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: January 28, 2021

Thesis Declaration

Abstract

Table of Contents

1 Introduction

In May 2019, Oracle published an announcement [3] 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.

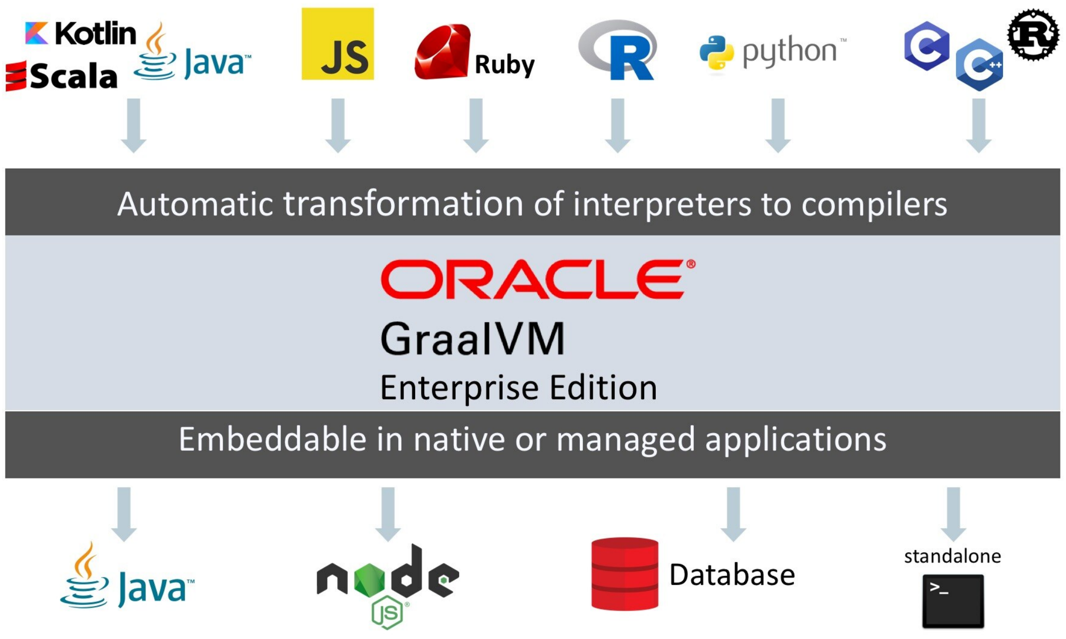
Kotlin
Scala
Java
JS
Ruby
python
Automatic transformation of interpreters to compilers
ORACLE
GraalVM
Enterprise Edition
Embeddable in native or managed applications
Java
node
Database
standalone

Figure 1: High-level architecture of GraalVM [3].

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 [6].

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 popular

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

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 [1]. 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 <INSERT NAME HERE>. 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 <INSERT NAME HERE>. 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, <INSERT NAME HERE>, 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 <INSERT NAME HERE> 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 [4] and examples for DSLs range back to at least 1957 where a DSL called *APT* to program numerically-controlled machines was developed [4]. Eventhough there is no standardized definition of the term *domain specific language*, the definitions presented by different authors are very similar:

- Martin Fowler [1] defines DSLs as follows: "a computer programming language of limited expressiveness focused on a particular domain."
- For Markus Voelter et al. [9], 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 [8] 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 [1] 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 [4]) 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. [4] 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 [1] 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. [9]

	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 [9]

```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.

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 [1], where the distinction between DSL an general-purpose language is not as clear, is the *R language*[1]. R is a programming language for statistical computing and is therefore generally focused on a particular domain. Despite 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 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 more easy and faster to write, read, and understand due to the fact that less code is necessary to solve a problem [9]. Furthermore, through its limited expressiveness DSLs are much more restrictive which leads to both making less mistakes as well as fixing defects more quickly [1]. Voelter et al. [9] 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 [7]), 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 [1].
- **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 [9]. 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*[2] 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 [1], DSLs offer 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 [1] 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. [9] 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 [1], 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. [9], 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 [1] 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. [9] recommends businesses to keep an open mind and to throw things overboard, if necessary.
- **Blinkered Abstraction:** Another problem Martin Fowler [1] 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

Different approaches have been presented in the past to develop DSLs. One of these was posed by Mernik et al. [4] 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. [4] 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. [4] 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. [9] 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. [4] which is very similar to the traditional waterfall model, Voelter et al. [9] 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 4 and 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 [1] 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 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

```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.

```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.

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 26 and 27 of the appendix.

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

```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.

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 [2]) 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 [5], 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* [5] was employed. The book uses widespread techniques to build languages which are also highlighted in Fowler's work about DSLs [1]. 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.

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

```
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.

as the end position, the source code itself, and the list of tokens which will be returned in the end.

```
  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.

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.

```java
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.

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.

```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));
}
```

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

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 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 28 and 29 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 [5].

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

```
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

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.

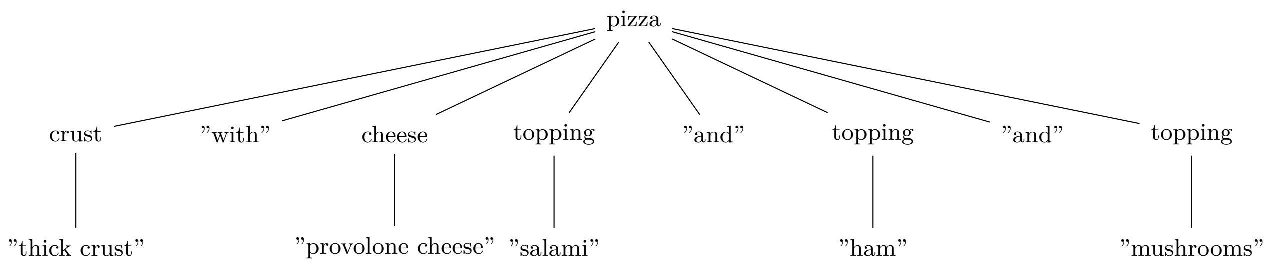

Figure 2: *thick crust with provolone cheese and salami and ham and mushrooms* represented as an AST

The main task of the parser in the process presented by Nystrom [5] 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.

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*

```
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.

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.

```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.

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 30 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.

```
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.

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 concrete code can actually be constructed. To address this problem, Bob Nystrom presents a popular technique in his work [5] 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 [5].

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.

```
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.

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 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.

```
// 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.

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

```
// 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.

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 be consumed and the whole timer statement is returned.

```java
// 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 31 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.

```java
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.

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.

```
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.

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 *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 pattern* in his work [5] 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)[1] or *ANTLR* (ANother Tool for Language Recognition)[2] 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

```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.

of this thesis will be based on.

3 Overview of GraalVM

3.1 Motivation

Why do we need Graal?
Write more of Java in Java itself.

3.2 Features

3.2.1 GraalVM Compiler

Explanation and some benchmarks
source: `https://www.youtube.com/watch?v=sFf15TvSXZ0`

1. What is a JIT compiler

   When compile Java Source with javac –> Java Bytecode At Runtime –> Bytecode is compiled in Machine Code Machine Code delivers usually much better perf

2. Why write JIT compiler in Java

   C2 is the JIT compiler written in C++ Developers of JIT think C2 is too old now, too hard to maintain Developing in Java tends to be easier and more productive than in C++

3. JVM compiler interface

   Allows to plugin a custom JIT compiler for the JVM written in Java In thesis interface can shown here: `https://github.com/openjdk/jdk/blob/master/src/jdk.internal.vm.ci/share/classes/jdk.vm.ci.runtime/src/jdk/vm/ci/runtime/JVMCICompiler.java` Takes bytecode and returns new bytecode

4. Graal JIT compiler process

   The compiler first represents code in graphs Every node will then be transformed into machine code

5. Optimisations

   - Basically changes the nodes
   - Canonicalisation: e.g. –x –> x
   - Global value numbering: remove redundant code: (a + b) * (a + b) -> only (a + b) once calculated
   - Lock coarsening: two synchronized locks immediately after each other -> change to only once

3.2.2 Native Images

Explanation and some benchmarks yet again

3.2.3 Truffle Framework

Basic explanations: why is there a truffle framework and what is achievable

3.2.4 Polyglot Applications

Basic explanations: what's possible here

4 Domain Specific Languages in GraalVM

4.1 Technical Overview

How to build DSLs with GraalVM?

4.2 <INSERT NAME OF DSL>

Introduce the DSL of this thesis here 1 2 3 4 5

$$A = \{ x \mid x \in (A \cap B) \} \tag{1}$$

4.3 Implementation of <INSERT NAME OF DSL>

Highlight some key aspects of implementation

4.4 Evaluation

Evaluate the DSL and GraalVM, highlight pain points etc.

5 Integration of Domain Specific Languages

5.1 Technical Overview

How do polyglot applications technically work?

5.2 Integration of <INSERT NAME OF DSL>

Showcase how it's done using the thesis DSL

5.3 Evaluation

Evaluation how good this actually works

6 Conclusion

Business as usual

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

A Completion of Code Listings

A.1 Internal Timer DSL

```java
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 26: All remaining classes to define a periodic timer task.

```java
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 27: 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", ONCE,
            "every", EVERY, "after", AFTER, "seconds", SECONDS, "minutes", MINUTES,
            "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 28: The whole lexer class of the external timer scheduling DSL.

```
    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 29: 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 30: The timer configuration classes of the AST.

```
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 31: 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 32: The complete recursive descent parser (Continuation).