Since language workbenches do not play any role in this thesis, they will not be given further attention.

For all these types of DSLs, the boundary which determines whether something is or is not a DSL is quite blurry. According to Mernik et al. [10] it is helpful to think of DSLs in terms of a scale where for both types of DSLs different extremes lie on each end of this scale.

For internal DSLs, the distinction has to be made between a normal *application programming interface* (API) and an actual internal DSL. For Fowler [3] the difference lies in the nature of a DSL to define a new language in form of a grammar. The documentation of an API can offer a good indication whether the module or library exposes a normal API or a DSL. In the case of APIs, methods usually can be documented by themselves and therefore have a self-sufficient meaning. In a DSL, however, methods usually do not hold any meaning by themselves but can only be interpreted in context of a larger expression.

```
apiTestClient.perform(get("/users"))
    .andDo(print())
    .andExpect(status().isOk());
```

Listing 1: The Spring framework offers internal DSLs for testing purposes.

Listing 1 depicts code using a testing library offered by the Spring framework to check the behaviour of a RESTful backend. The testing library offers a variety of static methods combined with elegant method chaining to fluently define a test. For instance, the *andDo* method expects an object that implements a *ResultHandler* interface. The static method *print* constructs such an instance and passes it to the *andDo* method. It is therefore apparent that a standalone executing of the *print* method would not result in anything meaningful. The *print* method as well as the *andDo* method can only be reasonably evaluated when they are both combined with each other.

On the other hand, listing 2 shows the definition and usage of a *builder pattern* to create instances of a hypothetical *Person* class. In this case it is arguable whether *PersonBuilder* exposes an internal DSL. Each method of the builder such as *name* or *age* can be independently described by setting an attribute of the resulting person, i.e. each method has a self-sufficient meaning by itself. Additionally, except having to call *newPerson* at the beginning and *build* at the end, the creation of a new person is not dependent on any grammar which an actual language should be composed of.

For external DSLs one has to differentiate between a DSL and a general-purpose language, though the boundary is not as blurry as with internal DSLs. In their work, Voelter et al. [20] provide a table of characteristics for programming languages depicted in table 1. While both general-purpose languages and DSLs can and will have characteristics of both columns, actual DSLs should possess more properties from the third than from the second column. A good example presented by Martin Fowler [3], where the distinction between DSL an general-purpose language is not as clear, is the *R language*[3]. R is a programming language for statistical computing and is therefore generally focused on a particular domain. Despite

[3]https://www.r-project.org/

```java
public class PersonBuilder {
    private String name;
    private Integer age;
    private String placeOfBirth;

    public static PersonBuilder newPerson() {
        return new PersonBuilder();
    }

    public PersonBuilder name(String name) {
        this.name = name;
        return this;
    }

    public PersonBuilder age(Integer age) {
        this.age = age;
        return this;
    }

    public PersonBuilder placeOfBirth(String placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
        return this;
    }

    public Person build() {
        return new Person(this.name, this.age, this.placeOfBirth);
    }
}

// Usage
PersonBuilder.newPerson()
    .name("John Doe")
    .age(21)
    .build();
```

Listing 2: It is arguable whether *PersonBuilder* can be considered to be a DSL.

that, R offers features beyond this scope and is *Turing-complete*, i.e. it offers mechanisms for control flow such as loops or conditions combined with the possibility to define variables and functions. The language can be (and is!) therefore employed for purposes it was not

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Table 1: Characteristics of general-purpose languages (GPLs) and DSLs [20]

initially intended for. Thus, though it partly complies with the characteristics of a DSL, it should be categorized as a general-purpose language. A popular and widely spread example for an actual external DSL is *regular expressions*. Its domain is very small and well-defined (matching text), it is not Turing-complete, and it offers only the amount of features and syntax to excel for its purpose.

2.2 Benefits and Problems of Domain Specific Languages

After having DSLs defined and categorized, the question arises why developers of software systems should actually build and use DSLs. What are potential benefits as well as problems of DSLs? By weighing each of the advantages and downsides of DSLs, software professionals will be able to decide whether or not a DSL could potentially help to solve a certain problem.

The following advantages are often presented to support the usage of DSLs:

- **Productivity:** Since DSLs are specialized to express a certain aspect of a system, the code of the DSL will be easier and faster to write, read, and understand due to the fact that less code is necessary to solve a problem [20]. Furthermore, through its limited expressiveness DSLs are much more restrictive which leads to both making less mistakes as well as fixing defects more quickly [3]. Voelter et al. [20] even argue that DSLs may be so restrictive that it is impossible to write invalid expressions at all.

- **Communication:** Good communication in software projects is, according to research (see [17]), a very important critical success factor for projects to succeed. Since software professionals develop systems for a wide variety of industries, they have often to be in contact with experts of the particular industry, so called *domain experts*. Due to their specialized syntax, DSLs offer the possibility for domain experts to read and correct source code and therefore highly improve the communication between tech and non-tech project stakeholders [3].

- **Platform Isolation:** Eventhough the following advantage generally applies to external DSLs only, it is an interesting argument to be made. Since external DSLs have a custom syntax and can be parsed and executed by a host language, the DSL itself is often not tied to a certain platform [20]. For most external DSLs it does not matter whether its code is parsed and executing using for example Java or C#. Therefore, external DSLs allow their code to be migrated in case companies switch to different general-purpose languages or execution platforms. It will be apparent later on in this thesis that GraalVM is able to expand this advantage even further.

- **Alternative Computational Model:** Most general-purpose languages follow the *imperative style* of computation: the computer is told what to do in a certain sequence with features such as control flow and variables. For some problems, however, different approaches are more suitable and easier to utilize. Build automation is one of these problems: build tools such as *Apache Maven*[4] generally offer a *declarative style* to describe the build of a software system. Instead of focusing on *how* something should be done, the declarative style of programming concentrates on *what* should happen, leaving the *how* to a different layer of the system. According to Martin Fowler [3], DSLs offer

[4]`https://maven.apache.org/`

a resembling advantage since it is also possible to employ a different computational model than the main language of the application with which it is easier to express or define certain aspects of the respective domain.

Contrary to these advantages, the usage of DSLs also comes with some problems and threats. Among them are the following:

- **Language Cacophony:** This term was coined by Martin Fowler [3] and states that learning new languages is generally hard. Therefore, it is apparent that combining multiple languages for a project complicates the development compared to only using a single language. It is therefore necessary to determine whether or not learning a DSL is less costly opposed to understanding and working on the problems at hand without a DSL.

- **Cost of Building:** The most obvious problem of creating a DSL is the initial cost of building it. However, not only the initial costs of implementing the DSL has to be taken into account. Throughout time the DSL has to be maintained and extended as well. Voelter et al. [20] emphasise that in order for a language to remain relevant it has to be actively maintained and evolved to not become a liability. Moreover, according to Fowler [3], it is not common for developers to know the techniques which are necessary to build DSLs which further aggravates the cost of implementing one. This cost of building can of course be mitigated if the DSL is reused throughout different projects.

- **Inflexibility:** According to Voelter et al. [20], investing in reusable artifacts locks businesses into a certain way of operation. When using a DSL, especially if the usage thereof leads to productivity gains, the company could hold onto its DSL for too long or even extend it furtherly. Martin Fowler [3] describes this issue as the *ghetto language problem*, where a language, built in-house, is being utilized in more and more systems of the company as well as being continually extended with features. In the long run, this will lead the company to be inflexible regarding technological innovations and shifts in the industry as well as making it harder to hire staff. As a consequence, Voelter et al. [20] recommends businesses to keep an open mind and to throw things overboard, if necessary.

- **Blinkered Abstraction:** Another problem Martin Fowler [3] highlights is the situation where developers are too confident about their DSL and try to fit the world to work with their language, instead of changing the language in accordance to the world. Thus, software professionals must view their DSL to be constantly under development, instead of regarding it as being finished.

As a conclusion, there are two possible reasons not to use a DSL. First, in case none of the benefits of a DSL applies to the problem at hand it is naturally not a fitting tool to solve that problem. Secondly, if the costs and risks of building a DSL outweigh its potential benefits. Otherwise it can be worthwhile to consider building or using a DSL to benefit from the potential prospects as set out in this section.

2.3 Development Processes for Domain Specific Languages

A frequently cited approach in research for the development of DSLs was posed by Mernik et al. [10] which divides the creation of DSLs in five phases: *decision*, *analysis*, *design*, *implementation*, and *deployment*. These phases should not be viewed as strictly sequential; in case questions or problems arise related to earlier phases of the development cycle, developers should step back again to solve these issues.

In the first step, companies or development teams should first decide whether or not the creation of a DSL will help them solve their problem. This should include a cost analysis or a research to determine whether similar DSLs already exist which could be reused. Furthermore, benefits and risks, as the ones illustrated in the previous section, can be taken into account to decide whether the usage of a DSL could be worthwhile.

After having decided to implement a new DSL, the analysis phase consists of gathering knowledge about the respective domain. According to Mernik et al. [10] this might include questioning domain experts, studying documents or other sources of information, as well as conducting customer surveys. The aim of this phase is to be able to describe important concepts of the domain, to be familiar with the terminology of the domain, as well as to understand its semantics.

The third step in the process, the design phase, consists of determining first whether to build an internal or an external DSL. Both types of DSLs are accompanied by various advantages and benefits which will become more clear in the upcoming section that explains approaches to implement both types of DSLs. Afterwards, the DSL designers have to specify their language design either *informally* or *formally*. The informal design is generally a description in natural human language supplemented by illustrative programs written in the intended DSL. On the other hand, the formal design consists of a concrete specification of the language using special notations. As examples for these notations, Mernik et al. [10] propose regular expressions and grammars to define the language syntax and *abstract rewriting systems* or *abstract state machines* to specify its semantics. Since these notations are out of the scope of this work they will not be explained in further detail here.

After having established the design of the DSL, the final two steps, implementation and deployment, can be conducted. As previously mentioned, the former will be illustrated in the upcoming section. Regarding the latter, Voelter et al. [20] highlight that it is very important to view the implemented DSL as any other product of the company. This means that the DSL has to have concrete release schedules where reported issues must be fixed and resolved. Documentation and support staff should be available for the DSL to help in case problems arise. Viewing the DSL as a product therefore leads to a higher acceptance which is critical for its successful deployment.

Contrary to the aforementioned proposal by Mernik et al. [10] which is very similar to the traditional waterfall model, Voelter et al. [20] suggest a more iterative process to develop the language. Developers should first focus on a small part of the domain, acquire knowledge for only this part, and then immediately build the corresponding part of the DSL. Only after having finalized this piece of the DSL, the developers should move to new requirements. Naturally, this approach can only be successful if it is paired with regular refactoring of the language whenever the understanding of the domain was deepened.

Developing DSLs using GraalVM generally does not differ from the processes which were outlined in this section since approaches like these are largely tool agnostic. Nonetheless, GraalVM impacts the decision making regarding which type of DSL should be implemented and how it should be built. How this impact comes into practice will be showcased in chapter 5 where the realization of the DSL for mathematical expressions on top of GraalVM is explained in detail.

2.4 Implementation of Domain Specific Languages

In order to compare and evaluate the implementation of DSLs with the frameworks offered by GraalVM, an overview of how DSLs can be built without additional technologies is necessary. The following section explains how internal and external DSLs can be implemented. For both types, a language for the same and rather simple problem will be built. The Java SDK ships with a powerful timer facility to schedule tasks for future and recurring execution. A *TimerTask* defines such a task which can be run once or repeatedly in the future. Listing 3 displays how a TimerTask can be created and scheduled. In this example, the string *Hello World* will be printed periodically every 1000 milliseconds with a delay of 5000 milliseconds. If the last parameter is omitted, *Hello World* would be only printed once after 5000 milliseconds have elapsed.

```java
var timer = new Timer();

timer.schedule(new TimerTask() {
    @Override
    public void run() {
        System.out.println("Hello World");
    }
}, 5000, 1000);
```

Listing 3: After five seconds print *Hello World* every second by using a TimerTask.

The internal and external DSLs which will be presented in the further course of this section will serve as a layer on top of this API and will enable developers to schedule tasks in a more fluent manner. The primary objective of both upcoming DSLs, however, is to illustrate prevalent approaches to implement both types of DSLs.

2.4.1 Internal Domain Specific Languages

Internal DSLs are generally more approachable than external DSLs due to the fact that external DSLs require more techniques such as grammars and parsers in order to build them. On the flip side, internal DSLs are largely constrained by their host language. There are general-purpose languages such as Ruby or Lisp which are very flexible regarding their syntax or offer specialized functionalities, such as macros in Lisp, to create custom languages. Other programming languages like Java or C++ have more restrictive syntactic rules in comparison which affects the look and feel of internal DSLs.

To build and structure internal DSLs different approaches exist and are employed. However, since this thesis covers GraalVM, a technology based on Java, a common way to build internal DSLs using *object-oriented programming* (OOP) will be illustrated. To create internal DSLs using an OOP host language, Martin Fowler argues [3] that the DSL itself and the actual objects which the DSL utilizes should be separate from each other. Internal DSLs should be built in form of so called *expression builders* which should not define any domain logic but only offer constructs to build expressions of the DSL. The actual logic should be located in another layer hidden behind the expression builder which the builder utilizes once the DSL expression should be executed. This approach enables separate testing of the domain logic and the expression builder as well as the possibility to replace the expression builder with an external DSL if necessary. In the context of the timer scheduling DSL, the Java timer API represents the layer of the domain logic while a separate layer of expression builders has to be implemented.

Listing 4 depicts some DSL expressions which exemplify how the internal timer scheduling DSL should look like. The timer itself is configured using an API similar to a builder pattern while static methods act as descriptive parameters, like setting what the timer should execute or the delay of the timer.

```
timer()
    .execute(print("Hello World repeatedly!"))
    .repeatedly()
    .every(minutes(1))
    .after(seconds(30))
    .setup();

timer()
    .execute(print("Hello World once!"))
    .once()
    .after(seconds(10))
    .setup();

timer()
    .execute(print("Hello World once now!"))
    .once()
    .rightNow()
    .setup();
```

Listing 4: Some expressions to schedule future and potentially periodic tasks.

The implementation of the static methods for the different units of time and for the timer tasks is rather short so they will be attended to first. Listing 5 and 6 depict two classes which are structured in a similar fashion. Both classes are final and therefore cannot and should not be extended. Furthermore, both have private constructors to prohibit the creation of instances of both classes. The implementation of the *Duration* class is self-explanatory and converts different units of time to milliseconds, since the Java SDK expects milliseconds for the scheduling of timers. Static methods of the *Tasks* class should create instances of the *TimerTask* class offered by the Java SDK which will be scheduled and executed after the configuration of the timer has completed. In this example only a simple *print* task exists, though more complex tasks like syncing databases or sending emails would be possible.

The method chaining with which the timer is constructed is built using separate classes. Each class offers the developer one or more possibilities to configure the timer and returns an instance of a new class which defines the next step of configuration. Each step therefore acquires a part of the configuration and passes it on to the next step. In the final step and

```java
public final class Duration {
    private Duration() {}

    public static long seconds(long n) {
        return n * 1000;
    }

    public static long minutes(long n) {
        return seconds(60 * n);
    }

    public static long hours(long n) {
        return minutes(60 * n);
    }
}
```

Listing 5: *Duration* offers static methods for different units of time.

```java
public final class Tasks {
    private Tasks() {}

    public static TimerTask print(String message) {
        return new TimerTask() {
            @Override
            public void run() {
                System.out.println(message);
            }
        };
    }
}
```

Listing 6: *Tasks* offers static methods for different timer tasks, here only a print task.

class, all the obtained information is used to configure and schedule an actual timer using the Java API. The first class in this hierarchy is shown in listing 7. It offers the static *timer* method which was the initial method with which each DSL expression has to start according to the language design of listing 4. This method creates the actual instance of the builder class which only possesses one instance method called *execute*. Since *execute* expects an instance of type *TimerTask,* it fits perfectly to the static methods of the *Tasks* class from listing 6 which should return predefined objects of type *TimerTask*.

The *execute* method creates an instance of another class called *TimerExpressionBuilder-WithTask* which is displayed in listing 8 and defines the next possible steps of the timer configuration. The developer can choose between either calling *repeatedly* or *once* which both create different subsequent objects to differentiate between a timer task that should be executed only once and one that should be run multiple times. All remaining steps and expression builder classes follow a similar structure and can be viewed in listing 116 and 117 of the appendix.

Since each step of the DSL is in a separate class, the type system makes it impossible

```java
public final class TimerExpressionBuilder {
    private TimerExpressionBuilder() {}

    public static TimerExpressionBuilder timer() {
        return new TimerExpressionBuilder();
    }

    public TimerExpressionBuilderWithTask execute(TimerTask task) {
        return new TimerExpressionBuilderWithTask(task);
    }
}
```

Listing 7: *TimerExpressionBuilder* defines the starting point of the DSL.

```java
public final class TimerExpressionBuilderWithTask {
    private final TimerTask task;

    public TimerExpressionBuilderWithTask(TimerTask task) {
        this.task = task;
    }

    public RepeatableTimerExpressionBuilder repeatedly() {
        return new RepeatableTimerExpressionBuilder(this.task);
    }

    public SingleTimerExpressionBuilder once() {
        return new SingleTimerExpressionBuilder(this.task);
    }
}
```

Listing 8: *TimerExpressionBuilderWithTask* marks the next step of configuration of the timer.

to create invalid DSL expressions. The DSL therefore serves as a good example for the advantage mentioned in section 2.2 which specified that the usage of restrictive DSL offer productivity improvements by making it impossible to write invalid code. If all methods would be defined in a single class, a developer could potentially call the methods *once* and *repeatedly* after each other which would result in ambigous code. Furthermore, considering that code completion is offered by nearly every *integrated development environment* nowadays, the developer is piloted through the creation of the expression, since the code completion will only offer the next methods according to the hierachy of the expression builder classes.

Although different approaches exist to build internal DSLs (as an example see [4]) this section does not aim to compare techniques to implement internal DSLs but to unveil their characteristics. Since the DSL piggybacks on Java, it is clear that interacting with the DSL is rather straightforward. Executing the DSL is not different from executing Java code; data and objects that are passed between the DSL and Java, such as the configuration of a timer, do not have to be translated in any way since both share the same runtime. The next section will showcase that these advantages are not as easily available when using external DSLs. Yet, through its functionalities, GraalVM is able to blur the line between both of these approaches as will become apparent during the implementation and evaluation of the DSL for mathematical expressions.

2.4.2 External Domain Specific Languages

External DSLs compared to internal ones come with a much greater syntactic freedom. This liberality concerning the syntax, however, goes along with a more complex implementation. The basic principles with which external DSLs are build are very similar to the ones of general-purpose languages, though developers of DSLs do not have to know the techniques as in depth as general-purpose language developers. Interestingly, according to Bob Nystrom [11], the techniques with which languages are build have not really changed since the early days of computing.

Before explaining the approach with which the external DSL for scheduling timers is implemented, the structure and syntax of the intended language will be presented first. Listing 9 presents some example code of the external DSL. It is apparent that the syntax of the DSL does not follow the syntactic rules of Java anymore. Timers are grouped in *timer* and *end* pairs and allow the same configurable features as with the internal DSL.

```
timer
  print "Hello World"
  repeatedly
  every 30 seconds
  after 2 minutes
end

timer
  print "Hello World once!"
  once
  after 10 seconds
end

timer
  print "Hello World now!"
  once
  right now
end
```

Listing 9: Some external DSL expressions to schedule future and potentially periodic tasks.

To build this DSL, a process based on Bob Nystrom's online book *Crafting Interpreters* [11] was employed. The book uses widespread techniques to build languages which are also highlighted in Fowler's work about DSLs [3]. This process divides the evaluation of language expressions into at least three steps.

The first step is called *lexing*. A *lexer* takes the code of the language and splits it into individual tokens. A token is a data structure which is associated to a certain type and might contain a value. Listing 10 lists all types of tokens of the DSL as an enum. Every keyword is a different token type, in addition to the two datatypes which the DSL supports: strings and numbers. Lastly, an *EOF* token type marks the end of the source code.

```
public enum TokenType {
    TIMER, REPEATEDLY, ONCE, RIGHT, NOW,
    PRINT, AFTER, EVERY, STRING, NUMBER,
    SECONDS, MINUTES, HOURS, END, EOF
}
```

Listing 10: All types of tokens of the DSL.

The token itself is a simple class with, as previously mentioned, attributes for the type of the token and the value. It is presented in listing 11. Note that the value will be *null* for most types of tokens except strings and numbers since keywords do not hold any literal values.

```
public class Token {
    private final TokenType type;
    private final Object value;

    public Token(TokenType type, Object value) {
        this.type = type;
        this.value = value;
    }

    public TokenType getType() {
        return type;
    }

    public Object getValue() {
        return value;
    }
}
```

Listing 11: The *Token* class for the lexer.

The lexer moves character by character through the source code, tries to identify tokens, stores them in a list, and in the end returns that list of tokens. Listing 12 depicts the basic structure of such a lexer. The attributes include the start position of the current read as well as the end position, the source code itself, and the list of tokens which will be returned in the end.

As long as the lexer has not reached the end of the source code, i.e. the start position is greater than the length of the source code, the lexer tries to read the next token. Listing 13 illustrates how the lexer identifies the next token. By comparing the character of the current position, the lexer can judge what it will expect as a next token. If for example the current character is a double quote, the lexer can assume that the next token should be a string.

After the decision has been made regarding the expectation of the next token, the lexer tries to find the end of this token. Listing 14 shows how this is accomplished for strings.

With the help of the peek method which returns the character of the current end position, the lexer is able to find the end of the string by searching for the second double quote. In case it reaches the end of the source code before finding the second double quote, the lexer

```
  public class Lexer {
      private int startOfToken = 0;
      private int endOfToken = 0;
      private final String code;
      private final List<Token> tokens = new ArrayList<>();

      public Lexer(String code) {
          this.code = code;
      }

      public List<Token> getTokens() throws TimerDSLException {
          while (!isAtEnd()) {
              readNextToken();
              this.startOfToken = this.endOfToken + 1;
              this.endOfToken = this.startOfToken;
          }

          tokens.add(new Token(EOF, null));
          return tokens;
      }
}
```

Listing 12: Basic structure of the *Lexer* class.

```
private void readNextToken() throws TimerDSLException {
    var nextChar = code.charAt(this.startOfToken);

    if (List.of(' ', '\r', '\t', '\n').contains(nextChar)) {
        // Ignore whitespaces
    } else if ('"' == nextChar) {
        string();
    } else if (isDigit(nextChar)) {
        number();
    } else if (isAlpha(nextChar)) {
        keyword();
    } else {
        throw new TimerDSLException("Unexpected character");
    }
}
```

Listing 13: The lexer identifies the next token by checking the first character of the next token.

throws an exception, otherwise the value of the string is extracted from the source code and saved as a string token in the list of tokens.

The approach for identifying numbers or keywords is very similar and can be viewed in the complete definition of the lexer class in listing 118 and 119 of the appendix.

In the second step of the whole evaluation, a *parser* takes this list of tokens to generate an *abstract syntax tree* (AST) according to the grammatical rules of the language. The grammar is generally a *context-free grammar* (CFG) which is often notated in a flavour of the *Backus-Naur form* (BNF). Listing 15 illustrates how a grammar could be defined using a version of the BNF which Bob Nystrom uses in his work [11].

```
private void string() throws TimerDSLException {
    endOfToken++;
    while (peek() != '"' && !isAtEnd()) endOfToken++;

    if (isAtEnd()) throw new TimerDSLException("Unterminated string");

    endOfToken++;
    var value = code.substring(startOfToken + 1, endOfToken - 1);
    tokens.add(new Token(STRING, value));
}
```

Listing 14: The lexer tries to find the end of the string to then get the value between the start and end position.

```
pizza   → crust "with" cheese "and" (topping "and" | topping)+
crust   → "thin crust" | "thick crust"
cheese  → "mozzarella cheese" | "provolone cheese"
topping → "mushrooms" | "extra cheese" | "salami" | "ham"
```

Listing 15: A simple grammar for configuring pizzas

A CFG has *terminals* and *nonterminals*. A terminal is like a literal value of the grammar, for example *mozzarella cheese* or *mushrooms*. Terminals mark end points and cannot be replaced with more symbols. Nonterminals on the other hand are references to other rules which allow the construction of more complex expressions. The *pizza* nonterminal is the starting point of the grammar with a *crust* nonterminal at the beginning. The *crust* nonterminal offers two possible terminals (specified by the | sign): either a *thin crust* or a *thick crust*. At the end of the pizza nonterminal there are again two possibilities. It is either allowed to choose a topping combined with an *and* terminal (to be able to have multiple toppings) or just a single topping. The + sign specifies the same as with regular expressions. It marks that a certain rule can occur once or more times while a * would indicate that a rule can be utilized zero or more times. The parentheses group these possibilities regarding the toppings together and signify that the + sign can only be applied to the toppings. This way an arbitrary amount of toppings is possible. The following sentences would be valid according to the grammar:

- thin crust with mozzarella cheese and mushrooms
- thick crust with provolone cheese and salami and ham and mushrooms

With the help of a grammar, it is also possible to represent an expression in form of a tree, the AST. Figure 2 visualizes the second sentence from above in the form of an AST which conforms to the defined grammar.

The main task of the parser in the process presented by Nystrom [11] is to build an AST representation of the tokens for easier future processing. To understand how such a parser can be build, the implementation of a parser for the timer scheduling DSL will be subsequently illustrated. Listing 16 depicts a possible grammar for the DSL (as seen in listing 9) in BNF.

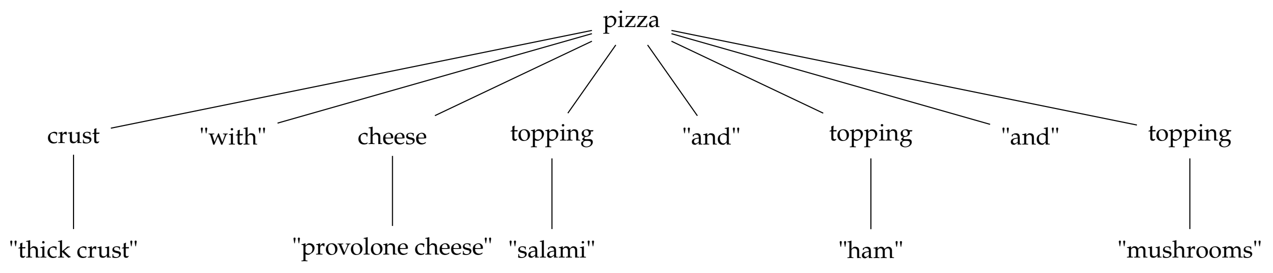


Figure 2: *thick crust with provolone cheese and salami and ham and mushrooms* represented as an AST

```
program             → timer_stmt+
timer_stmt          → "timer" command (once_timer | repeated_timer) "end"
command             → "print" STRING
once_timer          → "once" after_configuration
repeated_timer      → "repeatedly" "every" NUMBER time_unit after_configuration
after_configuration → "right" "now" | "after" NUMBER time_unit
time_unit           → "seconds" | "minutes" | "hours"
```

Listing 16: The grammar of the external timer DSL in BNF.

A program written in the DSL consists of one or more *timer statements*. Each *timer statement* has to start with the terminal *timer* and has to end with the terminal *end*. Between *timer* and *end*, the first expected nonterminal is the command. Currently only the *print* command is supported which expects a string. After the command, two different possiblities exist to configure the timer: a *once timer* and a *repeated timer*. The *once timer* only expects a configuration for the delay of the command while the *repeated timer* expects the configuration of the period of the command in addition.

The implementation is surprisingly simple, once well understood. The first step is to define the AST datastructure. Listing 17 shows the root element of the tree: a *timer statement*. The class has two attributes which resemble the children of the root: a *command* and the configuration of the timer.

Since for the purposes of this example only a *print* command is supported, the command class is rather simple, although it is laid out to be extended at will. Listing 18 depicts the *Command* class which is abstract and which includes the *print* command as a static nested class. Naturally, the *print* command only has one "child" which is the message it should print.

The *timer configuration* has a resembling structure and is presented in listing 120 in the appendix for the sake of completion. The *TimerConfiguration* class itself is again abstract but has different subclasses. Analogous to the grammar, a timer configuration is either a *once timer* or a *repeated timer*. The *once timer* has only a time setting for the delay, while the *repeated timer* has an additional time setting for the period.

It is noticable that the composition of the AST is very similar to the composition of the grammar. This is due to the AST being a representation of the syntactic structure of the code, as previously mentioned. The question now arises, however, how the AST of some

```java
public class TimerStmt {
    private final Command command;
    private final TimerConfiguration configuration;

    public TimerStmt(Command command, TimerConfiguration configuration) {
        this.command = command;
        this.configuration = configuration;
    }

    public Command getCommand() {
        return command;
    }

    public TimerConfiguration getConfiguration() {
        return configuration;
    }
}
```

Listing 17: The root element of the AST.

```java
public abstract class Command {
    public static class PrintCommand extends Command {
        private final String message;

        public PrintCommand(String message) {
            this.message = message;
        }

        public String getMessage() {
            return message;
        }
    }
}
```

Listing 18: All commands are subclasses of the *Command* class.

concrete code can actually be constructed. To address this problem, Bob Nystrom presents a popular technique in his work [11] which is called *recursive descent*. In simple words, recursive descent parsing is a translation of the grammar into programming language code. Many of today's programming language implementations are based on the recursive descent parsing technique, such as the GCC or the Roslyn C# compiler [11].

As presented in listing 19, the parser for the timer DSL has only two attributes: the list of tokens and the current position of the parser in this aforementioned list.

As was mentioned, the recursive descent technique is a translation of the grammar into code. The first rule of the grammar specifies that a program consists of one or more timer statements. Therefore the method with which the parser will be called has to reflect this rule, as shown in listing 20.

First, a new list of timer statements, which are the root nodes of the AST, is created. The same list will be returned at the end of the method. Afterwards, since the rule expects at

```java
public class Parser {
    private final List<Token> tokens;
    private int current;

    public Parser(List<Token> tokens) {
        this.tokens = tokens;
        this.current = 0;
    }
}
```

Listing 19: The basic structure of the timer DSL parser.

```java
// program → timer_stmt+
public List<TimerStmt> parse() throws TimerDSLException {
    var timerStatements = new ArrayList<TimerStmt>();
    timerStatements.add(timerStmt());

    while (!isAtEnd()) {
        timerStatements.add(timerStmt());
    }

    return timerStatements;
}
```

Listing 20: The initial method with which the parser will be called and which reflects the first rule of the grammar.

least one timer statement, the code also adds at least one element to that list. Subsequently, additional timer statements are added to the list until the end of the list is reached, i.e. an *EOF* token is encountered.

The *timerStmt* method corresponds to the next rule in the grammar and is outlined in listing 21.

Throughout the parser there are two helpful methods: *consume* and *match*. The *consume* method expects a token of a certain type at the current position. In case the type of the current token corresponds to this expected type, the token is returned and the *current* attribute of the parser incremented, if not then an exception with a given message is thrown. The *match* method, however, only returns a boolean which is true if the given type is equal to the type of the current token. It does not change the position of the parser inside the list of tokens. Therefore, in the first line of the *timerStmt* method a *TIMER* token is expected, since every timer statement has to start with the *timer* keyword. In case no *TIMER* token exists at that position, an exception is thrown with the message *Expected 'timer' at the beginning of definition.* Since the *command* nonterminal follows the *timer* keyword, the method calls a *command* method in the next step which handles the *command* rule. After the command, there are two possibilities: either a *once timer* or a *repeated timer* configuration. Since the once timer has to start with the *once* keyword, the method checks whether the current token is of type *ONCE*. If yes it calls the *onceTimer* method, otherwise it expects a *REPEATEDLY* token and calls the corresponding method. At the end of the timer statement, the *END* token must

```
// timer_stmt → "timer" command (once_timer | repeated_timer) "end"
private TimerStmt timerStmt() throws TimerDSLException {
    consume(TIMER, "Expected 'timer' at the beginning of definition.");

    var command = command();

    TimerConfiguration config = null;

    if (match(ONCE)) config = onceTimer();
    else {
        consume(REPEATEDLY, "Expected 'once' or 'repeatedly' after command.");
        config = repeatedTimer();
    }

    consume(END, "Expected 'end' at the end of definition.");

    return new TimerStmt(command, config);
}
```

Listing 21: The *timerStmt* method which corresponds to the *timer_stmt* rule of the grammar.

be consumed and the whole timer statement is returned.

```
// command → "print" STRING
private Command command() throws TimerDSLException {
    consume(PRINT, "Expected 'print' command.");
    var message = consume(STRING, "Expected 'string' after 'print'.");
    return new PrintCommand((String) message.getValue());
}
```

Listing 22: The *command* method currently only has the print command as a possibility.

As a final example, listing 22 depicts the *command* method. The method first expects a *PRINT* token, followed by a *STRING* token. In the case of the string, the returned token of the *consume* method is actually saved in a variable, to pass it to the *PrintCommand* AST element. The remaining methods of the recursive descent parser (see 121 in the appendix) for this DSL work very similar to the examples that were presented above. All methods of the parser correspond to one rule of the grammar. The parser then utilizes these methods to descent recursively according to the grammar to construct an AST in the end.

The final step of the processing of the DSL is to walk through the AST returned by the parser and interpret it. Since this DSL is rather simple, the *interpreter* is implemented using a very naive approach. The basic structure of the interpreter is visible in listing 23.

The interpreter receives the list of statements, which was built by the parser, through its constructor. It has a public *interpret* method which iterates over each statement and evaluates it. Listing 24 shows the *evaluate* method which accepts a single timer statement and performs the actual evaluation.

The interpreter first "walks" to the *command* node of the statement to build an instance of the *TimerTask* class provided by the JDK. It then checks whether the configuration is a

```
public class Interpreter {
    private List<TimerStmt> timerStatements;

    public Interpreter(List<TimerStmt> timerStatements) {
        this.timerStatements = timerStatements;
    }

    public void interpret() throws TimerDSLException {
        for(var stmt: timerStatements) {
            evaluate(stmt);
        }
    }
}
```

Listing 23: The basic structure of the interpreter.

```
private void evaluate(TimerStmt stmt) throws TimerDSLException {
    var timer = new Timer();
    var timerTask = buildTask(stmt.getCommand());

    if (stmt.getConfiguration() instanceof TimerConfiguration.OnceTimer) {
        var onceTimer = (TimerConfiguration.OnceTimer) stmt.getConfiguration();
        timer.schedule(
            timerTask,
            getMillis(
                onceTimer.getAfterSetting().getNumber(),
                onceTimer.getAfterSetting().getUnit()
            )
        );
    } else {
        var repeatedTimer = (TimerConfiguration.RepeatedTimer) stmt.getConfiguration();
        timer.schedule(
            timerTask,
            getMillis(
                repeatedTimer.getAfterSetting().getNumber(),
                repeatedTimer.getAfterSetting().getUnit()
            ),
            getMillis(
                repeatedTimer.getEverySetting().getNumber(),
                repeatedTimer.getEverySetting().getUnit()
            )
        );
    }
}
```

Listing 24: The *evaluate* method with which a statement is evaluated by the timer.

once timer or a *repeated timer* and schedules the timer using the remaining nodes of the AST. Listing 25 presents the remaining methods of the interpreter which are used by the *evaluate* method.

It is noticable that the *instanceof* checks could make the code of the interpreter quite obscure if the DSL is much more complex. For this reason, Bob Nystrom presents the *visitor*

```java
private TimerTask buildTask(Command command) throws TimerDSLException {
    if (command instanceof Command.PrintCommand) {
        var message = ((Command.PrintCommand) command).getMessage();
        return new TimerTask() {
            @Override
            public void run() {
                System.out.println(message);
            }
        };
    } else throw new TimerDSLException("Unknown command type");
}

private long getMillis(long number, TimerConfiguration.TimeUnit unit) {
    if (unit == TimerConfiguration.TimeUnit.SECONDS) {
        return number * 1000;
    } else if (unit == TimerConfiguration.TimeUnit.MINUTES) {
        return number * 1000 * 60;
    } else {
        return number * 1000 * 60 * 60;
    }
}
```

Listing 25: Remaining methods of the interpreter.

pattern in his work [11] as a possibility to cleanly structure the interpreter without having to resort to *instanceof* checks when walking through the AST. However, in the case of this simple DSL, the visitor pattern would have been overkill as a solution.

Since the demonstrated approach in this section is very intensive regarding the amount of code that has to be written, naturally, tools to aid creating external DSLs have been released in the past. Technologies such as *YACC* (Yet Another Compiler-Compiler)[5] or *ANTLR* (ANother Tool for Language Recognition)[6] are able to perform tasks such as taking the grammar of a language as input and produce a lexer and a parser as output. Indeed, to create the DSL for mathematical expressions with GraalVM, the ANTLR library will be utilized to automate some steps outlined in this section. Nonetheless, these tools and the frameworks of GraalVM itself are based on concepts such as CFGs and ASTs. Thus, a very manual approach of creating DSLs was presented to introduce the theoretical background on which the upcoming chapters of this thesis will be based on.

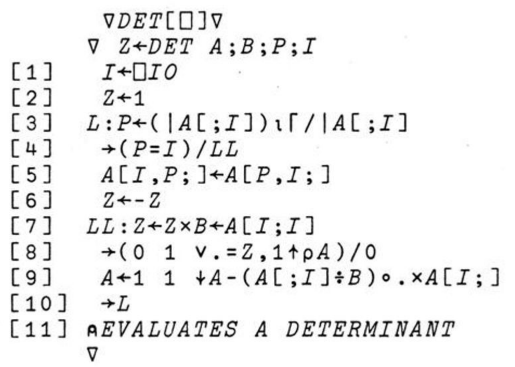
[5]http://dinosaur.compilertools.net/
[6]https://www.antlr.org/

3 Introduction of AML

To reasonably evaluate GraalVM as a platform to build DSLs upon, a new and nontrivial DSL had to be designed to act as an object of study. The following chapter introduces this DSL by outlining its motivation and specifying its features. Furthermore, since the implemented language is an external DSL, its language grammar, as introduced in the previous chapter, will be presented and discussed.

3.1 Motivation

In 1962 a book with the title *A Programming Language* was published by the author Kenneth E. Iverson [6]. In the book a new programming language was presented with the same name as the title of the book: *A Programming Language (APL)*. APL was created to provide a very concise notation for programs related to the area of applied mathematics such as algorithms. It utilises a large spectrum of special graphic symbols where each symbol represents a certain function or operator. As an example, figure 3 depicts a program written in APL which computes the mathematical determinant of a matrix.



```
      ∇DET[⎕]∇
     ∇ Z←DET A;B;P;I
[1]   I←⎕IO
[2]   Z←1
[3]  L:P←(|A[;I])⍳⌈/|A[;I]
[4]   →(P=I)/LL
[5]   A[I,P;]←A[P,I;]
[6]   Z←-Z
[7]  LL:Z←Z×B←A[I;I]
[8]   →(0 1 ∨.=Z,1↑⍴A)/0
[9]   A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]  →L
[11] ⍝EVALUATES A DETERMINANT
     ∇
```

Figure 3: Computes the mathematical determinant of a matrix, copied from [16].

It should be quickly evident that APL code cannot be understood without proper introduction to the language and its symbols. Nonetheless, the language attracted interest both academically and commercially with Iverson receiving the Turing Award later on in 1979 [16].

Similarly to APL, the DSL of this thesis is also aimed at writing code to efficiently solve mathematical problems. Contrary to APL, however, the DSL tries to mimic existing mathematical notation as far as possible instead of inventing a new notation. While it is far from being as extensive (and controversial) as APL, due to its resembling underlying idea, the DSL was named with respect to APL as *A Mathematical Language* or short *AML*.

The rationale behind a mathematically focused language is the deficiency of popular procedural languages to allow programmers to write concise and readable code which executes mathematical operations. To explain this inability more clearly, a simple mathematical computation shall be presented. Let three arbitrary sets A, B, and C be given where the set D is computed by calculating the intersection of A with the set difference of B and C:

$$\begin{aligned} A &= \{1,2,3,4,5\} \\ B &= \{2,3,4,5,6\} \\ C &= \{3,4\} \\ D &= A \cap (B \setminus C) \end{aligned}$$

To perform this task, listing 26 depicts a possibility to determine the set D using Java.

```java
var setA = Set.of(1, 2, 3, 4, 5);
var setB = Set.of(2, 3, 4, 5, 6);
var setC = Set.of(3, 4);

var setD = new HashSet<>(setB);
setD.removeAll(setC);
setD.retainAll(setA);

setD.forEach(System.out::println);
```

Listing 26: Determine the set D using a procedural language such as Java.

Although the actual computation can be inferred from the code, it does not clearly resemble the original mathematical notation anymore. Furthermore, as the complexity of the mathematical task or algorithm increases, the ability of the code to properly represent the mathematical expressions decreases. A possibility for developers to work against this problem is to add comments to the code, highlighting the original intent. As an example, the Java code from listing 26 could be annotated with a comment similar to listing 27.

```java
...

// D = A ∩ (B \ C)
var setD = new HashSet<>(setB);
setD.removeAll(setC);
setD.retainAll(setA);

...
```

Listing 27: Determination of the set D with the help of a comment.

However, instead of using comments which specify the original mathematical definition, it would naturally be simpler for the comment itself to be the source code. AML tries to offer this capability to software developers by using unicode and keeping its syntax as close to mathematical notation as possible. The determination of the set D expressed in AML would result in the source code depicted in listing 28.

Although the usage of unicode and its similarity to mathematical notation are its key characteristics, AML is also able to more clearly represent mathematical expressions due to its immutability. In the Java code in listing 26, it was first necessary to assign D to a copy of the set B. Afterwards, the set operations were performed by mutating the set D. Since mutation is not possible in mathematics, the Java code is therefore not able to properly

```
A ← {1, 2, 3, 4, 5};
B ← {2, 3, 4, 5, 6};
C ← {3, 4};
D ← A ∩ (B \ C);
```

Listing 28: Determination of the set D in AML.

represent the mathematical definition. AML on the other hand prohibits the redefinition of variables and returns for every operation a new value without modifying its inputs.

Thus, the key aim of AML is to help software professionals to write algorithms, or mathematical expressions in general, more easily and closely to their original definition and thereby increasing the readability of the code as well as reducing the possibility of bugs.

3.2 Features

AML incorporates four datatypes: numbers, fractions, booleans, and sets. Mathematical expressions can be composed of varying datatypes although every datatype supports a set of distinct operations.

Numbers can either be integers or decimal numbers; there is no distinction made between the two. A number can be of arbitrary length, therefore computations involving large numbers are possible. Listing 29 depicts the usage of numbers in AML in addition to common operations involving numbers. Comments in AML are represented by a double hyphen at the beginning of a line.

```
-- Common Operations such as addition, subtraction, multiplication,
-- division, modulo, and exponentiation are supported
(((1.5 + 10.4 - 5.9) · -6 ÷ 3) mod 10) ^ 2; -- ⇒ 4.00

-- Unicode operations involving numbers:

-- floor:
⌊1.1⌋; -- ⇒ 1

-- ceil:
⌈1.4⌉; -- ⇒ 2

-- factorial
100!; -- ⇒ 9.33262...991560894146397615651829E+157
```

Listing 29: Numbers and numeric operations in AML.

Fractions are very similar to numbers although in some cases they benefit from a higher precision. They are automatically simplified after each operation and, depending on the type of operation, might be automatically converted to a number. Fractions and numbers are the only datatypes which can be combined for some arithmetic operations. Listing 30 exemplifies the usage and features of fractions.

```
-- Similarly to numbers, all common operations are supported
-- for fractions as well, although they might be converted to
-- numbers when necessary
(((1/2 + 1/3 - 1/6) · -1/2 ÷ 1/6) mod 3) ^ 2; -- ⇒ 4

-- Unicode operations involving fractions:

-- floor:
⌊(7/3)⌋; -- ⇒ 2

-- ceil:
⌈(14/10)⌉; -- ⇒ 2
```

Listing 30: Fractions and their operations in AML.

Booleans are utilized in, and result from, comparisons and logical expressions. In AML only *true* or *false* are actual booleans; there is no implicit conversion from any other datatype to a boolean such as in JavaScript. Listing 31 illustrates how to write logical expressions and how to use booleans in AML.

```
-- ⊤ denotes true
-- ⊥ denotes false

-- AML supports and (∧), or (∨), xor (⊕), and negation (¬)
⊤ ∨ ⊥ ∧ ¬⊤ ⊕ ⊤; -- ⇒ ⊤

-- All usual comparative operations are supported
2 ≤ 2 ∧ 3 ≥ 3 ∧ 1 < 2 ∧ 2 > 1 ∧ 3 = 3 ∧ 2 ≠ 3; -- ⇒ ⊤

-- Logical expression can utilize implication and equivalence
1 < 2 ∧ 2 < 3 ⇛ 1 < 3; -- ⇒ ⊤
2 ≠ 3 ⇔ 3 ≠ 2; -- ⇒ ⊤
```

Listing 31: Logical expressions and booleans in AML.

Sets are the last datatype in AML and contain a unique assemblage of values of any datatype. With sets the power of unicode operations is the most apparent in AML. Listing 32 showcases the capabilities of the set datatype.

To be actually useful, however, AML offers two additional constructs: variables and functions. As previously mentioned, AML is immutable and therefore prohibits the redefinition of both variables and functions. Once the value of a variable or the expression of a function has been set, it remains the same across the lifetime of the variable or function. Variables are very similar to other programming languages and do not offer any special features. After defining a variable it can be utilized in any AML expression as might be desired. The definition and usage of variables is portrayed in listing 33.

Functions on the other hand are different to the ones known in other programming languages. First, functions are first class citizens in AML such as any other value. They can therefore be passed to other functions as a parameter. Secondly, functions can only have

```
-- There are three possibilities to create sets.

-- 1. Using the set literal
{1, 2/3, ⊤, {1, 2}}; -- ⇛ {1, 2/3, ⊤, {1, 2}}

-- 2. Using a range
{1, ..., 10}; -- ⇛ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

-- 3. Using a set builder
{ x ∈ {1, ..., 10} | x mod 2 = 0 }; -- ⇛ {2, 4, 6, 8, 10}

-- Sets can be compared with each other
{1} ⊂ {1, 2} ∧ {1, 2} ⊃ {1} ∧ {1} ⊆ {1} ∧ {1} ⊇ {1}; -- ⇛ ⊤
{1} ⊄ {1} ∧ {1} ⊅ {1} ∧ {1} ⊈ {2, 3} ∧ {2, 3} ⊉ {1}; -- ⇛ ⊤

-- Sets support intersection (∩), union (∪), and difference (\)
{1, 2, 3} ∩ ({3, 4, 5} \ {3}); -- ⇛ {}

-- The cardinality of a set can be determined as well
|{1, 2, 3}|; -- ⇛ 3
```

Listing 32: Capabilities of the set datatype in AML.

```
x ← 2;
x = 2 ⇒ x ^ 2 = 4; -- ⇛ ⊤
```

Listing 33: Defining and using variables in AML.

one expression as their function body, similar to mathematical functions. Lastly, functions offer a few operations which makes them more useful. Listing 34 provides examples for the definition and usage of functions and their operations.

The last feature AML offers is logical quantification. There are three types of quantification: universal, existential, and unique quantification. Universal quantification verifies whether a certain condition holds for all elements of a set and returns a boolean indicating the result of this verification. Existential quantification determines whether at least one element of a set fulfills the condition while the uniqueness quantifier only results to true if a single element satisfies the condition. Listing 35 presents all three types of quantification.

With this set of features, AML is suitable for a variety of applications. For software relying on mathematical operations or algorithms, AML functions could be written which concisely express the underlying mathematical expressions, and could be imported into the host application and combined with the main program logic. Complex logical conditions which determine control flow in applications could be written in AML as well, instead of relying on the capabilites of the general-purpose language. Additionally, calculations, which require arbitrary length of numbers or high precision through the usage of fractions, could be performed in AML instead of rewriting them using APIs such as *BigInteger* in Java.

```
-- Definition and execution of a function
increment: (x) → x + 1;
increment(1); -- ⇒ 2

-- Functions can have a conditional body and can be recursive
fib: (n) →
  if n ≤ 1: n
  otherwise: fib(n - 1) + fib(n - 2);
fib(9); -- ⇒ 34

-- Functions can be composed
decrement: (x) → x - 1;
identity ← increment ∘ decrement;

 -- same as increment(decrement(1))
identity(1); -- ⇒ 1

-- And functions can be iterated
increment100 ← increment ^ 100;

-- same as increment(increment(...(increment(0))...))
increment100(0); -- ⇒ 100
```

Listing 34: Definition and usage of functions and their special operations in AML.

```
even: (x) → x mod 2 = 0;
S ← {1, 2, 3, 4};

-- universal quantification
∀(n ∈ S: even(n)); -- ⇒ ⊥

-- existential quantification
∃(n ∈ S: even(n)); -- ⇒ ⊤

-- unique quantification
∃!(n ∈ S: even(n)); -- ⇒ ⊥
```

Listing 35: Logical quantification in AML.

3.3 Language Grammar

As explained in section 2.4.2, the grammar of an external DSL is an essential concept on which its implementation builds upon. Due to this importance, the grammar of AML will be illustrated in further detail in this section so the implementation of AML on top of GraalVM will be more easily understandable.

The root element of the grammar is represented as the *program* rule. A program consists of at least one *function* or *expression*, however, functions and expressions can appear as much and as interchangeably as needful. Functions are written using an identifier as its name, zero or more parameters, and a single expression as its body. Expressions can be either *if conditions* or *assignments* and every expression has to end with a semicolon. The grammar rules of these basic building blocks of AML are viewable in listing 36.

```
program    → (function | expression)+
function   → IDENTIFIER ":" "(" params ")" "→" expression
params     → IDENTIFIER? | (IDENTIFIER ",")+ IDENTIFIER
expression → (ifcond | assignment) ";"
```

Listing 36: The basic building blocks of AML.

If conditions are easily explained as well. Every if condition consists of a conditional statement marked as the *logicEquivalence* rule, a *thenBranch* and an *elseBranch*. Both the branches are either another if condition, a function composition or a logical expression.

```
ifcond           → "if" logicEquivalence ":" thenBranch "otherwise" ":" elseBranch
thenBranch       → (ifcond | composition | logicEquivalence)
elseBranch       → (ifcond | composition | logicEquivalence)
```

Listing 37: Grammatical rules of if conditions in AML.

Before continuing with the next grammatical rules of AML, it is important to clarify how precedence is modeled in the DSL. Naturally, AML adheres to mathematical precedence rules such as multiplication having a higher precedence than addition. The question now arises how a grammar could represent these precedence rules. As outlined in section 2.4.2, through the definition of the grammar an AST can be built from the source code of the program which can then be interpreted. Since a syntax tree has to be interpreted from bottom to top, all operations further down in the tree have a higher precedence than operations further up in the tree. Figure 4 depicts a syntax tree for a simple calculation using operators of different precedence. It is clear that $2 \div 3$ would be evaluated before performing the addition with 1.

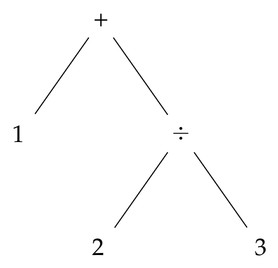


Figure 4: $1 + 2 \div 3$ as a syntax tree

The grammar of AML is modeled in a similar way which can be understood from the next grammatical rules in listing 38. A rule such as *logicEquivalence* does not necessarily mean, that an equivalence operation has to occur at this place. It does specify, however, that either an equivalence or an operation of higher precedence is taking place here.

```
assignment       → (IDENTIFIER "←" assignment) | composition | logicEquivalence
composition      → IDENTIFIER "∘" IDENTIFIER
logicEquivalence → logicImplication ("⇔" logicImplication)*
logicImplication → logicOr ("⇒" logicOr)*
logicOr          → logicXOr ("∨" logicXOr)*
logicXOr         → logicAnd ("⊕" logicAnd)*
logicAnd         → equality ("∧" equality)*
equality         → negation (("=" | "≠") negation)*
negation         → comparison | ("¬" comparison)
```

Listing 38: Assignments and logical expressions in AML.

With the part of the grammar of listing 38 it is possible to infer the following precedence rules: assignments have the lowest precedence. The right hand side of the assignment therefore has to be determined first before performing the assignment. For logical expressions, equivalence has less precedence than implication which in turn has less precedence than the logical or, up until negation having the highest precedence of all logical operations. For each logical operator, the structure is generally the same: either the next rule of higher precedence will be selected directly or an arbitrary amount of the next rule is combined using the current logical operator. Composition, i.e. *function composition*, is the only special case since it only affects functions and cannot be combined with other operations. It is therefore not part of the precedence hierarchy.

The next rule, *comparison*, differentiates between the operations for numbers/fractions and sets as well as quantifications. While AML allows sets and numbers to be part of a single logical expression, it does not allow calculations of numbers to be intertwined with set operations. Since the operations for sets follow a very similar pattern, only the rules (see listing 39) for numbers and fractions will be explained in further detail.

```
comparison       → quantification | numComparison | setComparison
numComparison    → term (("<" | ">" | "≤" | "≥") term)*
term             → factor (("+" | "-") factor)*
factor           → exponentiation (("·" | "÷" | "mod") exponentiation)*
exponentiation   → fraction ("^" fraction)*
fraction         → numUnary ("\\" numUnary)?
numUnary         → numNegation | factorial | floor | ceil | numPrimary
numNegation      → "-" numPrimary
factorial        → numPrimary "!"
floor            → "⌊" numPrimary "⌋"
ceil             → "⌈" numPrimary "⌉"
numPrimary       → call | number | IDENTIFIER | "(" logicEquivalence ")"
number           → NUMBER ("." NUMBER)?
call             → IDENTIFIER "(" arguments ")"
arguments        → logicEquivalence? | (logicEquivalence ",")+ logicEquivalence
```

Listing 39: Expressions involving numbers and fractions in AML.

The comparative operations have the lowest precedence for all numeric expressions since calculations should be evaluated before comparing their results. The precedence for the common operations such as addition, subtraction or exponentiation are the same as the mathematical rules and require no further explanation. Fractions have a lower precedence than unary expressions such as negation or factorial in order to allow developers to write fractions such as the following:

$$\frac{n!}{-5}$$

The double backward slash does not indicate that AML requires two slashes for a fraction, it only requires one. However, grammar processing tools, such as ANTLR, which will be utilized for the implementation of the DSL, require a backslash to be escaped. Therefore it is represented as such in listing 39.

The highest precedence for numeric expressions, specified by the *numPrimary* rule, are numbers themselves, variables (indicated by the *IDENTIFIER* terminal), function calls, or nested expressions which are grouped using parentheses. It is obviously necessary that these language elements have the highest precendence since the value of a number has to be determined first before any operation can take place.

By applying the same principle for set operations, it is possible to construct a mathematically correct AST from AML source code by using the provided grammar to accurately compute all mathematic operations which AML supports. For the sake of completeness, the entire grammar can be viewed in the appendix (see 123).

Due to the fundamental theory which was presented in the previous and the current chapter regarding the AML DSL and DSLs in general, the stage is set to see how this theory can be put into practice with GraalVM and how the DSL outlined in this chapter could be implemented and embedded in other applications.

4 Overview of GraalVM

Since its initial release, GraalVM has slowly gained traction in the software industry. New frameworks such as Quarkus[7], which builds on top of GraalVM, have been released. GraalVM is successfully employed by more and more companies to run JVM applications in production with Twitter, one of the biggest enterprises using GraalVM, reporting a decrease of 18% of machines required to run their Tweet service [12]. In addition to that, enterprises with custom languages, such as Nvidia or Goldman Sachs, are reimplementing custom programming languages for GraalVM [12].

When new developers start to dive in GraalVM, its name can be a bit confusing at first. According to Oliver Fischer [2], the name *GraalVM* was chosen for marketing reasons and is not quite suitable to characterise this new technology since it implies that GraalVM is a wholly new virtual machine. In truth, GraalVM is an assemblage of multiple runtimes, like the Java HotSpot VM or a JavaScript runtime, with a new compiler called *Graal*, and some additional tooling. The following chapter gives a high-level overview of the parts of GraalVM and sheds light upon their practical relevance. Before explaining the core components of GraalVM, however, it is necessary to answer what motivated the creation of GraalVM in the first place.

4.1 Motivation

As previously mentioned, the name GraalVM is actually derived from its central part: the Graal *just-in-time* (JIT) compiler. A JIT compiler is employed to compile source code or bytecode at runtime to machine code which can be executed much more quickly and therefore speeds up the execution of a program. This concept is nothing new when programming with a Java VM. Since its initial release in 1999, the HotSpot VM has incorporated two JIT compilers [2]: C1, a fast and lightly optimizing compiler which is suitable for desktop applications, and C2, a very aggressive optimizing compiler for server applications.

However, a problem with both of these JIT compilers is that the way people write programs has changed over the course of time. In the past, software developers were more careful about creating objects when not necessary since it was quite costly to instantiate new objects [2]. Today, however, APIs such as *streams* are preferred compared to low level constructs like *for loops* due to their conciseness, eventhough a lot of short lived objects are created when using them. Furthermore, Java is not the sole language which is executed on a Java VM anymore. Languages such as Scala, Clojure or Kotlin, which are widespread among software developers as well, are build for the JVM but utilise different constructs for which the current JIT compilers were not intended for [21]. Due to changing requirements like these regarding the JVM, it is necessary to adapt the HotSpot VM and its JIT compilers to the way developers write and execute their applications today. Because of the old age of the HotSpot VM and its JIT compilers, and the technical debt that has accumulated over the course of time, these changes are very hard to be put to practice with the existing JIT compilers [15]. In summary, a big motivation for the new Graal compiler was to replace the existing and somewhat outdated JIT compilers of HotSpot with a new compiler tailored to the requirements software developers are having today.

[7]`https://quarkus.io/`

A second reason behind Graal is the goal to rewrite more and more parts of the Java runtime in Java itself. The current HotSpot VM is build in C++, a highly complex language which is hard to master and to write correct and bug-free code with. Cliff Click, a developer of the C2 compiler goes even as far as saying he would never write a VM in C or C++ again [18]. Using Java allows the virtual machine developers to use a more secure language and be more productive at the same time. Using Java to write the Java runtime itself may sound paradox at first, but is possible as will be laid out later on.

Lastly, the initial creator of GraalVM, Thomas Wuerthinger, stated that another motivation behind GraalVM was to create a universal JIT compiler for all languages [21]. Programming languages are generally fairly similar, but for most popular languages which are in usage today new JIT compiler were created and have to be maintained individually. Therefore, another aim behind the creation of Graal was to build a compiler, obviously combined with further tooling, which is able to compile not only the traditional JVM languages but any programming language.

4.2 Features

As outlined previously, GraalVM consists of multiple components working together to enable the functionality which the technology provides. It can be downloaded from the official website of GraalVM[8] and installation instructions are provided by Oracle as well[9]. There are two editions of GraalVM, the community edition and the enterprise edition. The community edition is free for all purposes while the enterprise edition costs but comes with additional performance and security benefits. This thesis relies on the GraalVM community edition version 21.0.0 based on Java 11.

Out of the box, GraalVM ships with the following components [12]:

- GraalVM comes with three runtimes: the Java HotSpot VM, a JavaScript and Node.js runtime, and a LLVM runtime. Each runtime uses the Graal JIT compiler by default and is able to interact with code of a different language through the polyglot capabilities of GraalVM.
- To install, update, and remove additional components and runtimes such as a Ruby or Python runtime, GraalVM offers a component updating tool called *gu*.
- Libraries to work with GraalVM, such as polyglot APIs or the Truffle framework with which new languages can be implemented on top of GraalVM, are bundled with the technology as well.

The most prominent of these components and capabilities which are also relevant for this thesis will be discussed in further detail in the following sections.

4.2.1 Graal Compiler

The Graal JIT compiler is the heart of GraalVM. To understand how it works, it is necessary to take a step back and revise how code in a JVM is actually executed. Figure 5 illustrates how source code is executed in the HotSpot JVM.

[8]`https://www.graalvm.org/downloads/`
[9]`https://www.graalvm.org/docs/getting-started/#install-graalvm`

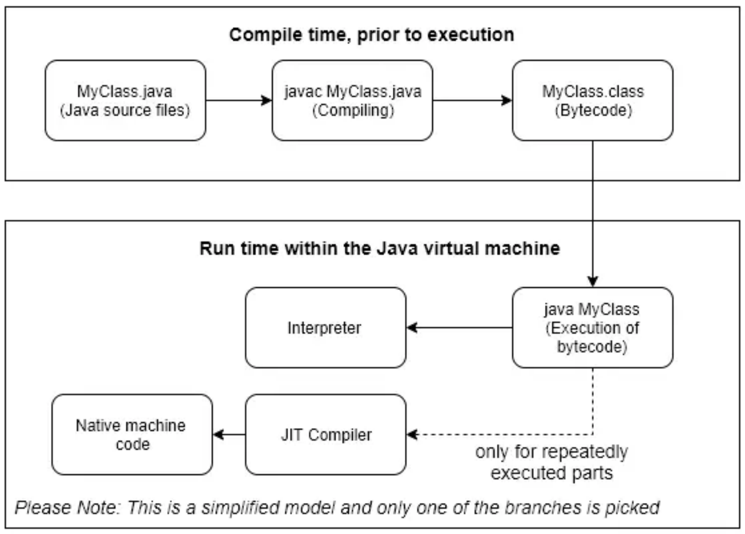

Compile time, prior to execution
MyClass.java (Java source files)
javac MyClass.java (Compiling)
MyClass.class (Bytecode)
Run time within the Java virtual machine
Interpreter
java MyClass (Execution of bytecode)
Native machine code
JIT Compiler
only for repeatedly executed parts
Please Note: This is a simplified model and only one of the branches is picked

Figure 5: Simplified workflow depicting how code is executed within the HotSpot JVM, taken from [14].

Whenever source files are compiled, the Java compiler produces bytecode which the JVM can understand and interpret. In case parts of the bytecode are executed very frequently, Thomas Wuerthinger states about 10,000 times [21], a JIT compiler kicks in and compiles this bytecode to native machine code to speed up the execution.

In the past, when using the HotSpot JVM these JIT compilers were either the C1 or the C2 compiler. With Java 9, however, a new *JDK Enhancement Proposal* (JEP) was incorporated: JEP 243[10] which consists of a *JVM compiler interface* (JVMCI) allowing compilers written in Java to be utilized as a dynamic compiler for the JVM. In other words, this new compiler interface allows developers to plug in custom built Java JIT-compilers to the JVM which is running their program [15].

The concrete interface which a dynamic compiler has to implement is generally quite simple. Listing 40 shows the interface taken from the current OpenJDK repository[11].

The only method a JIT compiler has to actually implement is *compileMethod* which must take a *CompilationRequest* containing a certain method, compile it, and return the result. Both the request and the result can be boiled down to basically byte arrays with some meta data, so in simple words, the *compileMethod* takes an array of bytes containing the bytecode of the method and transforms it into an array of bytes containing the machine code [15].

It is now apparent, that JVMCI paved the way to implement Graal as a new JIT compiler written in Java. Furthermore, it is clear how a high-level language like Java is able to perform a task for which, traditionally, low-level languages like C/C++ were employed. Java is as capable as C/C++ to take an array of bytes and transform it into a different array of bytes.

In general, the Graal compiler is implemented in a similar wary as the C2 compiler [15]. It takes the bytecode it should compile and first transforms it into a graph. This graph can be optimised and modified, and then transformed into machine code. As an example (taken

[10]`https://openjdk.java.net/jeps/243`
[11]`https://github.com/openjdk/jdk`

```
package jdk.vm.ci.runtime;

import jdk.vm.ci.code.CompilationRequest;
import jdk.vm.ci.code.CompilationRequestResult;

public interface JVMCICompiler {
    int INVOCATION_ENTRY_BCI = -1;

    CompilationRequestResult compileMethod(CompilationRequest request);

    default boolean isGCSupported(int gcIdentifier) {
        return true;
    }
}
```

Listing 40: The JVMCICompiler interface.

from [15]), the simple *average* method in listing 41 is transformed by Graal into the graph depicted in figure 6.

```
int average(int a, int b) {
  return (a + b) / 2;
}
```

Listing 41: A simple expression to showcase the graph Graal builds [15].

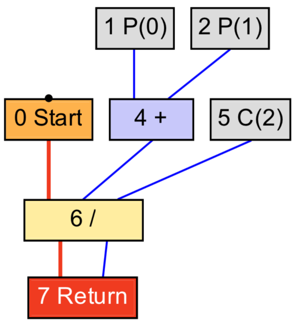


Figure 6: The graph representation of the average method, copied from [15].

The graph clearly represents the two parameters *a* (*P(0)*) and *b* (*P(1)*) being added together and divided by 2 (*C(2)*) which in the end is returned. Since Java is an object oriented language, it is perfectly suited as a language for Graal to operate on a graph datastructure because a graph can easily be represented as objects connected by references to each other.

Among the optimisations which Graal performs are [15]:

- **Canonicalisation:** Graal tries to simplify and rearrange nodes of the graph as much as possible. As an example, a double negation such as `--x` is simplified to just `x`.

- **Global value numbering:** Listing 42 displays a simple calculation where the addition of *a* and *b* is unnecessarily performed twice. During its global value numbering optimisation phase, Graal compares the nodes of its graph to find identical nodes. In case it finds some, it can reuse the result of one node for the other similar nodes, comparable to a cache. In this case, Graal would optimise the calculation code to only execute the addition of *a* and *b* once and reuse the result for the second addition.

```
int calculate(int a, int b) {
    return (a + b) * (a + b);
}
```

Listing 42: A simple calculation which Graal could optimise [15].

- **Lock coarsening:** In case Graal encounters multiple synchronisations on the same resource immediately after each other, it can combine these synchronisations to a single synchronisation. Listing 43 depicts a method where this optimisation could occur. While it is unlikely, that a software developer would actually write code where synchronisations on the same resources happen immediately after each other, such a constellation could occur after other optimisations by Graal have taken place.

```
void task() {
    synchronized (monitor) {
        // perform some work
    }
    synchronized (monitor) {
        // perform some more work
    }
}
```

Listing 43: Graal could use its lock coarsening optimising technique here [15].

There are multiple practical applications of this new compiler. The average JVM developer or enterprise could use Graal as its JIT compiler instead of using the traditional HotSpot compilers. While Graal does not outperform other JIT compilers like C2 for every single task or application, benchmarks (see [1] or [8]) have shown that Graal already has the upper hand in many cases. It can therefore be worthwhile to check whether using Graal reaps performance benefits for one's own JVM application as well.

Since Graal is basically just a library which provides the implementation of the *JVMCI-Compiler* interface, it is quite well decoupled from the rest of the JVM. Graal can therefore be employed for different purposes as well, not just JIT compiling JVM bytecode at runtime. Principally, *ahead-of-time compiling*, i.e. compiling the program to machine code before executing it as done with C/C++, is not inherently different than just-in-time compiling. That is why GraalVM is able to provide a technology called *native image* where JVM code can be ahead-of-time compiled to native applications using the Graal compiler. When executing an application built with native image, no JVM needs to be present, which firstly results in a much smaller overhead to run the applications and secondly in a dramatically faster startup time. GraalVM is therefore very suitable for Microservice based systems where services should be able to be quickly scaled up and down on small commodity machines.

4.2.2 Truffle Framework

This thesis, however, is concerned with a different application of the Graal compiler: the Truffle framework. Truffle is a technology bundled with GraalVM to build custom language implementations using Java which utilize the Graal compiler in a very unique way.

Generally, a language implementation in Truffle is built as an AST interpreter [5], similar to the one presented in section 2.4.2. Language developers have to structure their language as AST nodes and provide methods for evaluating each node of the AST. The methods of the AST nodes are then called recursively to evaluate the complete syntax tree. Building languages as AST interpreters has the advantage of being one of the easiest and straightforward ways to implement interpreters [23]. Yet choosing an AST interpreter for a language comes with a cost: performance. Traversing through the syntax tree and executing each node one at a time tends to be quite slow [11]. Hence, general-purpose language developers generally resort to better performing but more complex *bytecode interpreters* where source code is first translated to bytecode and then interpreted by a virtual machine. With the Truffle framework, however, the developers of GraalVM try to emphasise on the ease of developing languages as AST interpreters while making it performant at the same time.

A very important technique with which the Truffle framework tries to achieve good performance is through its concept of *specialization* and *tree rewriting* [23]. To illustrate this concept, listing 44 depicts a very simple function in a dynamic language such as JavaScript which performs an addition using two values and returns it.

```
function add(a, b) {
  return a + b;
}
```

Listing 44: A simple function written in a language like JavaScript [23].

For the purpose of this example, let the AST interpreter of this language represent additions using an AST node called *AddNode*. Since additions could be performed on a variety of different concrete data types such as integers, doubles, or strings, the AddNode would have to provide implementations for all these data types, in Truffle called specializations.

These specializations are used by the framework in an interesting way: the framework tries to optimize the AST of the program by constantly rewriting the nodes of the tree according to the specializations that where actually used. Figure 7 presents possible types of an AddNode for different data types. Before any addition takes place, the AddNode is in an uninitialized state. In case an integer addition should be performed by the node, it rewrites itself to an integer AddNode. In case no other data types have to be handled by the node, it becomes *stable* after a while, i.e. it is assumed that this node will only handle integer inputs

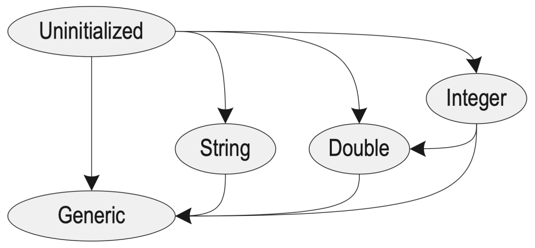


Figure 7: Transitions between different specializations of the AddNode, taken from [23].

in the future as well. However, if the add node encounters a double as input, it rewrites itself again to a more generic type. In this case a rewrite to a double AddNode would suffice to correctly perform an addition for both integers and doubles.

The tree rewriting technique of Truffle offers two benefits. First, since the framework tries to determine the most specific implementation for its AST nodes, it also determines how to evaluate each node in the fastest way possible since each node is optimized for the inputs it receives [23]. Secondly, through this profiling, it is possible to just-in-time compile the program using just the parts of the language interpreter which are necessary to execute the specialized AST [22], a process called *partial evaluation*. At this point, the Graal JIT compiler comes into play again. It is utilized by Truffle to generate efficient machine code of the specialized AST to improve the performance of the program execution.

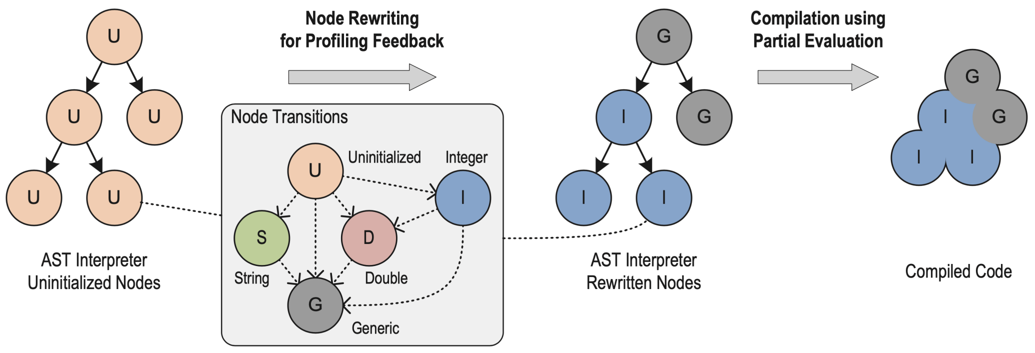


Figure 8: The whole process from the uninitialized AST to efficient compiled code, copied from [22].

The complete approach is summarised in figure 8 taken from [22]. In the beginning of the execution, all AST nodes are uninitialized since they were not executed yet. After executing the application and rewriting AST nodes for a while, Truffle declares the AST as being stable which is then compiled to efficient machine code. It is important to note that the assumption of the specialization of a node does not have to be true for ever. In case the machine code produced by the Graal compiler is unable to perform an operation due to an unexpected data type, the program execution is again handled by the AST interpreter which can then rewrite the node and a new JIT compilation can be produced.

Although the concept of specializations is the most characteristic mechanism of how the Truffle framework optimizes the language implementation, it is not the only one. The framework offers a wide variety of annotations and *compiler directives* to interact and help the

Graal compiler to produce machine code which performs as efficiently as possible. Examples include *Branch Profiling* where code branches, e.g. in if-else conditions, which are unlikely to be visited, will not be compiled by Graal until they are entered for the first time; *Explode Loop* which unrolls a loop with a constant number of iterations, i.e. instead of compiling the loop it concatenates the body of the loop as often as the number of interations; or *Compilation Finals* where language developers instruct Graal to treat a certain variable as a constant in its compilation eventhough the interpreter itself does not.

To conclude, what does Truffle offer for the development of DSLs? First, it offers one of the easiest ways to develop a language through the implementation of an AST interpreter using a well known and productive language: Java. Secondly, Truffle provides mainly three performance advantages: it uses Hotspot which is probably one of the most advanced virtual machines in existence, it offers JIT compilation by default, and it provides capabilities to enhance the performance of the JIT compiled code. Developers can therefore focus on their language implementation instead of spending time to speed up their virtual machine, garbage collection, or creating a JIT compiler. Although these benefits of Truffle may not directly be relevant for DSLs since performance is generally not the most important factor for them and JIT compilers would not be created for DSLs, it nontheless influences them indirectly. Due to its appealing advantages for general-purpose languages, language developers can and could be more and more interested in porting or developing their language using Truffle. As will be laid out in the next section, DSLs created with Truffle could then be used by all of these languages. The GraalVM team itself has already created a wide range of language implementations for different programming languages including Ruby, Python, JavaScript, and interestingly an LLVM bytecode interpreter called *Sulong*[12]. Since LLVM itself supports a wide range of programming languages like C/C++, Rust, or Fortran, these languages are supported by GraalVM as well. Listing 45 illustrates how a simple C program can be built using the LLVM *clang* compiler and run using GraalVM's LLVM interpreter.

```
$ cat hello.c
#include <stdio.h>

int main(void) {
    printf("Hello World\n");

    return 0;
}

# Get the location of the LLVM compiler and compile the source file
$ export LLVM_TOOLCHAIN=$(lli --print-toolchain-path)
$ $LLVM_TOOLCHAIN/bin/clang hello.c -o hello
$ $GRAAL_HOME/bin/lli hello
Hello World
```

Listing 45: Build and run a C program using GraalVM

An overview of how DSLs written with Truffle are supported by all Truffle languages is the topic of the next section and marks the last big feature of GraalVM related to the development of DSLs.

[12] https://github.com/oracle/graal/tree/master/sulong

4.2.3 Polyglot Applications

One of the biggest arguments in favor of GraalVM for DSLs is its ability to run polyglot applications, i.e. applications consisting of multiple different programming languages. Reasons to compose multiple languages include the following [5]:

- No programming language is suitable for all existing problems. On the contrary, most languages excel in a certain area of software development with the cost of having deficits in other areas. Although C or C++ distinguishes itself in low-level programming domains, it is not as fitting for web development as Ruby is. Therefore, interoperability between languages allows developers to chose the right language for the problem at hand.
- When rewriting software from one language to another, interoperability allows to perform this migration gradually. The project can be rewritten part by part, carefully making sure that the system as a whole still behaves correctly.
- Composing multiple languages improves the reuse of code. The overhead of porting libraries from one language to another could be circumvented by just using the library with its original programming language directly through interoperability mechanisms.

Generally, the topic of interoperability between programming languages is not a new topic. *Foreign function interface (FFI)* (see [7]) is a mechanism which allows one programming language to utilize functionalities from a different programming language. It was initially created in Python and since then has spread to various other programming languages, sometimes through other terms such as *Java Native Interface (JNI)* in Java. The downside of FFI, however, is the focus on the composition of a pair of languages instead of an arbitrary set of languages [5]. The Microsoft *Common Language Runtime (CLR)* is another example of a runtime which is able to run multiple languages that can interact and exchange objects with each other. The most prominent examples for languages running on the CLR are F#, C#, as well as Visual Basic. The approach to share objects between languages, however, is achieved by requiring every language to use a predefined representation of its data according to a fixed specification, the *Common Language Specification (CLS)*.

GraalVM offers more flexible possibilities regarding the interoperability of languages than existing approaches [5]:

- With GraalVM an arbitrary set of languages can be combined in a single application, not only a pair of languages.
- Programming languages can be of different paradigmas, i.e. it does not matter whether a programming language is object-oriented or not, statically typed or dynamically typed, manages its memory explicitly or automatic, or has safe or unsafe memory access. Languages can be composed regardless of their paradigma.
- Languages build on top of GraalVM can have their individual data representation and do not have to comply to a predefined object model.
- Existing languages are not affected and do not have to be modified when new languages are added to GraalVM but can be combined with the new languages by default.

- Language boundaries are largely invisible since foreign objects or functions can be accessed using the usual operators of the host language.

To illustrate these capabilities, a very small, exemplary polyglot application will be constructed. The purpose of this application is very simple: the program should print the first *n* fibonacci numbers, where *n* is given as a program argument. Although it would be senseless to write this simple program as a polyglot application, it is suitable for the purpose of this illustration. The program will use three languages, JavaScript for the implementation of the fibonacci algorithm, C to call the JavaScript fibonacci function as often as necessary to build the sequence, and Ruby to offer the functionality as a CLI application. The C program is shown in listing 46.

```c
#include <graalvm/llvm/polyglot.h>
#include <stdlib.h>

int *fibseq(int n) {
  int *ret = (int *) malloc(n * sizeof(int));
  const char *fib_func_src = "const fib = (n) => n <= 1 ? n : fib(n-1) + fib(n-2); fib;";
  int (*func)(int) = (int (*)(int)) polyglot_eval("js", fib_func_src);

  for (int i = 1; i <= n; i++)
    ret[i-1] = func(i);

  return polyglot_from_i32_array(ret, n);
}
```

Listing 46: Repeatedly call the JavaScript fibonacci function to build up a sequence.

The initial step of the *fibseq* function is the creation of the sequence as an array of integers which at the end of the function has to be transformed into a polyglot array to be processeable by other languages. The code of the JavaScript function, which creates a *fib* function and returns it, must first be evaluated and the result must be cast to a function pointer of correct type. Lastly, the JavaScript function is called as often as necessary and the array of fibonacci numbers is filled with its results.

The second part of this application, the Ruby source file, is depicted in listing 47. The code first verifies that an argument has been passed to the program. Afterwards it loads the compiled C code, executes the *fibseq* function with the program argument, transforms it into a Ruby array, and prints it in a well formatted manner to the command line output. To compile and execute the complete program, the commands of listing 48 must be executed.

From this example it is clearly visible that the creation of polyglot applications with GraalVM is easy and straightforward since the overhead of running code as well as transforming data representations from other languages to native objects is very small.

The approach GraalVM uses for its interoperability relies on the concept of language-agnostic *messages* [5]. If a language tries to perform an action with a foreign object, it maps this action to a certain message which gets resolved by the foreign language. Taking 47 as an example, the *fibseq* function call is mapped to an *execute* message. This message can then be resolved by the foreign language, in this case the C or more specifically the LLVM interpreter, which provides an implementation for the message and the specific object. There are a

```
if ARGV.size != 1
  puts 'usage: script.rb <MAX FIB NUMBER>'
  exit(-1)
end

clib = Polyglot.eval_file('llvm', 'fibseq')
puts clib.fibseq(ARGV.first.to_i).to_a.join(', ')
```

Listing 47: Retrieve the size of the fibonacci number sequence and call the C function.

```
# Get the location of the GraalVM LLVM tools
$ export LLVM_TOOLCHAIN=$(lli --print-toolchain-path)

# Compile the C code first
$ $LLVM_TOOLCHAIN/clang fibseq.c -lgraalvm-llvm -c -o fibseq

# Execute the Ruby script
$ $GRAAL_HOME/bin/ruby --jvm --polyglot app.rb 10
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Listing 48: Execute the fibonacci sequence polyglot application.

variety of different messages for reading or writing members and array elements, executing methods or functions, and checking datatypes of the foreign objects. Language developers therefore only have to provide implementations for the messages which each data object should support to allow foreign languages to utilize the objects of the developed language.

Although the arguments and illustrations of this section were based on general-purpose languages, the same benefits hold for the realm of DSLs. Not only general-purpose languages can be composed in an application, DSLs can be embedded in the same way. After implementing a DSL, embedding it is not limited to only the language with which it was implemented, but is possible for all existing and future languages which GraalVM supports. Since general-purpose languages can easily share data objects between each other with GraalVM, it is also effortless to share data or objects which were produced by the DSL with a host language and use it natively therein. GraalVM hence blurs the line between internal and external DSLs in a certain way. Although DSLs built with Truffle are certainly external DSLs since they are not limited by the syntax of any host language, they can be embedded in other languages in a relatively straight forward manner similar to an internal DSL.

All in all, GraalVM poses a new and compelling platform for software developers to create DSLs upon in order to leverage and benefit from the creation of their DSLs even more.

5 Implementation of AML

While the intent of the previous chapter was to introduce GraalVM and provide the theoretical benefits of it as a platform to build DSLs upon, this chapter addresses the question of how DSLs can be practically built with the Truffle framework by using AML as an example. Similar as outlined in section 2.4.2 with the task scheduling DSL, the evaluation of an AML program can be divided into three steps: lexing, parsing, and interpreting. However, as will be shown throughout this chapter, much of the manual labour, that was necessary for the other external DSL, will be handled by libraries and the Truffle framework to make the life of a language developer easier.

This chapter starts by highlighting the fundamentals of a Truffle language and an approach to parse and lex language source code using ANTLR. Afterwards, the implementation of the different parts of AML will be explained feature by feature from its datatypes to its functions. The final section is dedicated to show how interoperability can be added to a Truffle language and how it can be employed in actual applications. Although many parts of the AML source code will be shown throughout the following sections, the complete code is also available on GitHub[13].

5.1 Fundamentals

To start out with the development of a Truffle language, only one library is necessary: the Truffle API. Listing 49 depicts how the Truffle API library can be added to a project using the *Maven* build tool, although any other build tool is possible as well. There are no requirements regarding the layout of the project except for the packaging of the language to be JAR. For AML, the standard project layout of Maven was employed.

```
<dependencies>
    <dependency>
    <groupId>org.graalvm.truffle</groupId>
    <artifactId>truffle-api</artifactId>
    <version>${graalvm.version}</version>
    </dependency>
</dependencies>
```

Listing 49: Add the Truffle API library to a build tool like Maven.

While it is possible to develop a Truffle language to be an executable JAR, it is preferable to treat the language as a non-executable library. With this approach, the language can be employed more flexibly and can be added as a component to GraalVM as will be outlined in a later section. Truffle languages therefore generally do not comprise a static *main* method. The actual entry point of a Truffle language is its *language class* which the framework uses to extract information about the language or to execute source code. A language class must extend the *TruffleLanguage* class and must be annotated by the *TruffleLanguage.Registration* annotation. Through this annotation different options can be set like the identifier of the language which

[13] https://github.com/bakku/aml

foreign languages must specify when they want to run code of this language, the mime type of the language, or an implementation of a file type detector which GraalVM can utilise to retrieve the language it should use for a certain source file. For a list of all possible options of this annotation, its documentation can be consulted[14]. Listing 50 presents the language class of AML without going into the details of its *parse* method yet.

```
@TruffleLanguage.Registration(id = AMLLanguage.ID, name = AMLLanguage.NAME,
    defaultMimeType = AMLLanguage.MIME_TYPE, ...)
public class AMLLanguage extends TruffleLanguage<AMLContext> {
    public static final String ID = "aml";
    public static final String NAME = "AML";
    public static final String MIME_TYPE = "application/x-aml";

    @Override
    protected AMLContext createContext(Env env) {
        return new AMLContext();
    }

    @Override
    protected CallTarget parse(ParsingRequest request) {
        // ...
    }
}
```

Listing 50: The language class of AML.

Although the *TruffleLanguage* parent class offers a variety of methods that can be overridden, in case of AML only two methods had to be implemented: a *createContext* method which every Truffle language has to override and a *parse* method that can be optionally overridden (although it is generally necessary).

The *parse* method is the entrypoint for new source code that should be prepared for execution. The source code itself is contained in the *ParsingRequest* which is passed to the method. Inside this method, a language developer has to implement the lexing and parsing of the source code and its transformation into an AST of Truffle nodes. This AST is then returned as a *call target* which is basically an object that contains the AST and can be called by Truffle to execute the AST when necessary. The creation of this call target will be laid out later in this chapter when the approach for lexing and parsing is discussed.

The *createContext* method must create and return an instance of the class which was specified as the generic type of the *TruffleLanguage* super class. It introduces another concept of a Truffle language - the *context*. The context of a language can be thought of as a global object to store data such as global variables or to provide the standard library of a language. To offer an example for such a global object, the context of AML is presented in listing 51.

The context of AML consists of two attributes, a *FrameDescriptor* and a *MaterializedFrame*. While a frame is an object that holds values of variables, which are more abstractly called *frame slots* in Truffle, a *FrameDescriptor* describes the available slots in a frame. In case of this context, the frame and frame descriptor act as a storage for global variables and functions.

[14] https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/TruffleLanguage.Registration.html

```java
public class AMLContext {
    private final FrameDescriptor globalFrameDescriptor;
    private final MaterializedFrame globalFrame;

    public AMLContext() {
        this.globalFrameDescriptor = new FrameDescriptor();
        this.globalFrame = Truffle.getRuntime()
            .createVirtualFrame(null, this.globalFrameDescriptor)
            .materialize();
        addBuiltIns();
    }

    private void addBuiltIns() {
        this.globalFrame.setObject(
            this.globalFrameDescriptor.addFrameSlot("π"),
            AMLNumber.PI
        );

        this.globalFrame.setObject(
            this.globalFrameDescriptor.addFrameSlot("τ"),
            AMLBoolean.of(true)
        );

        // ...
    }

    public MaterializedFrame getGlobalFrame() {
        return globalFrame;
    }

    public FrameDescriptor getGlobalFrameDescriptor() {
        return globalFrameDescriptor;
    }
}
```

Listing 51: The context of AML which holds global variables and its standard library.

The usage of both the frame and its descriptor can be seen in the *addBuiltIns* method. Calling the *addFrameSlot* method of the frame descriptor with an identifier for the slot (in this case π) creates a new frame slot and returns it. This frame slot is then assigned to the mathematical value of π inside the frame so it can be referenced from everywhere.

Another essential concept in Truffle is the AST root node which represents the first node in the AST and therefore provides the entry point for the evaluation of the complete AST. To define a root node, the framework provides an abstract class called *RootNode* that must be extended. Listing 52 provides the implementation of the root node in AML.

The constructor of a root node does not have to fulfil any requirements. In case of AML, the constructor of the root node receives a single AST node and stores it as an attribute. This attribute is annotated with *Child* to signify that it represents a single child of the root node in the resulting AST. Multiple child nodes could be denoted with the *Children* annotation. These annotations are employed in later listings as well and are necessary for Truffle to differentiate between attributes which are actual AST elements and attributes holding plain data.

The only method a root node has to implement is the *execute* method. This method

```
public class AMLRootNode extends RootNode {
    @Child
    private AMLBaseNode root;

    public AMLRootNode(AMLBaseNode root) {
        super(null);
        this.root = root;
    }

    @Override
    public Object execute(VirtualFrame frame) {
        try {
            return root.executeGeneric(frame);
        } catch (AMLRuntimeException ex) {
            return new AMLError(ex.getMessage());
        }
    }
}
```

Listing 52: The root node provides the entry point for the evaluation of an AST.

receives a frame, in this case a *VirtualFrame*, as a parameter which is provided by Truffle when the root node's *execute* method is called and therefore the AST evaluated.

The distinction between the *MaterializedFrame*, that was utilised in the language context, and a *VirtualFrame* is the following: a *VirtualFrame* is ephemeral and is created every time a new AST is evaluated and destroyed after the evaluation completes. It is not allowed to store a reference to a *VirtualFrame* as an attribute of an object. This restriction is necessary due to optimisations which the Graal compiler carries out on a *VirtualFrame*. On the other hand, a *MaterializedFrame* is not ephemeral so references to such a frame can be stored as an attribute. Due to its missing optimisations, however, *MaterializedFrames* perform worse than *VirtualFrames*. Therefore, while a *MaterializedFrame* can be employed for global variables as in the context of AML, the *VirtualFrame* should only be used for variables which exist solely during the evaluation of a single AST. Thus, the *VirtualFrame* has to be passed from node to node to share the access to the frame throughout the complete AST.

Through the implementation of the *execute* method in the root node, it is also possible to understand how error handling is implemented in AML. In case an error occurs in any node of the AST, the node can throw an *AMLRuntimeException*. This exception is then propagated through the AST up to the root node where it is catched and returned as an instance of an error type.

To understand the remaining part of the *execute* method, namely the execution of the child node, the class hierarchy of the standard AST nodes must be discussed as the last topic of this section. While the *RootNode* class of Truffle should only be used for the root node of an AST, Truffle provides another abstract class for all the other nodes of the AST called *Node*. Instead of using this class directly, it is recommended to create a custom abstract class, that extends *Node*, which acts as a superclass for all other concrete nodes. This is necessary since the *Node* class itself does not provide an *execute* method for concrete nodes to implement. Truffle more or less delegates the complete responsibility of structuring the AST nodes to the language developer. AML's custom abstract class *AMLBaseNode*, that acts as a parent class for all concrete nodes, is presented in listing 53.

```
public abstract class AMLBaseNode extends Node {
    public abstract Object executeGeneric(VirtualFrame frame);
}
```

Listing 53: The custom abstract node class which is the superclass of all AML AST nodes.

According to this listing, the method which every node in AML has to implement is the *executeGeneric* method. This method receives the frame given by the parent node, evaluates the node, and returns the result as a generic object. Through this abstract class, the flexibility of building an AST is improved due to two benefits. First, since every node is of type *AMLBaseNode*, it is not necessary for any node to know the specific type of its children nodes. A parent node only has to know how to evaluate its child nodes, in this case by calling the *executeGeneric* method of each child node. This design therefore allows a flexible nesting and execution of ASTs by representing an AST as a decoupled tree of *AMLBaseNode* objects. This flexibility is already visible in the root node of listing 52: since it receives an instance of *AMLBaseNode* as its only child, it is not bound to any concrete node type. It therefore does not matter to the root node whether the AST only consists of an addition of two numbers or of a large algorithm.

5.2 Lexing and Parsing

For the external timer scheduling DSL in section 2.4.2, the lexing and parsing was performed using custom written classes. Although this might be feasible for small languages or implementations which require a very fine grain control, the work load naturally increases the bigger the language gets. However, since the general approach for lexing and parsing languages is very generic, a multitude of libraries and tools exist to automate the generation of lexers and parsers.

Many existing Truffle languages use a popular library available for Java, called ANTLR, to generate their lexers and parsers. ANTLR is a technology which takes a grammar as input, similar to the one of AML in listing 123 in the appendix, and generates classes to lex and parse source code to produce an AST, as well as predefined interfaces and base classes to walk through the syntax tree. To set it up, the ANTLR runtime library has to be added to the list of dependencies as in listing 54. Furthermore, to not have to generate the ANTLR classes manually whenever the grammar changes, it is possible to automatically generate these classes during the build of the project. For Maven, ANTLR offers a plugin to perform this task which can be set up as shown in listing 55. In the configuration of the plugin, it is possible to define which interfaces and base classes should be generated to aid walking through the syntax tree that ANTLR produces. For this functionality, ANTLR offers two patterns, the *visitor* pattern and the *listener* pattern. In case of AML, the visitor interface was utilized as will be shown later. Through this Maven plugin, the ANTLR classes can either be generated directly by executing `mvn antlr4:antlr4`, or they will be generated when running common Maven lifecycles such as `mvn package` or `mvn test`.

```
<dependency>
  <groupId>org.antlr</groupId>
  <artifactId>antlr4-runtime</artifactId>
  <version>4.7.2</version>
</dependency>
```

Listing 54: The ANTLR library has to be added to the list of dependencies as shown here with Maven.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.antlr</groupId>
      <artifactId>antlr4-maven-plugin</artifactId>
      <version>4.7.2</version>
      <configuration>
        <listener>false</listener>
        <visitor>true</visitor>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>antlr4</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 55: Instead of generating the ANTLR classes manually they can be generated during the build with the help of a plugin.

Regarding the grammar, not a lot of changes have to be carried out to the previous definition of AML's grammar to make it processable for ANTLR. First, it is necessary to place the grammar file at a correct location for ANTLR to find it, if adhering to the recommended approach of automatically generating the classes. For the aforementioned Maven plugin, ANTLR expects the grammar at `src/main/antlr4/org/example/language/<NAME>.g4`. For other build tools or plugins, it might be necessary to place the grammar file at a different location. Secondly, the name of the grammar has to be defined at the first line of the ANTLR grammar file. Thirdly, instead of using arrows (→) to separate the name of the rule, ANTLR uses colons and requires each rule to be concluded by a semicolon. Finally, instead of using string literals for the terminals of the grammar, it is recommended to give names for each terminal in order to use this name later on as a constant. The benefit of doing this will become more clear later in this chapter. To provide an example for these necessary changes, listing 56 provides an excerpt of the AML grammar from the appendix for which the correct ANTLR grammar will be shown in listing 57.

On generation, ANTLR produces a lexer, a parser, and a visitor (and/or listener in case it

```
// ...
numComparison    → term (("<" | ">" | "≤" | "≥") term)∗
term             → factor (("+" | "-") factor)∗
factor           → exponentiation (("·" | "÷" | "mod") exponentiation)∗
// ...
```

Listing 56: Excerpt of the AML grammar which will be transformed to ANTLR's flavour of writing grammars.

```
// src/main/antlr4/dev/bakku/aml/language/AML.g4
grammar AML;
// ...
numComparison : term ((LT | GT | LTE | GTE) term)∗ ;
term          : factor ((PLUS | MINUS) factor)∗ ;
factor        : exponentiation ((MULTIPLY | DIVIDE | MOD) exponentiation)∗ ;
// ...
LT            : '<' ;
LTE           : '≤' ;
GT            : '>' ;
GTE           : '≥' ;
PLUS          : '+' ;
MINUS         : '-' ;
MULTIPLY      : '·' ;
DIVIDE        : '÷' ;
MOD           : 'mod' ;
```

Listing 57: Part of the AML grammar written in ANTLR's flavour of writing grammars.

was enabled). These are sufficient to parse and prepare source files in order to execute them with Truffle. In order to show the application of these classes it is necessary to take a step back to the previous section of this chapter and to provide an implementation for the *parse* method of the language class in listing 58. To recap, the *parse* method of the language class acts as the entry point for new code to be parsed and prepared for execution.

The first method which *parse* calls is a static method of a utility class, called *parseTree*, which performs the complete lexing, parsing and transformation of AML source code into a Truffle AST. The method returns an instance of the *AMLBaseNode* class that was introduced in the previous section. This instance is stored in the *node* variable and wrapped in a root node, with which a call target can be created and returned. This call target can then be called by Truffle to execute the complete Truffle AST. In case any exceptions occur during this process, a call target will be returned which only contains one AST node, an instance of *AMLReturnErrorNode* which, as the name implies, just returns an error. How nodes are defined and created is the topic of a later section in this chapter.

The definition of the *AMLSyntaxTreeParser* utility class, which does most of the work in this process, is shown in listing 59. Its *parseTree* method requires two arguments, the context of AML to prepare frame slots for global variables, and the source code of the *ParsingRequest* that was passed to the former *parse* method by the Truffle framework. It first creates an ANTLR error listener which is basically an implementation of an interface with a set of hook

```
@Override
protected CallTarget parse(ParsingRequest request) {
    try {
        var node = AMLSyntaxTreeParser.parseTree(
            getCurrentContext(AMLLanguage.class),
            request.getSource()
        );
        return Truffle.getRuntime().createCallTarget(
            new AMLRootNode(node)
        );
    } catch (AMLParserException ex) {
        return Truffle.getRuntime().createCallTarget(
            new AMLRootNode(new AMLReturnErrorNode(new AMLError(ex.getMessage())))
        );
    }
}
```

Listing 58: Implementation of the parse method of the AML language class.

```
public class AMLSyntaxTreeParser {
    public static AMLBaseNode parseTree(AMLContext ctx, Source source) {
        var listener = new AMLErrorListener();
        var code = source.getCharacters().toString();
        var lexer = new AMLLexer(CharStreams.fromString(code));
        lexer.addErrorListener(listener);
        var stream = new CommonTokenStream(lexer);
        var parser = new AMLParser(stream);
        parser.addErrorListener(listener);
        var tree = parser.program();

        if (listener.hasErrorOccurred()) {
            throw new AMLParserException("parsing completed with errors");
        }

        var visitor = new AMLAntlrVisitor(ctx);
        return visitor.visitProgram(tree);
    }
}
```

Listing 59: The *AMLSyntaxTreeParser* combines the complete process of lexing, parsing and transforming AML source code to a Truffle AST.

methods. Whenever ANTLR encounters an error during lexing and parsing, it calls the hook methods for all listeners which were added to the lexer or parser. Through this listener, it is possible to check, at the end of the *parseTree* method, whether errors occurred during lexing or parsing, to then throw an exception which is catched by the *parse* method as shown in the previous listing. The source code that was passed as an argument is converted to an ANTLR *CharStream* to create an instance of *AMLLexer* with it. The *AMLLexer* is the first of the classes that ANTLR automatically generates and is directly passed as an ANTLR *CommonTokenStream* to a new instance of *AMLParser*, the second class that ANTLR generates. This parser has the capability to produce an AST, also predefined by ANTLR, by calling a method with the same name as the entrypoint of the grammar, in AML's case the *program* rule. However, since the AST which ANTLR produces is specific to ANTLR and in no way related to a Truffle AST, it has to be transformed into one. This is the task of the *AMLAntlrVisitor* which is a subclass of the previously mentioned visitor class that ANTLR provides.

The *AMLAntlrVisitor* is therefore the last missing piece to understand the complete evaluation process of AML source code. Although multiple examples of its working will be explained in the next sections which cover the individual features of AML, its fundamental manner of operation will be illustrated here. Through its visitor, ANTLR provides an easy way to walk through a complete ANTLR AST by defining *visit* methods for each grammatical rule that can be recursively called. A language developer can extend this visitor and implement its *visit* methods to perform certain actions according to the structure of the ANTLR AST. In case of AML, a visitor was implementated to transform an ANTLR AST into a Truffle AST. To exemplify the functioning of this visitor, listing 60 presents its class including the first of these visitor methods: *visitProgram* and *visitExpression*.

The *AMLAntlrVisitor* extends the *AMLBaseVisitor* of ANTLR with the specification of *AMLBaseNode* as its generic type. This generic type signifies that the return type of every *visit* method must be *AMLBaseNode*. The choice of *AMLBaseNode* as generic type is obvious for AML due to the fact that an AML AST is basically just a tree of *AMLBaseNode* objects. Therefore, to transform an ANTLR tree to a tree of *AMLBaseNode* objects, the *visit* methods which transform the ANTLR tree must naturally return this type.

As argument, every *visit* method receives a context object which holds information about the current position in the ANTLR AST and should not be mistaken with the previously mentioned Truffle context. Every grammar rule has a specific context class which is generated by ANTLR and which has methods to retrieve the next children of the current node of the AST according to the grammatical rules of the language. As an example, for both rules in listing 61, ANTLR generates a corresponding context class, in this case *LogicAndContext* and *QuantificationContext*. Both classes define generic methods and attributes such as the *children* attribute that holds a list of all its containing child nodes (both terminals and nonterminals), or the *getChildCount* method to retrieve the amount of children of the current node. However, both classes also provide methods which are specific for the type of context. While the *LogicAndContext* class will hold a *equality* method to return a list of all *EqualityContext* children of its *logicAnd* node, the *QuantificationContext* class will offer a *universal*, *existential*, and a *uniqueness* method which all return a single *UniversalContext*, *ExistentialContext*, or *UniquenessContext* object. By using the methods of the individual contexts in the *visit* methods, it is therefore possible to determine how the ANTLR AST is structured and how to continue the transformation to the AML AST.

In case of the *visitProgram* method of listing 60, the *program* rule of AML has one or more *function* or *expression* rules which can occur interchangeably. The *visitProgram* method therefore iterates over all children of the current node, filters out terminals such as newlines, and then calls for each child either the *visitFunction* method if the child is a function, or the *visitExpression* rule if it is an expression. Since an *expression* is either an *if condition* or an *assignment*, the *visitExpression* method checks whether the expression either contains an *if condition* to then recursively call the *visitIfCond* method, or it executes the *visitAssigment* method. This approach of recursively visiting the children nodes continues throughout the *AMLAntlrVisitor* where each *visit* method translates the ANTLR AST, node by node,

```java
public class AMLAntlrVisitor extends AMLBaseVisitor<AMLBaseNode> {
    private final AMLContext context;

    public AMLAntlrVisitor(AMLContext context) {
        this.context = context;
    }

    @Override
    public AMLBaseNode visitProgram(AMLParser.ProgramContext ctx) {
        var nodes = ctx.children
            .stream()
            // filter all newlines
            .filter(c -> !(c instanceof TerminalNode))
            .map(c -> {
                if (c instanceof AMLParser.FunctionContext) {
                    return this.visitFunction((AMLParser.FunctionContext) c);
                } else {
                    return this.visitExpression((AMLParser.ExpressionContext) c);
                }
            }).toArray(AMLBaseNode[]::new);

        return new AMLProgramNode(nodes);
    }

    @Override
    public AMLBaseNode visitExpression(AMLParser.ExpressionContext ctx) {
        if (ctx.ifcond() != null) {
            return this.visitIfcond(ctx.ifcond());
        }

        return this.visitAssignment(ctx.assignment());
    }

    // ...
}
```

Listing 60: The first method of the visitor: *visitProgram*.

```
logicAnd       → equality ("∧" equality)*
quantification → universal | existential | uniqueness
```

Listing 61: Two rules of the AML grammar.

to a hierarchical structure of nested *AMLBaseNode* objects. Since every *visitFunction* and *visitExpression* call in the *visitProgram* method produces its own hierachy of *AMLBaseNode* objects, each hierarchy is in the end stored in an *AMLBaseNode* array and wrapped by an *AMLProgramNode*. This node is again an *AMLBaseNode* and takes an array of child nodes which it sequentially executes.

Through the process that was outlined in this section, the evaluation of source code in the Truffle framework should principally be clear. First, the framework provides source code through the *parse* method of the language class. This source code is then passed through the lexer and parser which were generated by ANTLR to build an ANTLR AST. Afterwards, the ANTLR AST is transformed to a Truffle AST by walking through the tree and transforming the ANTLR nodes to Truffle nodes. In the end the Truffle nodes are wrapped by a root node and a call target is provided to Truffle that allows the framework to execute the AST at will.

5.3 Datatypes

The last topic, which has to be illustrated before covering the individual Truffle nodes, are AML's datatypes. Since data is extensively created, read, and modified in the nodes, the implementation of datatypes is important for the comprehension of the remaining sections of this chapter.

When building languages with Truffle, a lot of control regarding performance is outsourced and not part of the responsibility of the language developer anymore. Examples include the virtual of machine of the language, which in the case of a Truffle language is HotSpot, or the JIT compiler, for Truffle languages the Graal compiler. These technologies are provided by Oracle and are not in the jurisdiction of the language developer except through tweaks or functionalities which the individual technologies provide. However, the structure and implementation of datatypes is the sole responsibility of language developers and can have measurable impact on performance.

This connection between performance and the internal representation of datatypes can be easily demonstrated using the example of integers. Java offers different possibilities to store and work with integers: primitives such as *int* or *long* and objects such as *Integer*, *Long*, or *BigInteger*. Since every Truffle language is implemented in Java, language developers therefore have to decide which of these options to utilise for their implementation. Each possibility comes with both upsides and downsides. Primitives are very performant, though they do not offer any additional functionalities and could potentially overflow. Wrapper classes such as *Integer* or *Long* provide mediocre performance, can overflow as well, but offer additional capabilities such as conversions to other datatypes or the possibility to be stored in data structures like lists. Lastly, *BigInteger* is the worst choice regarding performance but is able to offer arbitrary length of numbers so programmers do not have to think about the possibility of an integer overflow. To make the difference in performance concrete, listing 62 presents the results of a small benchmark that was conducted for the sake of this example. The benchmark executes the addition of ten million integers for the types *int*, *Integer*, and *BigInteger*, and prints the running time for every type. The results indicate that the usage of *int* performs more than 30 times better than the usage of *BigInteger*.

```
$ java IntTest
BigInteger Test: 248 ms
Integer Class Test: 24 ms
Integer Primitive Test: 8 ms
```

Listing 62: Small benchmark that was conducted for this section to show the difference between *int*, *Integer*, and *BigInteger*.

A performance optimised implementation of integers therefore has to utilise the most basic datatype of Java as long as possible. For instance, initially all numbers could be stored as *int*. In case a number exceeds the limitations of the *int* datatype, it could be cast to a *long*, up until a cast to a *BigInteger* has to occur.

Approaches such as these, although possible to implement with Truffle and definitely necessary for performance oriented languages, are generally overkill for DSLs. In case of AML, the language implementation is focused on simplicity and keeping the complexity to a minimum while offering flexibility regarding its datatypes, for example by offering arbitrarily large numbers. The outlined tradeoff is the performance penalty AML has to take due to this decision. However, since performance is not a principal concern of the language, this tradeoff was readily made.

All of AML's datatypes implement an interface called *AMLObject* which is presented in listing 63.

```java
public interface AMLObject {
  // methods which all datatypes must implement
}
```

Listing 63: The *AMLObject* interface which each datatype has to implement.

```java
public class AMLNumber implements AMLObject {
    private static final int PRECISION = 100;
    public static final AMLNumber PI = AMLNumber.of(
        BigDecimalMath.pi(new MathContext(PRECISION))
    );
    private final BigDecimal number;

    private AMLNumber(BigDecimal number) { this.number = number; }

    public static AMLNumber of(String number) {
        return new AMLNumber(new BigDecimal(number));
    }

    // more AMLNumber.of methods for ints and BigDecimals ...

    public AMLNumber add(AMLNumber other) {
        return new AMLNumber(this.number.add(other.number));
    }

    public AMLNumber divide(AMLNumber other) {
        if (other.number.equals(BigDecimal.ZERO))
            throw new AMLRuntimeException("Division by zero");

        return new AMLNumber(this.number.divide(other.number, new MathContext(PRECISION)));
    }

    // more operations such as multiply, modulo, factorial, ceil, isSmaller, ...
}
```

Listing 64: Numbers in AML are implemented as the *AMLNumber* class.

Using such a parent interface for all types offers two advantages: first, it is possible to make sure that every datatype defines all methods included in the interface if such a necessity occurs. Secondly, since all datatypes are a subtype of the interface, it is possible to use this interface at places in the language implementation where any datatype of the language is allowed. To provide a concrete example, AML offers sets which contain other datatypes of AML. Since the sets of AML should not be restricted to a single datatype, the set itself is defined to hold *AMLObject* instances and can therefore flexibly contain any datatype.

As specified in the introduction of AML, the language includes four datatypes: booleans, numbers, fractions, and sets. The structure of each datatype is actually fairly similar. To take an example, the number type of AML is implemented as shown in listing 64.

In simple words, every datatype in AML can be explained as being a wrapper of a Java type with a set of additional utility methods. In case of numbers, the internal Java type is *BigDecimal*. Because AML does not differentiate between integers and floating point numbers, the usage of a decimal number type for all numbers of AML is the most adequate choice. Furthermore, since *BigDecimal* offers arbitrarily large numbers, AML numbers do too. However, when considering that *BigDecimals* can not only be arbitrarily large but also arbitrarily precise, their precision needs to be limited in some way. Otherwise a calculation such as $1 \div 3 = 0.\overline{3333}$ could result in an infinitely large floating point number. For this reason, AML limits the precision of some operations to 100 digits, as shown in the *divide* method. Lastly, the immutability of AML is visible in the implementation of the number datatype as well. All operations such as *add* or *divide* do not mutate the current instance but return a new *AMLNumber* as their result.

To provide another example for the implementation of a datatype, listing 65 presents the set datatype of AML called *AMLSet*. As with the number type, the set implementation acts as a wrapper for a Java type, here a *LinkedHashSet*. Through the usage of *LinkedHashSet* instead of a different set implementation such as *HashSet*, AML sets are very easy to work with due to the fact that a *LinkedHasSet* retains the order of insertion. When considering the creation of a set that contains all numbers from one to a hundred, receiving $\{1, 2, ..., 100\}$ as a result is much easier to verify for users of the language compared to $\{73, 23, ..., 83\}$, i.e. a seemingly random sequence of the numbers of the set. The remaining structure of the set implementation is very similar to the number type as well. Again, all of the operations of the set type do not mutate the set but return a fresh instance and therefore represent the respective mathematical operation more correctly.

Since the implementation of the other two datatypes do not provide any further insight, no additional code listings will be provided for fractions and booleans here. In case of interest, the definition of both datatypes is available on GitHub in the code repository of AML[15, 16].

[15]https://github.com/bakku/aml/blob/master/language/src/main/java/dev/bakku/aml/language/runtime/types/AMLBoolean.java

[16]https://github.com/bakku/aml/blob/master/language/src/main/java/dev/bakku/aml/language/runtime/types/AMLFraction.java

```java
public class AMLSet implements AMLObject {
    private final LinkedHashSet<AMLObject> set;

    private AMLSet(LinkedHashSet<AMLObject> set) {
        this.set = set;
    }

    public static AMLSet of(AMLObject... values) {
        var s = new LinkedHashSet<>(Arrays.asList(values));
        return new AMLSet(s);
    }

    public AMLSet intersect(AMLSet other) {
        var result = new LinkedHashSet<>(other.set);
        result.retainAll(this.set);
        return new AMLSet(result);
    }

    public AMLSet union(AMLSet other) {
        var result = new LinkedHashSet<>(other.set);
        result.addAll(this.set);
        return new AMLSet(result);
    }

    public Stream<AMLObject> stream() {
        return this.set.stream();
    }

    // more operations such as contains, cardinality, isSubset, ...
}
```

Listing 65: Sets in AML are implemented as the *AMLSet* class.

5.4 Pure nodes

After having covered all the foundational concepts of AML, the heart of a Truffle language, i.e. the creation and inner workings of the Truffle AST nodes, can be discussed. To start off, this section covers the type of AST nodes of AML that perform mathematical operations such as calculating the intersection of sets or adding numbers, as well as nodes related to the creation of datatypes. Since these nodes do not perform any side effects but just take inputs and produce an output, they are called *pure nodes* in this thesis analogous to *pure functions* in functional programming.

In section 5.1 of this chapter, it was specified that every node in AML is a subclass of *AMLBaseNode* and has to provide an *executeGeneric* method to evaluate itself. Just by putting these two pieces of information into practice, it is already possible to implement basic Truffle nodes. As a first example for a simple Truffle node, the implementation of the set literal node to create new sets will be outlined in the following paragraphs. To be able to follow the explanation of the implementation more easily, listing 66 presents the grammar rule of the set literal.

According to the rule, the set literal should have zero or more child nodes where the result of each child node represents one entry in the set. Listing 67 presents how the node

```
setLiteral → "{" ((logicEquivalence ",")* logicEquivalence)? "}"
```

Listing 66: Grammar rule of the set literal in AML.

providing this functionality is implemented in AML.

```java
public class AMLSetLiteralNode extends AMLBaseNode {
    @Children
    private AMLBaseNode[] elements;

    public AMLSetLiteralNode(AMLBaseNode[] elements) {
        this.elements = elements;
    }

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        var result = Arrays.stream(elements)
            .map(e -> e.executeGeneric(frame))
            .distinct()
            .map(o -> {
                if (!(o instanceof AMLObject)) {
                    throw new AMLRuntimeException("only AML objects are allowed in sets");
                }

                return (AMLObject) o;
            })
            .toArray(AMLObject[]::new);

        return AMLSet.of(result);
    }
}
```

Listing 67: *AMLSetLiteralNode* takes a list of nodes, evaluates each one of them, and builds a set from the results.

The individual child nodes of the set literal are passed through the constructor as an array of *AMLBaseNode* objects. In the *executeGeneric* method, each of these child nodes is evaluated and only the unique results are retained. Since the *executeGeneric* method of each child node returns instances of type *Object*, a runtime check has to be performed to verify that all child nodes evaluated to an object of type *AMLObject* since only objects of this type are allowed in an *AMLSet*. Lastly, a set is created out of all the results which is returned and therefore passed to a parent node of the *AMLSetLiteralNode*.

The creation of the node is fairly easy to comprehend as well. As explained in section 5.2 about lexing and parsing, the *AMLAntlrVisitor* is responsible for transforming the ANTLR AST to a Truffle AST and therefore creates all nodes of the AST. Since the visitor has a *visit* method for every grammar rule, it naturally has one for set literals which is unsurprisingly called *visitSetLiteral*. This method is presented in listing 68.

Through the *SetLiteralContext* object, the method can retrieve a list of all logical equivalence child nodes, call the *visitLogicEquivalence* method for each one of them to retrieve the

```java
@Override
public AMLBaseNode visitSetLiteral(AMLParser.SetLiteralContext ctx) {
    var elements = ctx.logicEquivalence()
        .stream()
        .map(this::visitLogicEquivalence)
        .toArray(AMLBaseNode[]::new);

    return new AMLSetLiteralNode(elements);
}
```

Listing 68: *visitSetLiteral* method of the *AMLAntlrVisitor*.

corresponding *AMLBaseNode* objects, and in the end create the set literal node.

Another rather simple Truffle node is the implementation of an if condition. As with the set literal, the grammar rule of the if condition is depicted once more in listing 69.

```
ifcond     → "if" logicEquivalence ":" thenBranch "otherwise" ":" elseBranch
```

Listing 69: Grammar rule of the if condition in AML.

```java
public class AMLIfNode extends AMLBaseNode {
    @Child private AMLBaseNode conditionNode;
    @Child private AMLBaseNode thenNode;
    @Child private AMLBaseNode elseNode;

    public AMLIfNode(AMLBaseNode conditionNode, AMLBaseNode thenNode,
                     AMLBaseNode elseNode) {
        this.conditionNode = conditionNode;
        this.thenNode = thenNode;
        this.elseNode = elseNode;
    }

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        var result = this.conditionNode.executeGeneric(frame);
        if (!(result instanceof AMLBoolean))
            throw new AMLRuntimeException("condition does not result in a boolean");

        if (((AMLBoolean) result).isTrue()) {
            return this.thenNode.executeGeneric(frame);
        } else {
            return this.elseNode.executeGeneric(frame);
        }
    }
}
```

Listing 70: *AMLIfNode* basically translates an AML if condition into a Java if condition.

The grammar specifies that an if condition in AML receives three children, one that represents the conditional expression, one for the then branch, and one for the else branch. Listing 70 shows the code of the node which implements this grammar rule.

The implementation represents the semantics of an if condition quite clearly. As specified in the grammar, the node receives three *AMLBaseNode* objects for the condition, the then branch, and the else branch. Using these three objects, the node basically translates the AML if condition into a Java if condition. It first executes the conditional expression, verifies that the result is a boolean, and executes either the then branch or the else branch depending on the result of the condition.

The creation of the node in the ANTLR visitor is even easier than that of the set literal. Its implementation is illustrated in listing 71.

```
@Override
public AMLBaseNode visitIfcond(AMLParser.IfcondContext ctx) {
    return new AMLIfNode(
        this.visitLogicEquivalence(ctx.logicEquivalence()),
        this.visitThenBranch(ctx.thenBranch()),
        this.visitElseBranch(ctx.elseBranch())
    );
}
```

Listing 71: The *visitIfcond* method just delegates most of the work to the other visit methods.

Since the if condition is basically just a hierarchy of four *AMLBaseNode* objects, the if node as a parent and the other three as children, the only task the *visitIfcond* method has to deal with is the creation of the parent if node. The duty of determining the children nodes are all delegated to their respective *visit* methods which naturally continue to distribute this job to other *visit* methods until the child nodes of the if condition are fully resolved.

While these two examples should have provided insight into the implementation of simple Truffle nodes, using this approach to structure all nodes of a Truffle language is not feasible. To prove this point, the next operation which will be discussed is of a more complex nature: the equality check. Again the grammar of this operation will be presented first in listing 72.

```
equality → negation (("=" | "≠") negation)*
```

Listing 72: Grammar rule of the equality check.

Although the grammar rule is about equality and inequality, the following illustration will focus on the equality operator. Through the information of the following paragraphs, however, the inequality operator could be implemented fairly easily as well. The novel problem, that has to be solved for the implementation of this grammar rule, is the support for multiple datatypes. The equality check is not only defined for a single datatype such as numbers, it should also be possible to compare sets or fractions. An initial naive approach to implement this node is laid out in listing 73.

```
public class AMLEqualNode extends AMLBaseNode {
    @Child private AMLBaseNode left;
    @Child private AMLBaseNode right;

    public AMLEqualNode(AMLBaseNode left, AMLBaseNode right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        var left = this.left.executeGeneric(frame);
        var right = this.right.executeGeneric(frame);

        if (left instanceof AMLNumber) {
            if (right instanceof AMLNumber) return AMLBoolean.of(left.equals(right));

            if (right instanceof AMLFraction) {
                var rightAsFrac = (AMLFraction) right;
                return AMLBoolean.of(left.equals(rightAsFrac.toNumber());
            }
        }
        // other nested type checks for fractions, sets, ...
        throw new AMLRuntimeException("equality not defined for "
            + left.getClass().getName() + " and " + right.getClass().getName());
    }
}
```

Listing 73: A possible, yet naive, implementation of the equality node

While possible for operations with just few supported datatypes, or languages with a small amount of datatypes in general, structuring nodes in this manner does not scale well with increasing operator complexity or number of datatypes. To implement complex nodes more elegantly, the Truffle framework provides another, separate library, called *Truffle DSL*, which offers a set of annotations to help language developers create language nodes. As always, the first step to actually use this library is to add it to the dependencies of the project. The Maven coordinates of the Truffle DSL library are presented in listing 74.

```
<dependency>
  <groupId>org.graalvm.truffle</groupId>
  <artifactId>truffle-dsl-processor</artifactId>
  <version>${graalvm.version}</version>
</dependency>
```

Listing 74: Maven coordinates of the Truffle DSL library.

The principal mechanism of the DSL is the definition of basically only the implementation of the node for individual datatypes. The term for every individual implementation is *specialization*. These are actually the very same specializations that were illustrated in section 4.2.2 about the Truffle Framework and its performance improvements through this concept.

Since the usage is easier to comprehend when underlaid with an example, listing 75 presents the implementation of the equality node in AML using the Truffle DSL.

```java
@NodeChild("left")
@NodeChild("right")
public abstract class AMLEqualNode extends AMLBaseNode {
    @Specialization
    protected AMLBoolean compareNumbers(AMLNumber left, AMLNumber right) {
        return AMLBoolean.of(left.equals(right));
    }

    @Specialization
    protected AMLBoolean compareFractions(AMLFraction left, AMLFraction right) {
        return AMLBoolean.of(left.equals(right));
    }

    @Specialization
    protected AMLBoolean compareNumberAndFraction(AMLNumber left, AMLFraction right) {
        return AMLBoolean.of(left.equals(right.toNumber()));
    }

    @Specialization
    protected AMLBoolean compareFractionAndNumber(AMLFraction left, AMLNumber right) {
        return AMLBoolean.of(left.toNumber().equals(right));
    }

    @Specialization
    protected AMLBoolean compareSets(AMLSet left, AMLSet right) {
        return AMLBoolean.of(left.equals(right));
    }
}
```

Listing 75: Implementation of the equality node using the Truffle DSL.

The first difference between the DSL and the standard approach of defining nodes is the usage of an abstract class instead of a concrete class. When using the Truffle DSL, the *executeGeneric* method is not provided in the implementation of the node anymore, therefore the class naturally has to be abstract. The Truffle DSL utilises all the available information of the abstract class to then generate all the necessary boilerplate code of the concrete node, including the *executeGeneric* method.

At the top of the class, it is possible to see the first type of information which is necessary to provide to Truffle. Similar to the naive implementation of the equality check, the node contains two child nodes which is signified by the *NodeChild* annotation of the Truffle DSL occurring twice. Additional information regarding the nodes can be passed to this annotation such as a name for the node. It will be obvious later that Truffle uses this information to improve the readability of the generated class.

Every specialization, i.e. an individual implementation for a set of datatypes marked by the parameters and the return type, has to be annotated with *Specialization*. Although all specializations use *AMLBoolean* as their return type in this case, different return types would be possible.

At build time Truffle processes the abstract class to generate a concrete class which is always named similarly to the abstract class with the suffix *Gen*. In this case, the abstract class is called *AMLEqualityNode*, therefore Truffle generates a concrete node called *AML-EqualityNodeGen*. The basic layout of such a class that Truffle generates is presented in listing 76.

```java
public final class AMLEqualNodeGen extends AMLEqualNode {
    @Child private AMLBaseNode left_;
    @Child private AMLBaseNode right_;
    @CompilationFinal private int state_0_;

    private AMLEqualNodeGen(AMLBaseNode left, AMLBaseNode right) {
        this.left_ = left;
        this.right_ = right;
    }

    @Override
    public Object executeGeneric(VirtualFrame frameValue) {
        // ...
    }

    private AMLBoolean executeAndSpecialize(Object leftValue, Object rightValue) {
        // ...
    }

    public static AMLEqualNode create(AMLBaseNode left, AMLBaseNode right) {
        return new AMLEqualNodeGen(left, right);
    }
}
```

Listing 76: Implementation of the equality node using the Truffle DSL.

First, the concrete node extends the abstract class with the specializations, and therefore has access to each implementation of these specializations. As specified, the node contains two child nodes, *left* and *right*. It is clear here that Truffle uses the name that was given through the *NodeChild* annotation to give the child nodes in the generated class this corresponding name. Additionally, the generated class defines a state variable which is a key mechanism of the framework to enable Graal to generate efficient and specialized machine code. This state variable is marked with the *CompilationFinal* annotation which specifies that, although the attribute is not final in the interpreter itself, Graal should view this attribute as final when generating machine code. Furthermore, while the constructor of the class is private, it provides a static *create* method to create an instance of this node.

In the generated *executeGeneric* method things start to get interesting. Most of this method is presented in listing 77. Although the code is definitely a bit obscure, it is nonetheless possible to decipher how it works. Generally, the generated class operates very similar to the previously provided naive implementation of the equality node. It first evaluates both the left and the right value, to then perform type checks to determine the correct specialization. In case it finds a matching specialization, it calls the method of the previously defined abstract class. However, not only the type is checked in the if conditions, the value of the state attribute is compared, too. The reason for this comparison of the state will be outlined soon. In case none of the type checks matches, a special compiler directive method is invoked. In

```
@Override
public Object executeGeneric(VirtualFrame frameValue) {
    int state_0 = state_0_;
    Object leftValue_ = this.left_.executeGeneric(frameValue);
    Object rightValue_ = this.right_.executeGeneric(frameValue);
    if ((state_0 & 0b1) != 0 && leftValue_ instanceof AMLNumber) {
        AMLNumber leftValue__ = (AMLNumber) leftValue_;
        if (rightValue_ instanceof AMLNumber) {
            AMLNumber rightValue__ = (AMLNumber) rightValue_;
            return compareNumbers(leftValue__, rightValue__);
        }
    }
    if ((state_0 & 0b110) != 0 && rightValue_ instanceof AMLFraction) {
        AMLFraction rightValue__ = (AMLFraction) rightValue_;
        if ((state_0 & 0b10) != 0 && leftValue_ instanceof AMLFraction) {
            AMLFraction leftValue__ = (AMLFraction) leftValue_;
            return compareFractions(leftValue__, rightValue__);
        }
        if ((state_0 & 0b100) != 0 && leftValue_ instanceof AMLNumber) {
            AMLNumber leftValue__ = (AMLNumber) leftValue_;
            return compareNumberAndFraction(leftValue__, rightValue__);
        }
    }

    // more state & type checks for fraction & number comparison & sets

    CompilerDirectives.transferToInterpreterAndInvalidate();
    return executeAndSpecialize(leftValue_, rightValue_);
}
```

Listing 77: Implementation of the equality node using the Truffle DSL.

case the *executeGeneric* method is compiled to machine code, this directive invalidates the machine code and restores the responsibility of execution back to the Java Truffle interpreter. Afterwards another generated method with the name *executeAndSpecialize* is executed which is shown in listing 78.

The *executeAndSpecialize* method basically does the same as the *executeGeneric* method with only one slight but important difference. Just as the *executeGeneric* method, *executeAndSpecialize* performs type checks to determine the correct specialization for the current parameters. However, instead of using the state in its if conditions, the *executeAndSpecialize* method sets the state in case a specialization matches, before running the method of the specialization. In case the node is not defined for the given parameters and therefore no specialization matches, an exception is thrown.

The mechanism of the generated class is therefore the following: in case the node is executed the first time, the *executeGeneric* method cannot find a matching specialization, since the state attribute has not been set yet. Thus, the *executeAndSpecialize* method is executed, the state is set and the node correctly evaluated. If the node is executed for the second time with the node children evaluating to the same datatypes and therefore the same specialization, the *executeGeneric* method can then perform this task alone because the comparison of the state attribute will be truthy. In case the node is executed very often from then on with each execution using the same specialization, Graal will try to compile the *executeGeneric* method

```java
private AMLBoolean executeAndSpecialize(Object leftValue, Object rightValue) {
    int state_0 = state_0_;
    if (leftValue instanceof AMLNumber) {
        AMLNumber leftValue_ = (AMLNumber) leftValue;
        if (rightValue instanceof AMLNumber) {
            AMLNumber rightValue_ = (AMLNumber) rightValue;
            this.state_0_ = state_0 = state_0 | 0b1;
            return compareNumbers(leftValue_, rightValue_);
        }
    }
    if (rightValue instanceof AMLFraction) {
        AMLFraction rightValue_ = (AMLFraction) rightValue;
        if (leftValue instanceof AMLFraction) {
            AMLFraction leftValue_ = (AMLFraction) leftValue;
            this.state_0_ = state_0 = state_0 | 0b10;
            return compareFractions(leftValue_, rightValue_);
        }
        if (leftValue instanceof AMLNumber) {
            AMLNumber leftValue_ = (AMLNumber) leftValue;
            this.state_0_ = state_0 = state_0 | 0b100;
            return compareNumberAndFraction(leftValue_, rightValue_);
        }
    }

    // ...

    throw new UnsupportedSpecializationException(
        this, new Node[] {this.left_, this.right_}, leftValue, rightValue
    );
}
```

Listing 78: Implementation of the equality node using the Truffle DSL.

to machine code. Since Truffle instructed Graal to view the state attribute as constant through the *CompilationFinal* annotation, Graal can just omit every specialization except the currently active one because in Graal's view none of the other if conditions can ever be true. The resulting machine code is therefore very specific to its expected inputs and much smaller then the original *executeGeneric* method. In case the specialization suddenly does not match anymore, the call of the compiler directive, as previously mentioned, delegates the execution back to the Truffle interpreter. Afterwards, the interpreter calls *executeAndSpecialize* and updates the state attribute accordingly. Using the Truffle DSL to build nodes therefore does not only reduce the amount of code language developers have to write, it also grants performance benefits by default.

The last point which has to be discussed is the creation of the equality node in the ANTLR visitor. Since the generated class of Truffle provides a *create* method to build an instance, the visitor naturally has to utilise this static method to create the node. Listing 79 presents the *visit* method for the equality grammar rule.

In case the ANTLR equality node only contains one child, it is immediately clear that no equality check has to be performed and the *visitNegation* method, i.e. the next rule in the grammar hierarchy, can be called directly. In case the equality node has multiple children, the *AMLBaseNode* of the left-most *negation* is determined first. Afterwards, the method iterates

```java
@Override
public AMLBaseNode visitEquality(AMLParser.EqualityContext ctx) {
    if (ctx.getChildCount() == 1) {
        return this.visitNegation(ctx.negation(0));
    } else {
        AMLBaseNode ret = this.visitNegation(ctx.negation(0));

        int negationCounter = 1;
        int childrenCounter = 1;

        while (childrenCounter < ctx.getChildCount()) {
            var op = (TerminalNode) ctx.getChild(childrenCounter);

            switch (op.getSymbol().getType()) {
                case AMLLexer.EQ:
                    ret = AMLEqualNodeGen.create(
                        ret,
                        this.visitNegation(ctx.negation(negationCounter))
                    );
                    break;
                case AMLLexer.NEQ:
                    ret = AMLNotEqualNodeGen.create(
                        ret,
                        this.visitNegation(ctx.negation(negationCounter))
                    );
                    break;
                default:
                    throw new AMLParserException("Unsupported equality operation");
            }

            negationCounter++;
            childrenCounter = childrenCounter + 2;
        }

        return ret;
    }
}
```

Listing 79: Creation of the (in)equality nodes in the visitor.

over the remaining children of the current ANTLR node. In each iteration, the current operation is first checked in a switch statement. Here the constants for the terminals, which were specified in the ANTLR grammar file, come into play to make it easier to determine the correct symbol of the operation. An equality or inequality node is then created using the *create* method of the generated Truffle node class and the iteration is completed. Throughout iterations, the node of the previous iteration is nested inside the node of the current iteration. Multiple equality checks will therefore in the end build a hierarchy of nested nodes with the left-most equality check being the deepest node in the hierarchy and thus the first to be executed.

Although the usage of the Truffle DSL is usually superior compared to the approach of manually defining concrete nodes, the latter technique has its advantages as well. The nodes which the DSL generates generally have all a similar structure: first evaluate all children, then check which specialization should be utilized, and finally return the result. In some cases, it might be necessary to not evaluate a certain child node yet. The implementation of the if condition presented one of these cases where the then and else child nodes were executed depending on the result of the conditional node. In a case such as this, the approach of manually defining the node and therefore the *executeGeneric* method is more fitting. Therefore, as a rule of thumb, the Truffle DSL should be employed for most nodes, while the approach of manually defining nodes should be utilised if the generic node structure of the DSL is not applicable.

5.5 Variables

While the previous section dealt with pure nodes whose principal characteristic was to compute a result using a set of inputs and return it, other types of nodes have to interact with the surrounding environment of the language implementation to fulfil their respective task. Among these types of nodes are the ones that handle the writing and reading of global and local variables. Although the usage of the assignment operator in AML is only possible in the global scope and therefore will always define a global variable, local variables are created for parameters of functions or in quantifications and set builders. For better understanding and as a recapitulation, listing 80 highlights how the two types of variables are created and read in AML.

```
-- Write and read (use) a global variable
x ← 1 + 2;
x ^ 2;

-- Function parameters (e.g. y) or variables in
-- a quantification or set builder initializer (e.g. z) are local variables.
someFunction: (y) → x + y;
∀(z ∈ {1, 2, 3} : z - x < 0);
```

Listing 80: Reading and writing global and local variables in AML.

The essential concept in Truffle related to variables, namely the *frame*, was already introduced in a previous section. As it was outlined, there are two types of frames, a *MaterializedFrame* which holds global variables and is part of the language context in AML, and the *VirtualFrame* for local variables that is created by Truffle when evaluating an AST and is passed throughout the whole AST through the *executeGeneric* method in AML. In total, there are three nodes which operate on these frames to provide the functionality of variables: one to write global variables, one to write local variables, and one to read both types of variables.

The easiest of these nodes is responsible for the definition of local variables and thus will be attended to first. The implementation is depicted in listing 81 and uses the Truffle DSL. The node makes use of an additional annotation of the Truffle DSL, the *NodeField*. While the *NodeChild* annotation indicates to Truffle to create an AST node attribute inside the generated class, the *NodeField* annotation instructs the framework to add a common attribute, which is not an AST node, to the generated class and its constructor. Additionally, the framework generates implementations for all abstract getter and setter methods which the language

```
@NodeChild("value")
@NodeField(name = "identifier", type = String.class)
public abstract class AMLWriteLocalVariableNode extends AMLBaseNode {
    protected abstract String getIdentifier();

    @Specialization
    protected Object write(VirtualFrame frame, Object value) {
        frame.setObject(
            frame.getFrameDescriptor().addFrameSlot(getIdentifier()),
            value
        );

        return value;
    }
}
```

Listing 81: Implementation of the AML node which defines local variables.

developer defines in the abstract class such as *get + field name* or *set + field name*. In case of the AML local variable write node, an attribute for the identifier of the variable has to be generated in order to create a frame slot for it in the virtual frame. In addition to the identifier, the node has an additional AST node as a child which, when evaluated, determines the value of the variable that should be written.

The node contains only a single specialization since the implementation is the same across all AML datatypes. In order to use the *VirtualFrame* in a specialization, it has to be present as the first parameter of that specialization. The generated *executeGeneric* method of the Truffle class will then pass the frame to the method of the specialization. The remaining part of the specialization is self-explanatory: a new frame slot is created using the frame descriptor of the *VirtualFrame* which is then set to the computed value of the child node. Since this node is only utilised in conjunction with functions, quantification, or set builders, examples for the creation of this node will be presented in the next section which illustrates the implementation of these functionalities of AML.

The definition of global variables in AML is nearly as simple as it is with local variables. Since the node that enables the creation of global variables provides an implementation of a rule of the AML language grammar, namely the *assignment* grammar rule, the rule will be presented once again in listing 82.

```
assignment → (IDENTIFIER "←" assignment) | composition | logicEquivalence
```

Listing 82: Grammar rule of an assignment in AML.

Because the assigment rule resolves to the *composition* or the *logicEquivalence* nonterminal only if the assignment rule does not actually consist of an assignment, the global variable node supplies an implementation just for the left-most possibility of the rule. Analogous to local variables, a global variable assignment consists of an identifier and a child node which determines the value of the variable. The resulting implementation is therefore quite similar, as shown in listing 83, with only the utilised frame being different.

```
@NodeChild("value")
@NodeField(name = "globalFrame", type = MaterializedFrame.class)
@NodeField(name = "identifier", type = String.class)
public abstract class AMLWriteGlobalVariableNode extends AMLBaseNode {
    public abstract MaterializedFrame getGlobalFrame();
    public abstract String getIdentifier();

    @Specialization
    public Object write(Object value) {
        var slot = getGlobalFrame()
            .getFrameDescriptor()
            .findFrameSlot(getIdentifier());

        if (slot != null)
            throw new AMLRuntimeException(
                "attempt to rewrite already defined variable " + getIdentifier()
            );

        getGlobalFrame().setObject(
            getGlobalFrame().getFrameDescriptor().addFrameSlot(getIdentifier()),
            value
        );

        return value;
    }
}
```

Listing 83: Implementation of the definition of a global variable in AML.

Similar to the identifier of the new variable, an attribute for the global frame, to which the variable will be written, is created through the *NodeField* annotation. Before writing the variable to the global frame, however, the frame descriptor is first examined to check whether a slot with the same identifier already exists. If such a slot is already present, the variable definition would actually update the value of the variable and therefore violate the immutability of AML. This verification is not necessary for local variables since AML only allows a single variable for the initializers of set builders or quantification and the language does not allow multiple parameters with the same name for functions. Therefore, the redefinition of local variables is not possible and such a verification not necessary. Other than the highlighted differences, the remaining part of the node is completely similar to the node which handles the definition of local variables.

To create the node, the necessary attributes need to be populated by the ANTLR visitor. This is achieved as presented in listing 84. By using the ANTLR context of the current position inside the ANTLR AST, the method can determine if an actual assignment was present in the source code, or if it should call the next *visit* method in the grammar hierarchy. In case of an existing assignment, *visitAssignment* creates the node with all necessary information and returns it. Since the AML language context was passed from the language class through the parsing methods to the ANTLR visitor class, the visitor has access to the context and can therefore share the global frame with the new node. The *visitAssignment* method furthermore provides the child node for the value of the variable to the newly created node and the

```
@Override
public AMLBaseNode visitAssignment(AMLParser.AssignmentContext ctx) {
    if (ctx.assignment() != null) {
        return AMLWriteGlobalVariableNodeGen.create(
            this.visitAssignment(ctx.assignment()),
            this.context.getGlobalFrame(),
            ctx.IDENTIFIER().getSymbol().getText()
        );
    } else if (ctx.composition() != null) {
        return this.visitComposition(ctx.composition());
    } else {
        return this.visitLogicEquivalence(ctx.logicEquivalence());
    }
}
```

Listing 84: Creation of a node for the definition of a global variable in AML.

identifier through which the variable should be accessible.

Lastly, after having the definition of local and global variables established, the reading of variables has to be implemented as well. Since AML always tries to find the requested variable in the local *VirtualFrame* first and falls back to the global *MaterializedFrame* if the variable is not present in the local frame, only one node for the reading of variables is sufficient. In the grammar, the usage of variables is part of two grammar rules which are depicted in listing 85. Both the *numPrimary* and the *setPrimary* rules are nearly at the bottom of the grammar hierarchy and are therefore evaluated quite early when executing the Truffle AST. The reading of variables is offered by these rules through the *IDENTIFIER* nonterminal.

```
numPrimary → call | number | IDENTIFIER | "(" logicEquivalence ")"
setPrimary → call | setLiteral | setEllipsis | setBuilder | IDENTIFIER |
    "(" logicEquivalence ")"
```

Listing 85: Grammar rules which contain the reading of variables.

The task of resolving these identifiers to values is performed by the node illustrated in listing 86. Since the implementation does not utilise any new concepts or functionalities of the Truffle framework, it should be readily understandable. The node is the first of all presented nodes which does not contain any additional child nodes. Thus, this node will always mark the bottom of a certain branch in the whole AST of the program. As with the previous nodes related to variables, the node requires an attribute for the identifier of the variable as well as a reference to the global *MaterializedFrame*. The node first tries to fetch the variable from its local *VirtualFrame* and falls back to a global read in case an exception is thrown. If the variable is not present in both of the frames, an exception is thrown by the node and the evaluation of the AST is interrupted.

The creation of the node in the *visitSetPrimary* and *visitNumPrimary* methods is largely the same. Listing 87 shows the creation by using the *visitNumPrimary* method as an example. In case the current *numPrimary* consists out of a *logicEquivalence*, a *call* or a *number*, the

```
@NodeField(name = "globalFrame", type = MaterializedFrame.class)
@NodeField(name = "identifier", type = String.class)
public abstract class AMLReadVariableNode extends AMLBaseNode {
    public abstract String getIdentifier();
    public abstract MaterializedFrame getGlobalFrame();

    @Specialization
    protected Object read(VirtualFrame frame) {
        return readVar(frame, getGlobalFrame(), getIdentifier());
    }

    public Object readVar(VirtualFrame localFrame,
                          MaterializedFrame globalFrame, String identifier) {
        try {
            var slot = localFrame.getFrameDescriptor().findFrameSlot(identifier);
            return Objects.requireNonNull(localFrame.getObject(slot));
        } catch (NullPointerException | FrameSlotTypeException ex) {
            // local read failed, try global read
            return globalRead(globalFrame, identifier);
        }
    }

    private Object globalRead(MaterializedFrame globalFrame, String identifier) {
        try {
            var slot = globalFrame.getFrameDescriptor().findFrameSlot(identifier);
            return Objects.requireNonNull(globalFrame.getObject(slot));
        } catch (NullPointerException | FrameSlotTypeException ex) {
            throw new AMLRuntimeException("could not retrieve: " + identifier);
        }
    }
}
```

Listing 86: AST node implementation for resolving variables.

```
@Override
public AMLBaseNode visitNumPrimary(AMLParser.NumPrimaryContext ctx) {
    if (ctx.logicEquivalence() != null) {
        return this.visitLogicEquivalence(ctx.logicEquivalence());
    } else if (ctx.IDENTIFIER() != null) {
        return AMLReadVariableNodeGen.create(
            this.context.getGlobalFrame(),
            ctx.IDENTIFIER().getSymbol().getText()
        );
    } else if (ctx.call() != null) {
        return this.visitCall(ctx.call());
    }

    return this.visitNumber(ctx.number());
}
```

Listing 87: Creation of the variable read node using *visitNumPrimary* as example.

corresponding *visit* method is executed. If it is composed of an identifier, however, the previously introduced node is created by passing the global frame and the identifier as a string to a new instance of such a node.

The implementation of these three nodes is all what is necessary to enable the functionality of reading and writing variables in AML and enhance the utility of the language by quite a bit. Although other languages could demand more effort due to features such as flexibly nested scopes, the prohibition of variable shadowing, or closures, the details outlined in this section should provide a fundamental overview of how variables could be implemented when using the Truffle framework.

5.6 Functions

The most complex feature with regard to its implementation is the support of functions in AML. Before covering the concrete definition and invokation of functions, however, it is important to take a step back and think about the underlying principle of functions. In many languages, AML included, a function is actually a concrete application of a more abstract concept of the language. A good example can be found in a language such as Python. Python supports at least three possibilites to define a somewhat similar "callable construct": lambdas, methods, and functions. A lambda is an anonymous function with a concise syntax for its definition and can be passed as an argument to a function. A method is basically also a function, but is defined in a class and has access to other methods or attributes of that class. On top of that, Python offers the creation of actual standalone functions as well.

Similar to Python, AML offers different types of function-related functionalities too. First, the language supports simple named functions, i.e. an expression with a name assigned to it that can be executed as often as necessary. Named functions can moreover be utilised to build composed functions which is a nesting of two functions, or iterated functions which allows a function to be composed a certain amount of times with itself. All three types of functions can be called by the developer at will. Lastly, the set builder and the different types of quantification are based on a nested expression which is invoked for each element of a certain set as well. Listing 88 depicts all these different applications of functions in AML to demonstrate the connection between these capabilities.

```
-- Named functions, composed functions, and iterated functions.
f: (x) → x + 1;
g: (x) → x + 1;
h ← f ∘ g;
k ← f ^ 10;

-- These three types of functions can be directly called.
f(1); g(1); h(1); k(1);

-- Set builder and quantification use a callable expression as well.
-- Here the "x mod 2 = 0" expression.
{ x ∈ {1, ..., 10} | x mod 2 = 0 };
∀(x ∈ {2, 4, 6}: x mod 2 = 0);
```

Listing 88: Reading and writing global and local variables in AML.

The general point is that some language elements, in this case functions, can reappear in the form of other, slightly different language elements. Due to such similarities, it is important to think about a good design regarding the structure of these functionalities to not have too much duplicate code in the implementation while keeping the language flexible. The advantage of using Truffle and therefore Java to implement a language is the availability of the full power of an object oriented language to build a clean type hierarchy using classes and interfaces. Such a type hierarchy for functions is depicted in figure 9 by using AML as an example.

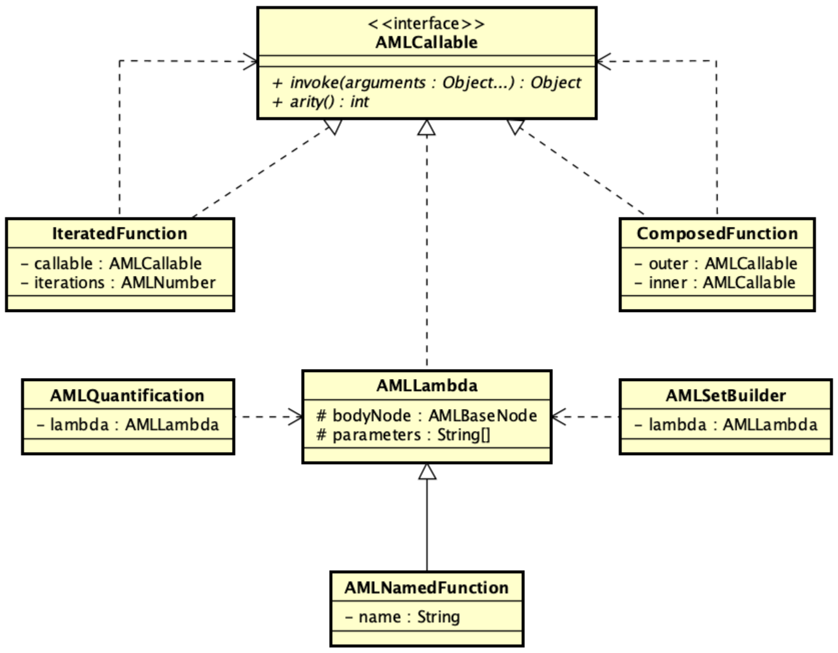


Figure 9: Type hierarchy of functions in AML.

The most basic type in the whole hierarchy is the *AMLCallable* interface. In order to implement it, a class has to provide two methods, a method called *invoke* which takes an array of objects, executes some logic, and returns an object again, and a method called *arity* which should return the amount of expected arguments as an integer. The most sophisticated implementation of this interface is the *AMLLambda* class which basically wraps an *AMLBaseNode*, marking its expression, with a list of parameters and executes this embedded node when *invoke* is called. The lambda class can then be reused for the named function which basically is just a lambda with an additional *name* attribute, and for the quantification and set builder nodes. The iterated and composed functions implement the *AMLCallable* interface as well and use the interface to flexibly allow any implementation of that interface to be iterated or composed. In the remainder of this section, parts of this type hierarchy will be individually presented and discussed.

The *AMLCallable* interface is self-explanatory and can be completely inferred from the previously presented class diagram. Nonetheless, for the sake of completion, it is presented in listing 89.

```java
public interface AMLCallable {
    Object invoke(Object... arguments);
    int arity();
}
```

Listing 89: The *AMLCallable* interface.

With the *AMLLambda* class things start to get interesting. Its complete definition is outlined in listing 90.

```java
public class AMLLambda implements AMLCallable {
    protected AMLBaseNode bodyNode;
    protected String[] parameters;

    public AMLLambda(AMLBaseNode bodyNode, String[] parameters) {
        this.bodyNode = bodyNode;
        this.parameters = parameters;
    }

    public Object invoke(Object... arguments) {
        if (arguments.length != arity())
            throw new AMLRuntimeException(arity() +
                " arguments expected but got " + arguments.length);

        List<AMLBaseNode> argumentNodes = new ArrayList<>();
        for(int i = 0; i < arity(); i++) {
            final var arg = arguments[i];

            argumentNodes.add(AMLWriteLocalVariableNodeGen.create(
                new AMLBaseNode() {
                    @Override
                    public Object executeGeneric(VirtualFrame frame) {
                        return arg;
                    }
                },
                parameters[i]
            ));
        }

        argumentNodes.add(bodyNode);

        return Truffle.getRuntime().createCallTarget(
            new AMLRootNode(
                new AMLProgramNode(argumentNodes.toArray(AMLBaseNode[]::new))
            )
        ).call();
    }

    public int arity() { return parameters.length; }
}
```

Listing 90: Structure of the *AMLLambda* class

As previously mentioned, the class receives an *AMLBaseNode* which marks the embedded expression that can be invoked at will. It furthermore receives an array of strings which specify the names of its parameters. This array of parameter names provides a very simple way of implementing the *arity* method of the interface by simply returning the length of the array. The *invoke* method starts out by verifying whether the amount of arguments that were given correctly matches the arity of the lambda. Afterwards, the node for local variables that was introduced in the previous section comes into play for the first time. After initialising an empty list for *AMLBaseNode* objects, the *invoke* method iterates over all its given arguments and adds a new instance of the local variable writing node to the list. Since the variable writing node expects an instance of *AMLBaseNode* which it evaluates to determine the value of the variable, an anonymous instance of *AMLBaseNode* is constructed which simply returns the argument of the current iteration. The name of the local variable that is created through the node is specified through the array which holds the names of all the parameters of the lambda. After this iteration, the actual expression of the lambda is added as the final element to the list of *AMLBaseNode* objects. This list is then wrapped in a program node, i.e. a node which sequentially executes all its children nodes, which in turn is wrapped by an instance of a root node and a call target. However, instead of letting Truffle call this call target as is done with the actual root node of the program AST, the call target is manually invoked and its result returned. Once the call target is executed, all the local variable nodes are evaluated sequentially to establish all necessary values for the lambda. Afterwards, the embedded expression of the lambda is executed in this environment of variables and its result returned. Since Truffle creates a new virtual frame whenever a new call target is created and executed, the local variables that will be created at the beginning of the lambda evaluation will not interfere with any variables of the outer scope of the lambda. To summarise the complete mechanism in simple words, each call to *invoke* can be described as a dynamic creation of a very small, internal AST which is immediately evaluated and its result returned.

The lambda implementation of the callable interface can be employed for multiple other features of AML. The first and simplest refinement of a lambda is the named function which extends a lambda by additionally storing a name for it and thus removing its anonymity. The definition of the named function is presented in listing 91.

```java
public class AMLNamedFunction extends AMLLambda {
    private final String name;

    public AMLNamedFunction(String name, AMLBaseNode bodyNode, String[] parameters) {
        super(bodyNode, parameters);
        this.name = name;
    }
}
```

Listing 91: A simple refinement of a lamba: the named function.

Whenever a named function is defined somewhere in AML source code, the function is stored in the global frame of the context and can be accessed from any other place. As with global variables, a Truffle AST node has to be created which performs the task of creating a named function instance and storing it in the global frame.

```
function → IDENTIFIER ":" "(" params ")" "→" expression
params   → IDENTIFIER? | (IDENTIFIER ",")+ IDENTIFIER
```

Listing 92: Grammar of a named function definition.

```java
public class AMLDefineFunctionNode extends AMLBaseNode {
    private String name;
    private String[] parameters;
    private MaterializedFrame globalFrame;
    private AMLBaseNode bodyNode;

    public AMLDefineFunctionNode(String name, String[] parameters,
                                 AMLBaseNode bodyNode, MaterializedFrame globalFrame) {
        this.name = name;
        this.parameters = parameters;
        this.bodyNode = bodyNode;
        this.globalFrame = globalFrame;
    }

    public Object executeGeneric(VirtualFrame frame) {
        if (hasDuplicateParameterNames())
            throw new AMLRuntimeException("function with duplicate argument names");

        var slot = this.globalFrame.getFrameDescriptor().findFrameSlot(this.name);

        if (slot != null)
            throw new AMLRuntimeException("redefinition of function " + this.name);

        slot = this.globalFrame.getFrameDescriptor().addFrameSlot(this.name);
        var func = new AMLNamedFunction(this.name, this.bodyNode, parameters);
        this.globalFrame.setObject(slot, func);
        return func;
    }

    private boolean hasDuplicateParameterNames() {
        return Arrays.stream(this.parameters).distinct().count() !=
            this.parameters.length;
    }
}
```

Listing 93: Implementation of a Truffle node which defines named functions.

According to the grammar of a named function definition in listing 92, a node creating such a function would have to receive an identifier for its name, a list of identifiers which mark its parameters, and an *AMLBaseNode* which represents the expression of the function. On top of that, the Truffle node requires a reference to the global frame to correctly store the named function in that frame. Listing 93 depicts the implementation of the node that defines functions and requires all the aforementioned information through its constructor. The first check, which is performed before the function definition, verifies whether all parameter names have a unique name. Without this check, the invokation of the function would potentially define local variables of the function environment multiple times. The second check ascertains that the named function was not already defined and is not present in the global frame yet. Afterwards a new instance of a named function can be created and stored inside the global frame.

```java
public AMLBaseNode visitFunction(AMLParser.FunctionContext ctx) {
    var argumentNames = ctx.params().IDENTIFIER()
        .stream().map(i -> i.getSymbol().getText()).toArray(String[]::new);

    return new AMLDefineFunctionNode(
        ctx.IDENTIFIER().getSymbol().getText(),
        argumentNames,
        this.visitExpression(ctx.expression()),
        context.getGlobalFrame()
    );
}
```

Listing 94: Creation of the function definition node.

The creation of the function definition node in the ANTLR visitor, as outlined in listing 94, has to first acquire all identifiers of the *params* rule which is contained inside the *function* grammar rule. Afterwards it can create the node by additionally extracting the name of the function, delegating the task of retrieving the expression of the function to the *visitExpression* method, and passing a reference to the global frame of the language context.

However, the *AMLLambda* implementation is not only used in terms of inheritance but also composition. Examples include the set builder and quantification in AML of which the former will be presented next. Contrary to the named function, a set builder is not stored in any frame but is evaluated immediately, therefore the definition and implementation of the set builder is solely defined inside a Truffle node.

```
setBuilder → "{" IDENTIFIER "∈" setPrimary "|" logicEquivalence "}"
```

Listing 95: Grammatical specification of the set builder in AML.

According to the grammatical specification in listing 95, a set builder expression is composed of three parts: a body of the builder in form of the *logicEquivalence* rule which is repeatedly called to determine whether an element should be part of the resulting set, a set of which every element will be used once to execute the body of the builder, and an identifier with which the body of the builder references the current element. This composition can be very cleanly implemented by making use of the *AMLLambda* class.

The implementation of the set builder node in listing 96 illustrates how a lambda is utilised for this purpose. The constructor of the node receives the initializer of the builder, i.e. the *AMLBaseNode* which evaluates to the set over which the builder iterates, as well as the identifier and the body of the builder which are wrapped by an *AMLLambda*. The builder is executed by first evaluating the initializer and verifying whether the result resolves to a set. Afterwards, the iteration over the set can be performed by invoking the lambda with each element of the set, verifying whether the result was correctly evaluated to a boolean, and

```java
public class AMLSetBuilderNode extends AMLBaseNode {
    @Child private AMLBaseNode initializer;
    private AMLLambda lambda;

    public AMLSetBuilderNode(String identifier, AMLBaseNode initializer,
                             AMLBaseNode body) {
        this.initializer = initializer;
        this.lambda = new AMLLambda(body, new String[] { identifier });
    }

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        var initObj = initializer.executeGeneric(frame);
        if (!(initObj instanceof AMLSet))
            throw new AMLRuntimeException("only sets are allowed as initializer");

        var set = ((AMLSet) initObj).stream().filter(o → invokeBody(o).isTrue())
            .toArray(AMLObject[]::new);

        return AMLSet.of(set);
    }

    private AMLBoolean invokeBody(AMLObject obj) {
        var result = lambda.invoke(obj);
        if (result instanceof AMLError)
            throw new AMLRuntimeException(((AMLError) result).getMessage());

        if (!(result instanceof AMLBoolean))
            throw new AMLRuntimeException("body of set builder must return a boolean");

        return (AMLBoolean) result;
    }
}
```

Listing 96: Implementation of the set builder node.

only retaining the elements of the set for which the result was truthy. With these withheld elements of the original set, a new instance of *AMLSet* can be constructed and returned.

The creation of the node does not come with any new revelation. Through the *SetBuilderContext* all the necessary information for the set builder is acquired by extracting the identifier of each single element immediately and by calling the other *visit* methods to obtain the two necessary *AMLBaseNode* objects. This creation is displayed in listing 97.

However, not all function types of the AML hierarchy are related to the *AMLLambda* implementation. While much simpler in nature, the composed function and the iterated function provide additional implementations of the *AMLCallable* interface.

The composed function is basically a simple nesting of two callables where the result of the inner callable is directly passed as an argument to the invocation of the outer callable. Its implementation is therefore quite short, as can be viewed in listing 98. Since the class uses the *AMLCallable* type for both of its attributes, the implementation of the composed function is very flexible and can take any type of function which implements the callable interface. Furthermore, due to the fact that function composition is only possible with functions which have an arity of one, the resulting composed callable has to have an arity of one as well.

```
public AMLBaseNode visitSetBuilder(AMLParser.SetBuilderContext ctx) {
    return new AMLSetBuilderNode(
        ctx.IDENTIFIER().getSymbol().getText(),
        this.visitSetPrimary(ctx.setPrimary()),
        this.visitLogicEquivalence(ctx.logicEquivalence())
    );
}
```

Listing 97: Creation of the set builder in the ANTLR visitor.

```
public class AMLComposedFunction implements AMLCallable {
    private AMLCallable outer, inner;

    public AMLComposedFunction(AMLCallable outer, AMLCallable inner) {
        this.outer = outer;
        this.inner = inner;
    }

    @Override
    public Object invoke(Object... arguments) {
        return outer.invoke(inner.invoke(arguments));
    }

    @Override
    public int arity() { return 1; }
}
```

Listing 98: Definition of a composed function.

The grammar rule of the function composition in AML is very straightforward. Basically, the only way to compose two functions is to combine the variable names of these functions with the composition operator. Whether a named function, an iterated function, or a composed function is stored in the variables does not matter to the function composition. The rule itself is depicted in listing 99.

```
composition → IDENTIFIER "∘" IDENTIFIER
```

Listing 99: Grammar rule for creating a function composition.

Both identifiers in the grammar rule are modelled as variable read operations which the node for function composition has to evaluate to obtain the two callable instances. Since this evaluation would therefore consist of checking the types of both results of the variables read operations, the Truffle DSL allows to simplify the code of the node by directly expecting *AMLCallable* for both values through its specialization.

In order to prohibit the creation of invalid composed functions, the node in listing 100 furthermore verifies that both callables have an arity of one and throws an exception other-

```java
@NodeChild("left")
@NodeChild("right")
public abstract class AMLComposeFunctionsNode extends AMLBaseNode {
    @Specialization
    public AMLComposedFunction composeFunctions(AMLCallable left, AMLCallable right) {
        if (left.arity() != 1 && right.arity() != 1)
            throw new AMLRuntimeException("can only compose callables with arity = 1");

        return new AMLComposedFunction(left, right);
    }
}
```

Listing 100: *AMLComposeFunctionsNode* creates a new composed function using two callables.

wise. The result of this node can be combined with an assignment statement in AML code to assign a name to the composed function that is returned and to be able to reference it later on.

Lastly, the creation of this node, as handled by the ANTLR visitor in listing 101, creates the two variable read nodes, one for each identifier, and provides them to a new instance of the generated class of the function composition node.

```java
public AMLBaseNode visitComposition(AMLParser.CompositionContext ctx) {
    return AMLComposeFunctionsNodeGen.create(
        AMLReadVariableNodeGen.create(
            this.context.getGlobalFrame(), ctx.IDENTIFIER(0).getSymbol().getText()
        ),
        AMLReadVariableNodeGen.create(
            this.context.getGlobalFrame(), ctx.IDENTIFIER(1).getSymbol().getText()
        )
    );
}
```

Listing 101: *visit* method for the function composition in AML.

Although the previous illustrations should suffice to understand the structure and composition of functional concepts in AML, one final question remains. Since some of the concrete implementations of the callable interface can be stored as a variable, the manual invocation of these functions in AML source code has to be implemented as well. Luckily, due to the *AMLCallable* interface, a function call can be implemented in a rather straightforward manner. Function calls in AML are performed very similarly as in most other languages by using opening and closing parentheses after the identifier of the function. Inside the parentheses, a comma-separated list of arguments can be passed to this function. Again, as in other languages, each argument can be a complex expression and therefore has to be evaluated before passing it to the function. This layout of a function call is expressed in the grammar rules in listing 102.

With the help of the grammar, it is already possible to deduct certain requirements which the implementation of the function call has to fulfil. First, the call has to somehow resolve

```
call      → IDENTIFIER "(" arguments ")"
arguments → logicEquivalence? | (logicEquivalence ",")+ logicEquivalence
```

Listing 102: Grammar rules of a valid function call in AML.

```java
public class AMLCallNode extends AMLBaseNode {
    @Child private AMLBaseNode callableVar;
    @Children private AMLBaseNode[] arguments;

    public AMLCallNode(AMLBaseNode callableVar, AMLBaseNode[] arguments) {
        this.callableVar = callableVar;
        this.arguments = arguments;
    }

    @Override
    public Object executeGeneric(VirtualFrame frame) {
        var evaledArgs = Arrays.stream(arguments)
            .map(a -> a.executeGeneric(frame))
            .toArray();

        var result = retrieveFunction(frame).invoke(evaledArgs);

        if (result instanceof AMLError)
            throw new AMLRuntimeException(((AMLError) result).getMessage());

        return result;
    }

    private AMLCallable retrieveFunction(VirtualFrame localFrame) {
        var obj = callableVar.executeGeneric(localFrame);
        if (!(obj instanceof AMLCallable))
            throw new AMLRuntimeException("variable is not a callable");

        return (AMLCallable) obj;
    }
}
```

Listing 103: Truffle node implementation for a function call.

the variable of the identifier to a callable object. Similar to the function composition, the easiest solution for achieving this is by injecting a variable read node for the identifier into the function call node which the call node can then evaluate to obtain the callable object. Secondly, the call has to evaluate each argument sequentially to then pass the results of every argument to this obtained function. The Java code for the implementation of such a node is presented in listing 103. The *retrieveFunction* method in the class accomplishes the first of the previously outlined requirements. It executes the first child node to resolve the variable, checks the value of that variable to make sure it is an *AMLCallable*, and returns it. In the *executeGeneric* method, the second requirement is performed. The array of child nodes which represent the arguments are individually evaluated and then passed to the *invoke* method of the callable object. At the end of the method, the result of the function invocation is checked to determine whether an error occurred during the execution of the function, and the result is finally returned. All of these child nodes, which the call node utilises for its evaluation, are again created and prepared in the *visitCall* method of the ANTLR visitor and are passed to the constructor of the call node to create a new instance, as shown in listing 104.

```
public AMLBaseNode visitCall(AMLParser.CallContext ctx) {
    var functionVar = AMLReadVariableNodeGen.create(
        this.context.getGlobalFrame(),
        ctx.IDENTIFIER().getSymbol().getText()
    );

    var arguments = ctx.arguments()
        .logicEquivalence()
        .stream()
        .map(this::visitLogicEquivalence)
        .toArray(AMLBaseNode[]::new);

    return new AMLCallNode(functionVar, arguments);
}
```

Listing 104: Creation of the call node.

This rather code heavy section about the implementation of functions aimed at showing two things. One goal of this section was to present fundamental concepts and a potential approach to implement functions, or callable objects in general, in a language written with the Truffle framework. The second goal was to illustrate the importance of creating a well-designed structure or type hierarchy for the more complex elements of the language. Although this section proved this point by using functions as an example, the same could be necessary for other language elements as well. Many languages, such as C++, have different ways of representing structured data, like structs and classes. While both have their distinctive features, they nonetheless share certain similarities. Another example is the implementation of scopes. Especially in object oriented languages, different rules for the various language elements exist regarding their respective scopes. While the scope inside a class provides access to the current instance through *this* or *self*, the scope inside a function or the global scope does not provide any *this* or *self* reference. So yet again, although all scopes provide a basic set of universal features, concrete refinements of a scope could require certain individualities. Through such a design, as outlined in this section, not only do the language elements offer a certain level of flexibility which is otherwise hard to achieve, the resulting implementation can furthermore be simpler and shorter in its size. Investing the time and effort to create a capable and structured implementation is therefore well worth it.

5.7 Interoperability

After having covered all details regarding the implementation of the language features of AML, the purpose of the final section of this chapter is to elaborate how a language has to be enhanced to support interoperability with other Truffle languages, and to provide an overview of how a Truffle language can be employed in practice.

In order for a host language to embed a guest language in GraalVM, the host language has to create a so called *polyglot context*. During the creation of the context, the host language can specify certain properties of it, such as the list of permitted languages which the context can execute or whether the embedded code is allowed to perform actions such as file operations or use threading. While some Truffle general-purpose languages such as Java provide a very explicit API to configure this polyglot context, other languages like Ruby or JavaScript implicitly create the context and therefore allow a direct yet more coarse-grained access to polyglot features. In case some code of the guest language is evaluated through the context, the host language receives a very generic polyglot value, in Java typed as *Value*. During the introduction of GraalVM, a very high-level explanation of how languages interact with each other through language agnostic messages was presented in the section 4.2.3. The polyglot value that is returned by the code evaluation provides access to these messages to enable the host language to use objects of the guest language. Therefore, to support interoperability, the guest language has to provide implementations for all appropriate messages of its internal objects.

Since Truffle languages are implemented in Java and can internally use the inbuilt Java datatypes if they are sufficient for the language, the manual addition of messages may not be necessary in all cases. The interoperability layer of GraalVM automatically implements messages for the the following datatypes:

- *String* and *Character* are interpreted as *string* values
- Boolean is interpreted as a *boolean* value
- *Byte*, *Short*, *Integer*, *Long*, *Float*, and *Double* are interpreted as *number* values

In all other cases, the language developer must create custom type classes for the new Truffle language, have the types implement the so called *TruffleObject* interface that is provided by the framework, and manually add implementations for all necessary messages.

The interop messages are grouped according to very generic type concepts which can be found in nearly all languages. In the previously presented list, a few of these types were already introduced: *strings*, *booleans*, and *numbers*. Furthermore, there are messages for *executable* objects such as functions or methods, *instantiable* objects such as classes or datastructures,*pointers*, *arrays*, and quite a lot more. To provide some examples, the following paragraphs will outline the changes that were executed to support interoperability for the *AMLNumber*, *AMLSet*, and *AMLCallable* objects.

As previously outlined, in case the implemented Truffle language uses the inbuilt Java types for simple datatypes such as numbers, booleans, or strings, no further work is necessary and interoperability is supported by default. Since AML uses a custom datatype for its numbers, however, the interoperability messages for numbers must be implemented manually. Listing 105 presents the additional code which is necessary to add support for interoperability to the *AMLNumber* class.

The first change is the addition of the *ExportLibrary* annotation to the top of the class. A *library* in Truffle provides a list of messages that a class of a language, in this case *AMLNumber*, can implement which the framework can then use for certain purposes. While other types of libraries exist in Truffle, the only interesting library for now is the *InteropLibrary* that provides all messages which a language element may implement for interoperability. Secondly, as

```java
@ExportLibrary(InteropLibrary.class)
public class AMLNumber implements TruffleObject, AMLObject {
    // Just as a reminder, AMLNumber is using a BigDecimal internally
    private final BigDecimal number;

    // ...

    @ExportMessage
    public boolean isNumber() { return fitsInLong(); }

    @ExportMessage
    public boolean fitsInByte() {
        return this.number.toBigInteger().bitLength() < 8;
    }

    // fitsInShort, fitsInInt, fitsInLong, fitsInFloat, fitsInDouble are similar

    @ExportMessage
    public byte asByte() throws UnsupportedMessageException {
        if (fitsInByte())
            return this.number.byteValue();
        else
            throw UnsupportedMessageException.create();
    }

    // asShort, asInt, asLong, asFloat, asDouble are similar

    @ExportMessage
    public Object toDisplayString(boolean allowSideEffects) {
        return number.toString();
    }
}
```

Listing 105: Adding interoperability support to the *AMLNumber* class.

previously outlined, all custom datatypes need to implement *TruffleObject* which is easily done since *TruffleObject* does not define any required methods. After having performed these two changes, the messages of the *InteropLibrary* can be added to the class with each one being annotated with *ExportMessage*. The name, return type, and parameters of these messages are all specified by the library and have to be exactly the same as defined in *InteropLibrary*. The first of these messages is the *isNumber* message. Truffle provides an *is<Type>* message for all of its generic types, for instance *isBoolean* or *isString*, which can be called by the host language to identify the returned type of an embedded expression. Afterwards, the *AMLNumber* class defines a set of *fitsIn<NumberType>* messages where each method returns a boolean that indicates whether the *BigDecimal* number of the instance could theoretically be converted to the corresponding number type. These checks are moreover utilised in the *as<NumberType>* messages which perform the actual conversion to the various number types. Both the *fitsIn<NumberType>* and the *as<NumberType>* messages can be utilised by host languages to transform the numbers of the guest language to a type of the host language. Lastly, every datatype in AML defines the *toDisplayString* message which can be considered as a language agnostic *toString* method for returned types of embedded expressions.

Through these messages, the *AMLNumber* datatype is ready to be utilised by other languages through a polyglot context of GraalVM. Listing 106 and 107 show how an expression returning *AMLNumber* could be executed in Java or Ruby through GraalVM.

```java
var ctx = Context.create("aml");
Value result = ctx.eval("aml", "1 + 1;");

if (result.isNumber() && result.fitsInLong()) {
    System.out.println(result.asLong()); // ⇒ 2
}
```

Listing 106: Executing an expression that returns *AMLNumber* in Java.

```ruby
value = Polyglot.eval('aml', '1 + 1;')

if value.respond_to?(:to_i)
  puts value.to_i # ⇒ 2
end
```

Listing 107: Executing an expression that returns *AMLNumber* in Ruby.

From the two listings, it is immediately noticable that the code, which is necessary to embed expressions of other languages, is largely dependent on the respective language. In Java, an explicit polyglot context has to be initially created with *aml* as its only permitted language. In Ruby, a context is implicitly created and code can be immediately executed through a *Polyglot* module. In Java, messages are executed by using methods of the polyglot value object with identical names as the corresponding messages. In Ruby, however, the expression `value.respond_to?(:to_i)` executes the messages to check whether the returned value can be transformed into an integer and is therefore not related to the original naming of messages. By using such an individual approach, the polyglot capabilities feel native in every language, since each language can decide how they are syntactically represented.

Another AML datatype with different semantics than scalar datatypes, such as booleans, numbers, or strings, is the *AMLSet*. The current abstract concept regarding interoperability messages for sequential datastructures is an array. List or set datastructures in Truffle languages therefore have to be made available to other host languages through simple array primitives and must be laid out as such. Listing 108 presents some of the array messages which are implemented for the *AMLSet* class.

Again, as with *AMLNumber*, the *ExportLibrary* annotation and the addition of *TruffleObject* to the list of implemented interfaces is necessary before defining any messages. The *hasArrayElements* message provides host languages with the information that this datatype can be interpreted as an array-like datastructure. All the other messages enable host languages to access elements in the set or retrieve general information about the set such as the size. Although messages to modify array elements are also available, AML does not make use of them due to its immutability.

```java
@ExportLibrary(InteropLibrary.class)
public class AMLSet implements TruffleObject, AMLObject {
    // Sets in AML are stored as (ordered) linked hash sets
    private final LinkedHashSet<AMLObject> set;
    // ...

    @ExportMessage
    public boolean hasArrayElements() { return true; }

    @ExportMessage
    public Object readArrayElement(long index) {
        return set.toArray()[(int) index];
    }

    @ExportMessage
    public long getArraySize() { return set.size(); }

    @ExportMessage
    public boolean isArrayElementReadable(long index) {
        return index < this.getArraySize();
    }
}
```

Listing 108: Array messages for the *AMLSet* datatype.

```java
Value result = ctx.eval(
    "aml",
    "even: (x) → x mod 2 = 0;" +
    "{ x ∈ {1, ..., 10} | even(x) };"
);

if (result.hasArrayElements()) {
    System.out.println(result.getArrayElement(1).asInt()); // ⇒ 4
}
```

Listing 109: Usage of *AMLSet* objects in Java.

```ruby
value = Polyglot.eval(
  'aml',
  'even: (x) → x mod 2 = 0;' +
  '{ x ∈ {1, ..., 10} | even(x) };'
)

if value.respond_to?(:size)
  puts value[1].to_i # ⇒ 4
end
```

Listing 110: Usage of *AMLSet* objects in Ruby

The usage of the *AMLSet* datatype using Java in listing 109 is very similar as it was with the *AMLNumber* type. The messages which were previously outlined are directly accessible through methods of the returned polyglot value object. With Ruby in listing 110, the case is different however. Processing array-like datastructures from other languages is totally independent regarding its syntax and closely resembles the use of native Ruby arrays.

```java
@ExportLibrary(InteropLibrary.class)
public class AMLNamedFunction extends AMLLambda implements TruffleObject {
    // ...

    @ExportMessage
    public boolean isExecutable() {
        return true;
    }

    @ExportMessage
    public Object execute(Object[] arguments) {
        if (arguments.length != arity())
            throw ArityException.create(arity(), arguments.length);

        List<AMLObject> args = new ArrayList<>();
        for (var arg : arguments)
            args.add(HostToAMLConverter.convert(arg));

        return invoke(args.toArray(Object[]::new));
    }
}

public class HostToAMLConverter {
    public static AMLObject convert(Object argument) {
        var library = InteropLibrary.getUncached();

        if (library.isBoolean(argument)) {
            return AMLBoolean.of(library.asBoolean(argument));
        } else if (library.isNumber(argument)) {
            return AMLNumber.of(BigDecimal.valueOf(library.asDouble(argument)));
        } else if (library.hasArrayElements(argument)) {
            List<AMLObject> list = new ArrayList<>();

            for (int i = 0; i < library.getArraySize(argument); i++)
                list.add(convert(library.readArrayElement(argument, i)));

            return AMLSet.of(list.toArray(AMLObject[]::new));
        }

        throw UnsupportedTypeException.create(
            new Object[] { argument }, argument.getClass() + " is not supported"
        );
    }
}
```

Listing 111: Interoperable *executable* objects using *AMLNamedFunction* as an example.

The last datatype that will be discussed is the *AMLCallable*, or when using interoperability terms: the *executable* object. Because the *ExportLibrary* annotation is not currently supported for interfaces, each individual implementation of *AMLCallable* has to define the messages related to executable objects, instead of defining them directly in the *AMLCallable* interface. Two messages are used for the callable objects of AML: *isExecutable* and *execute*. The latter message is the most interesting one since it allows host languages to execute functions which were defined in AML. Due to this ability, however, the *execute* message has also a special requirement: it needs to transform arguments that were given by the host language to AML datatypes. Luckily, the interoperability library can be utilised in both directions. Similar to how a host language can access types of a guest language through messages, the guest language can access objects from the host language by using messages as well. Listing 111 presents how interoperable executable objects are defined by using the *AMLNamedFunction* class as an example.

The listing presents two classes at once. In the first class, *AMLNamedFunction*, the implementation for the two messages for executable objects, *isExecutable* and *execute*, is provided. In the *execute* message, the array of arguments which consists of generic objects is individually passed to a utility method to convert it into an datatype which AML can process. Inside this utility method, an instance of *InteropLibrary* is acquired, with which the generic objects can be tested to determine its underlying semantics. Whenever the generic interoperable datatype is determined in the individual if statements, the object can be converted into an AML object and is returned. Through this mechanism, every Truffle language which exports messages for its own datatypes can execute functions which are written in AML.

```java
Value result = ctx.eval(
    "aml",
    "even: (x) → x mod 2 = 0;"
);

if (result.canExecute()) {
    System.out.println(result.execute(5)); // ⇒ ⊥
}
```

Listing 112: Execution of a named function of AML with Java.

```ruby
value = Polyglot.eval(
  'aml',
  'even: (x) → x mod 2 = 0;'
)

puts value.call(1) == true # ⇒ false
```

Listing 113: Execution of a named function of AML with Ruby.

Listing 112 and 113 present two examples for the execution of AML functions in other Truffle languages, here again in Java and Ruby. Curiously, in case of the Java example, the GraalVM developers have decided to execute the *isExecutable* message through the *canExecute* method of the polyglot value instead of calling it *isExecutable* as was done with the previous datatypes.

During the implementation of AML, two strategies regarding the deployment of DSLs that are written with Truffle were ascertained. As outlined in the initial section of this chapter, Truffle languages are generally compiled and bundled as non-executable JAR files. Using this JAR file, a developer can choose to either manually add the language to the class path of an application, or to produce a so called *component* which can be added to GraalVM using the Graal Component Updater.

For the former approach, the startup options of an application using such a DSL would have to be tweaked so the necessary functionalities of the DSL are correctly resolved. Listing 114 presents the necessary flags which are required by GraalVM for an application using Java as a host language.

```
MAIN_APP_PATH="path/to/main/app.jar"
DSL_PATH="path/to/dsl.jar"

$GRAAL_HOME/bin/java -Dtruffle.class.path.append="$DSL_PATH" \
  -cp "$MAIN_APP_PATH":"$DSL_PATH" \
  org.example.app.Main
```

Listing 114: Necessary flags for a Java application using a DSL implemented in Truffle.

Through the `-Dtruffle.class.path.append` flag, all necessary language classes can be identified by Truffle and can be made available to other languages through the interoperability features. The advantage of this manual approach is the flexible replacement of the DSL with a newer version of it. After overwriting the existing JAR file of the DSL and restarting the application, the latest version of the DSL is immediately utilised by the host language. Especially during the development of the DSL where frequent updates occur, the ability to test these updates quickly makes this approach feasible. Since the language is only available by adding these necessary flags, however, it is not available by default for all installed GraalVM languages. When running utility programs of Truffle languages such as a Ruby or Python *read-eval-print loop (REPL)* where the addition of the presented flags are not available, the respective DSL will not be available as well.

This problem on the other hand is solved by the approach of building a component and installing it locally. To build it, the GraalVM team provides an example for a Truffle language implementation which includes a small shell script[17] presenting all the steps for creating GraalVM components. This script can be modified according to one's purposes and can then be executed to retrieve a new and modified language JAR file. Through the Graal Component Updater the JAR file can then be installed in GraalVM as shown in listing 115.

```
$ $GRAAL_HOME/bin/gu install -L aml-component.jar
```

Listing 115: Install a language component (here AML) using the Graal Component Updater.

[17]https://raw.githubusercontent.com/graalvm/simplelanguage/master/component/make_component.sh

After the installation, every locally available GraalVM language can use the DSL through their respective polyglot capabilities without having to manually add or configure the DSL using flags. However, the downside of this approach is the necessity of rebuilding and reinstalling the complete component each time the DSL is modified.

Therefore, while the first approach could be more feasible during the development of the DSL, the latter approach is superior after the DSL has become stable and is not updated frequently anymore.

6 Discussion and Future Work

7 Conclusion

Business as usual

References

[1] Martijn Dwars. Graal vs. c2: Battle of the jits. https://martijndwars.nl/2020/02/24/graal-vs-c2.html. Accessed: February 6, 2021.

[2] Oliver B. Fischer. An introduction to graalvm, oracles new virtual machine. https://jaxlondon.com/blog/an-introduction-to-graalvm-oracles-new-virtual-machine/. Accessed: February 1, 2021.

[3] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[4] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 855–865, 2006.

[5] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 78–90, 2015.

[6] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[7] Maurice Kherlakian. What is a foreign function interface (ffi) in php? https://www.zend.com/blog/php-foreign-function-interface-ffi. Accessed: March 22, 2021.

[8] Michael Larabel. Openjdk 8/11 vs. graalvm 20 vs. amazon corretto jvm benchmarks. https://www.phoronix.com/scan.php?page=article&item=openjdk-corretto-graalvm&num=1. Accessed: February 6, 2021.

[9] Scott Lynn. For building programs that run faster anywhere: Oracle graalvm enterprise edition. https://blogs.oracle.com/graalvm/announcement. Accessed: January 19, 2021.

[10] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[11] Bob Nystrom. Crafting interpreters. http://craftinginterpreters.com/. Accessed: January 15, 2021.

[12] Oracle. Graalvm homepage. https://www.graalvm.org/. Accessed: February 1, 2021.

[13] Tobias Pfeiffer. Benchmarking a go ai in ruby: Cruby vs. rubinius vs. jruby vs. truffle a year later. https://pragtob.wordpress.com/2017/01/24/benchmarking-a-go-ai-in-ruby-cruby-vs-rubinius-vs-jruby-vs-truffle-a-year-later/. Accessed: January 19, 2021.

[14] Philip Riecks. Graalvm an introduction to the next level jvm. https://rieckpil.de/whatis-graalvm/. Accessed: February 2, 2021.

[15] Chris Seaton. Understanding how graal works - a java jit compiler written in java. https://chrisseaton.com/truffleruby/jokerconf17/. Accessed: February 1, 2021.

[16] Leonard J. Shustek. The apl programming language source code. https://computerhistory.org/blog/the-apl-programming-language-source-code/. Accessed: March 16, 2021.

[17] Goparaju Purna Sudhakar. A model of critical success factors for software projects. *Journal of Enterprise Information Management*, 2012.

[18] Karine Vardanyan and Stephan Rauh. Eine jvm für die cloud: die graalvm. https://www.opitz-consulting.com/sites/default/files/Kompetenz/Fachartikel/PDF/java-aktuell-2020-01_Eine-JVM-fuer-die-Cloud-die-GraalVM_Vardanyan-Rauh_sicher.pdf. Accessed: February 1, 2021.

[19] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering*, pages 291–373. Springer, 2007.

[20] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, Guido Wachsmuth, et al. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[21] Thomas Wuerthinger. Graalvm with thomas wuerthinger. https://softwareengineeringdaily.com/wp-content/uploads/2018/08/SED644-GraalVM.pdf. Accessed: February 1, 2021.

[22] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[23] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 73–82, 2012.

A Completion of Code Listings

A.1 Internal Timer DSL

```
public final class RepeatableTimerExpressionBuilder {
    private final TimerTask task;

    public RepeatableTimerExpressionBuilder(TimerTask task) {
        this.task = task;
    }

    public PeriodicRepeatableTimerExpressionBuilder every(long period) {
        return new PeriodicRepeatableTimerExpressionBuilder(this.task, period);
    }
}

public final class PeriodicRepeatableTimerExpressionBuilder {
    private final TimerTask task;
    private final long period;

    public PeriodicRepeatableTimerExpressionBuilder(TimerTask task, long period) {
        this.task = task;
        this.period = period;
    }

    public FinalizedRepeatableTimerExpressionBuilder rightNow() {
        return after(0);
    }

    public FinalizedRepeatableTimerExpressionBuilder after(long delay) {
        return new FinalizedRepeatableTimerExpressionBuilder(this.task, this.period, delay);
    }
}

public final class FinalizedRepeatableTimerExpressionBuilder {
    private final TimerTask task;
    private final long period;
    private final long delay;

    public FinalizedRepeatableTimerExpressionBuilder(TimerTask task, long period, long delay) {
        this.task = task;
        this.period = period;
        this.delay = delay;
    }

    public void setup() {
        var timer = new Timer();
        timer.schedule(this.task, this.delay, this.period);
    }
}
```

Listing 116: All remaining classes to define a periodic timer task.

```
public static class SingleTimerExpressionBuilder {
    private final TimerTask task;

    public SingleTimerExpressionBuilder(TimerTask task) {
        this.task = task;
    }

    public FinalizedSingleTimerExpressionBuilder rightNow() {
        return after(0);
    }

    public FinalizedSingleTimerExpressionBuilder after(long delay) {
        return new FinalizedSingleTimerExpressionBuilder(this.task, delay);
    }
}

public static class FinalizedSingleTimerExpressionBuilder {
    private final TimerTask task;
    private final long delay;

    public FinalizedSingleTimerExpressionBuilder(TimerTask task, long delay) {
        this.task = task;
        this.delay = delay;
    }

    public void setup() {
        var timer = new Timer();
        timer.schedule(this.task, delay);
    }
}
```

Listing 117: All remaining classes to define a single timer task.

A.2 External Timer DSL

```
public class Lexer {
    private static final Map<String, TokenType> KEYWORDS = new HashMap<>();
    private int startOfToken = 0;
    private int endOfToken = 0;
    private final String code;
    private final List<Token> tokens = new ArrayList<>();

    static {
        KEYWORDS.putAll(Map.of(
                                "timer", TIMER, "print", PRINT, "repeatedly", REPEATEDLY, "once", ONC
                                "every", EVERY, "after", AFTER, "seconds", SECONDS, "minutes", MINUTE
                                "hours", HOURS, "right", RIGHT
                                ));
        KEYWORDS.putAll(Map.of(
                                "now", NOW, "end", END
                                ));
    }

    public Lexer(String code) {
        this.code = code;
    }

    public List<Token> getTokens() throws TimerDSLException {
        while (!isAtEnd()) {
            readNextToken();
            this.startOfToken = this.endOfToken + 1;
            this.endOfToken = this.startOfToken;
        }

        tokens.add(new Token(EOF, null));
        return tokens;
    }

    private void readNextToken() throws TimerDSLException {
        var nextChar = code.charAt(this.startOfToken);

        if (List.of(' ', '\r', '\t', '\n').contains(nextChar)) {
            // do nothing
        } else if ('"' == nextChar) {
            string();
        } else if (isDigit(nextChar)) {
            number();
        } else if (isAlpha(nextChar)) {
            keyword();
        } else {
            throw new TimerDSLException("Unexpected character");
        }
    }

    // Continues on the next page
```

Listing 118: The whole lexer class of the external timer scheduling DSL.

```java
private void string() throws TimerDSLException {
    endOfToken++;
    while (peek() != '"' && !isAtEnd()) endOfToken++;

    if (isAtEnd()) throw new TimerDSLException("Unterminated string");

    endOfToken++;
    var value = code.substring(startOfToken + 1, endOfToken - 1);
    tokens.add(new Token(STRING, value));
}

private void number() {
    while (isDigit(peek())) endOfToken++;
    tokens.add(
            new Token(NUMBER, Integer.parseInt(code.substring(startOfToken, endOfToken)))
            );
}

private void keyword() throws TimerDSLException {
    while (isAlpha(peek())) endOfToken++;
    var text = code.substring(startOfToken, endOfToken);

    if (KEYWORDS.containsKey(text))
        tokens.add(new Token(KEYWORDS.get(text), null));
    else
        throw new TimerDSLException("Unexpected keyword.");
}

private char peek() {
    return code.charAt(endOfToken);
}

private boolean isDigit(char c) {
    return c >= '0' && c <= '9';
}

private boolean isAlpha(char c) {
    return c >= 'a' && c <= 'z';
}

private boolean isAtEnd() {
    return startOfToken >= code.length();
}
}
```

Listing 119: The whole lexer class of the external timer scheduling DSL (continuation).

```java
public abstract class TimerConfiguration {
    public enum TimeUnit {
        SECONDS, MINUTES, HOURS
    }

    public static class OnceTimer extends TimerConfiguration {
        private final TimeSetting afterSetting;

        public OnceTimer(TimeSetting afterSetting) {
            this.afterSetting = afterSetting;
        }

        public TimeSetting getAfterSetting() {
            return afterSetting;
        }
    }

    public static class RepeatedTimer extends TimerConfiguration {
        private final TimeSetting everySetting;
        private final TimeSetting afterSetting;

        public RepeatedTimer(TimeSetting everySetting, TimeSetting afterSetting) {
            this.everySetting = everySetting;
            this.afterSetting = afterSetting;
        }

        public TimeSetting getEverySetting() {
            return everySetting;
        }

        public TimeSetting getAfterSetting() {
            return afterSetting;
        }
    }

    public static class TimeSetting {
        private final long number;
        private final TimeUnit unit;

        public TimeSetting(long number, TimeUnit unit) {
            this.number = number;
            this.unit = unit;
        }

        public long getNumber() {
            return number;
        }

        public TimeUnit getUnit() {
            return unit;
        }
    }
}
```

Listing 120: The timer configuration classes of the AST.

```java
public class Parser {
    private final List<Token> tokens;
    private int current;

    public Parser(List<Token> tokens) {
        this.tokens = tokens;
        this.current = 0;
    }

    // program → timer_stmt+
    public List<TimerStmt> parse() throws TimerDSLException {
        var timerStatements = new ArrayList<TimerStmt>();
        timerStatements.add(timerStmt());

        while (!isAtEnd()) {
            timerStatements.add(timerStmt());
        }

        return timerStatements;
    }

    // timer_stmt → "timer" command (once_timer | repeated_timer) "end"
    private TimerStmt timerStmt() throws TimerDSLException {
        consume(TIMER, "Expected 'timer' at the beginning of definition.");

        var command = command();

        TimerConfiguration config = null;

        if (match(ONCE)) config = onceTimer();
        else {
            consume(REPEATEDLY, "Expected 'once' or 'repeatedly' after command.");
            config = repeatedTimer();
        }

        consume(END, "Expected 'end' at the end of definition.");

        return new TimerStmt(command, config);
    }

    // command → "print" STRING
    private Command command() throws TimerDSLException {
        consume(PRINT, "Expected 'print' command.");
        var message = consume(STRING, "Expected 'string' after 'print'.");
        return new PrintCommand((String) message.getValue());
    }

    // once_timer → "once" after_configuration
    private OnceTimer onceTimer() throws TimerDSLException {
        current++;
        return new OnceTimer(afterConfig());
    }

    // Continues on the next page
```

Listing 121: The complete recursive descent parser.

```
// repeated_timer → "repeatedly" "every" NUMBER time_unit after_configuration
private RepeatedTimer repeatedTimer() throws TimerDSLException {
    consume(EVERY, "Expected 'every' after 'repeatedly'.");
    var number = consume(NUMBER, "Expected 'number' after 'every'.");
    return new RepeatedTimer(
                             new TimeSetting(Long.valueOf((Integer) number.getValue()), timeUnit()),
                             afterConfig()
                             );
}

// after_configuration → "right" "now" | "after" NUMBER time_unit
private TimeSetting afterConfig() throws TimerDSLException {
    if (match(RIGHT)) {
        current++;
        consume(NOW, "Expected 'now' after 'right'.");
        return new TimeSetting(0, TimeUnit.SECONDS);
    } else {
        consume(AFTER, "Expected 'right now' or 'after' as a time setting.");
        var number = consume(NUMBER, "Expected 'number' after 'after'.");
        return new TimeSetting(Long.valueOf((Integer) number.getValue()), timeUnit());
    }
}

// time_unit → "seconds" | "minutes" | "hours"
private TimeUnit timeUnit() throws TimerDSLException {
    if (match(SECONDS)) {
        current++;
        return TimeUnit.SECONDS;
    } else if (match(MINUTES)) {
        current++;
        return TimeUnit.MINUTES;
    } else {
        consume(HOURS, "Expected 'minutes', 'seconds', or 'hours' as time unit.");
        return TimeUnit.HOURS;
    }
}

private Token consume(TokenType type, String message) throws TimerDSLException {
    if (match(type)) {
        current++;
        return tokens.get(current-1);
    }

    throw new TimerDSLException(message);
}

private boolean match(TokenType type) {
    return tokens.get(current).getType() == type;
}

private boolean isAtEnd() {
    return tokens.get(current).getType() == EOF;
}
}
```

Listing 122: The complete recursive descent parser (Continuation).

B AML Language Grammar

```
program          → (function | expression)+
    function         → IDENTIFIER ":" "(" params ")" "→" expression
    params           → IDENTIFIER? | (IDENTIFIER ",")+ IDENTIFIER
    expression       → (ifcond | assignment) ";"
    ifcond           → "if" logicEquivalence ":" thenBranch "otherwise" ":" elseBranch
    thenBranch       → (ifcond | composition | logicEquivalence)
    elseBranch       → (ifcond | composition | logicEquivalence)
    assignment       → (IDENTIFIER "←" assignment) | composition | logicEquivalence
    composition      → IDENTIFIER "∘" IDENTIFIER
    logicEquivalence → logicImplication ("⇔" logicImplication)*
    logicImplication → logicOr ("⇒" logicOr)*
    logicOr          → logicXOr ("∨" logicXOr)*
    logicXOr         → logicAnd ("⊕" logicAnd)*
    logicAnd         → equality ("∧" equality)*
    equality         → negation (("=" | "≠") negation)*
    negation         → comparison | ("¬" comparison)
    comparison       → quantification | numComparison | setComparison

    quantification   → universal | existential | uniqueness
    universal        → "∀" "(" IDENTIFIER "∈" setPrimary ":" logicEquivalence ")"
    existential      → "∃" "(" IDENTIFIER "∈" setPrimary ":" logicEquivalence ")"
    uniqueness       → "∃!" "(" IDENTIFIER "∈" setPrimary ":" logicEquivalence ")"

// Numeric rules
    numComparison    → term (("<" | ">" | "≤" | "≥") term)*
    term             → factor (("+" | "-") factor)*
    factor           → exponentiation (("·" | "÷" | "mod") exponentiation)*
    exponentiation   → fraction ("^" fraction)*
    fraction         → numUnary ("\\" numUnary)?
    numUnary         → numNegation | factorial | floor | ceil | numPrimary
    numNegation      → "-" numPrimary
    factorial        → numPrimary "!"
    floor            → "⌊" numPrimary "⌋"
    ceil             → "⌈" numPrimary "⌉"
    numPrimary       → call | number | IDENTIFIER | "(" logicEquivalence ")"
    number           → NUMBER ("." NUMBER)?

// Set rules
    setComparison    → setOperations (("⊂" | "⊄" | "⊆" | "⊈" | "⊃" | "⊅" | "⊇" | "⊉")
                                      setOperations)*
    setOperations    → setUnary (("∪" | "∩" | "\\") setUnary)*
    setUnary         → cardinality | setPrimary
    cardinality      → "|" setPrimary "|"
    setPrimary       → call | setLiteral | setEllipsis | setBuilder |
    IDENTIFIER | "(" logicEquivalence ")"
    setBuilder       → "{" IDENTIFIER "∈" setPrimary "|" logicEquivalence "}"
    setLiteral       → "{" ((logicEquivalence ",")* logicEquivalence)? "}"
    setEllipsis      → "{" numUnary "," "..." "," numUnary "}"

    call             → IDENTIFIER "(" arguments ")"
    arguments        → logicEquivalence? | (logicEquivalence ",")+ logicEquivalence
```

Listing 123: The complete language grammar of AML.