JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Franz Miketta**

Submitted at
**Institute for System Software**

Supervisor
**DI Raphael Mosaner**

Co-Supervisor
**Dr. Christian Wirth**
**Dr. Daniele Bonetta**

July 1, 2021

# CONTRIBUTING THE ECMASCRIPT PROPOSAL “MODULE BLOCKS” TO GRAAL.JS

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor’s Program

Informatik

**JOHANNES KEPLER UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

# Statutory Declaration

I hereby declare under oath that the submitted thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, July 1, 2021

JOHANNES KEPLER
UNIVERSITY LINZ

# Abstract

The Graal.Js is an ECMAScript runtime engine and thus competes with engines the like of V8 by Google or SpiderMonkey by Mozilla. Although Graal.Js' environment, the GraalVM, provides the project vital vantages, the engine needs to stay ahead of the ECMAScript specification to offer full feature support. This is the entry point for this thesis' purpose, namely to implement an ECMAScript proposal from the TC9 proposals, in particular module blocks, in the existing Graal.Js project. Besides the initial task of changing the parser and the runtime, and conducting unit tests, a base framework for shipping code between processes is implemented. It should be declared for the avoidance of doubt that the code shipping framework is not part of the proposal but rather an addition to make it more feasible in practice.

The thesis starts with a theoretical briefing on the technologies in which the implementation resides in, i.e. the Java Virtual Machine, the GraalVM, the truffle implementation framework, Graal.Js and ECMAScript. Later on the proposal implementation and its integration into Graal.Js is explained in a brief way. Since the proposal and the base framework are connected by proximity of topic the framework's implementation also resides in Chapter 3. The implementation is followed up by unit tests which are mainly self-written but mixed with code snippets from the proposal's website to prove expected behavior. The thesis is then wrapped up with an outlook on future work and the thesis' conclusions. The thesis practical work, i.e. the implementation, is finished by contributing the final artifact to the open source version of Graal.Js on the platform github.com.

# Zusammenfassung

Das Graal.Js-Projekt umfasst eine Laufzeitumgebung für Skriptsprachen, die den ECMAScript-Standard implementieren. Aufgrund dieser Thematik muss sich die Graal.Js-engine mit anderen Laufzeitumgebungen wie der V8 von Google oder SpiderMonkey von Mozilla messen. Das Graal.Js-Projekt hat hierbei den Vorteil des gesamten GraalVM-Ökosystems auf seiner Seite. Ein zentraler

Punkt, um sich gegen Wettbewerber durchzusetzen bleibt allerdings der Featuresupport. Um diesen vollumfänglich gewähren zu können, ist es notwendig, dass Features, die noch nicht im ECMAScript Standard integriert wurden, bei denen allerdings davon ausgegangen werden kann, in der Laufzeitumgebung zu implementieren. Dieser Featuresupport bildet die Grundlage für die vorliegende Arbeit, deren Ziel es ist einen ECMAScript-Erweiterungsvorschlag aus den TC39 Erweiterungsvorschlägen, die sogenannten module blocks, im bestehenden Graal.Js-Interpreter zu implementieren. Nach der Implementierung im Parser und der Runtime und entsprechenden Tests wird weiters ein Framework zum Übertragen von Code zwischen Prozessen eingerichtet. Das Framework selbst ist nicht Teil des Proposals, sondern eine praktische Erweiterung für das Projekt selbst.

Die vorliegende Arbeit beginnt mit einer theoretischen Einführung in die beteiligten Technologien. Diese umfassen die Java Virtual Machine, die GraalVM, das Truffle Implementierungsframework, Graal.Js und ECMAScript. Folgend wird die Implementierung des Erweiterungsvorschlags und dessen Integration in das Graal.Js-Projekt erläutert. Im selben Kapitel wird aufgrund der thematischen Nähe auch das Übertragungsframework beschrieben. Danach werden die durchgeführten Tests gezeigt. Diese umfassen selbst geschriebene Tests und Tests, die von der Webseite des Erweiterungsvorschlags übernommen wurden, um erwartetes Verhalten zu prüfen. Die Arbeit wird mit einem Ausblick auf zukünftige Arbeit und einem Fazit abgeschlossen. Der praktische Teil der Arbeit findet seinen Abschluss durch Einbringen des fertigen Codes in die Open Source-Version von Graal.Js auf der Plattform github.com.

JOHANNES KEPLER
UNIVERSITY LINZ

# Contents

# 1 Introduction

This chapter serves as an introduction to the thesis by explaining the relevance for *ECMAScript*, *GraalVM* and *Graal.js* in general and in the specific case of the ECMAScript proposal *module blocks*. The chapter is wrapped up by an outline for the thesis.

ECMAScript [11] is the standardized version of the famous internet front-end scripting language *JavaScript*. The ECMAScript specification includes language features and their expected behavior each scripting language should have. The specification then in turn is implemented by so-called engines that run JavaScript code. JavaScript is on spot three on the PYPL popularity index for programming languages indicating its popularity by google search trends. [1] As a core technology of the internet it helped shaping the web as we see it today. An important part in the language becoming a core technology was browser support which had been a problem in the past. The particular support problem has been interoperability, i.e. where websites had to be programmed differently for different browsers due to their differences in JavaScript engine implementation. [26] These variations in turn then led to peculiar appearances or in the worst case non-working websites. That meant a huge programming overhead which wasn't feasible in the long run. This issue was fixed by ECMAScript. Meanwhile a browser's engine's support is determined by feature support of the ECMAScript specification. The language specification development doesn't stagnate but is amended via a proposal process which is divided into five stages and is overlooked by an installed ECMA-committee, the so-called *TC39*. Thus, new language features bundled into versions are developed in a standardized environment with the web community and vendors together as a team.

In 2019 Oracle released the GraalVM with active development up to present days. GraalVM [6] is a *Java Virtual Machine (JVM)* with multiple core features such as certain compiler optimizations, ahead-of-time compilation, polyglot programming, making it a multilingual runtime, *Low Level Virtual Machine (LLVM)* runtime and the *Truffle* language implementation framework. With the amount of supported programming languages and the foregoing mentioned features the GraalVM can be implemented into a variety of production environments. One environment seems of particular interest: Microservices on server environments. The GraalVM native image was able to lower startup time and memory footprint by a significant amount. [9] These savings won over the social media platform Twitter whose Microservices run on GraalVM. [20] The engine in return created savings for their CPU times which makes it have an environmental impact. The other core feature of GraalVM is the support of multiple languages via the Truffle framework. With this framework

different languages can be implemented on top of GraalVM. [15] The framework itself provides a base for tree-based interpreter implementations. The different language implementations are split up into single projects for each language. This setup leads to the project Graal.js.

Graal.js is the Truffle implementation of ECMAScript on the GraalVM. The project implements the language specification with the Truffle framework. In essence, the project transforms JavaScript source code into an *abstract syntax tree (AST)* to be executed by any JVM. [8] The main components to be introduced for the task include a parser, nodes for the resulting abstract syntax tree and a transformation logic for translating the parser produced intermediate representation to the aforementioned abstract syntax tree. The Graal.js interpreter can be executed on any Java-compliant JVM and provides full support for ECMAScript. Although it can be run on any JVM, execution with GraalVM has significant performance benefits, as it allows the automatic transformation of the AST into highly optimized machine code. All in all the system is an ECMAScript engine and thus, has to compete with other ECMAScript engines. With the official release of GraalVM 21 it showed to be on par with V8, Google's engine, and Spidermonkey, Mozilla's engine, with all three supporting 99% of the newest ECMAScript version. [17] The big advantage Graal.js has over the other two engines is being embedded in the GraalVM ecosystem allowing polyglot programs. To keep up with current development the Graal.js project aims to implement proposal that haven't gone the whole way of the aforementioned proposal process and are yet to be released as new features of ECMAScript to be ahead of time. One of these yet to pass the process proposals is the module block proposal.

Module blocks are an effort by *Daniel Ehrenberg* [1] and *Surma* [2] based on inline modules. Inlining modules is a feature which is missing from the current ECMAScript. The absence has resulted into workarounds with various issues. ECMAScript cannot share code between processes thus residing on a single thread. Every *module*, *worker* and *worklet* needs a separate file cluttering project folders. A worker is basically opening a new context to run a module in a separate thread, i.e. the worker thread. [25] Tasks short of stringification cannot be shared across agents. The multitude of problems cited can be addressed by module blocks. Module blocks are a stage 2 proposal on ECMAScript 262 and is scheduled to be implemented in a future version of ECMAScript. [4] Since it has a high relevance for polyglot programs its implementation is key to staying on top of the technology stack which brings us to the purpose of this thesis.

The main objective of this thesis is to implement the new ECMAScript stage 2 proposal module blocks into the Graal.js engine which is implemented via the Truffle framework. Furthermore testing for this implementation will be conducted. When the general workings are set a code shipping framework and benchmarks will be included.

The thesis is divided into five further chapters. Chapter 2 explains the theoretical background of the thesis. In the following Chapter 3 the implementation and the testing is addressed. Afterwards,

[1] https://github.com/littledan
[2] https://github.com/surma

Chapter 4, evaluates the implementation with extensive tests. The thesis is then rounded up in Chapter 5 and 6 by highlighting future work and a conclusion.

# 2 Background

This chapter lays out the theoretical background regarding the thesis's topic by first introducing the general concepts of a *high-level language virtual machine* (HLL-VM) like the Java Virtual Machine (JVM) and then going on to more specific features of the GraalVM. In the following, the abstraction layer, the Truffle API is explained before coming to the explicit project Graal.js which is built on top of the Truffle language implementation framework. The last section discusses the ECMAScript parts of the topic with a brief glance at the current state of modules and what the proposal tries to achieve.

## 2.1 High-Level language virtual machines

A high-level language virtual machine (HLL-VM) is an abstraction layer relieving the programmer from several tasks with the main feature being platform independent development. When starting to discuss the platform independence it is foremost important to note why programs are usually platform bound. The main reason are two technologies which are presented in the next paragraph.

### 2.1.1 Groundwork

The following paragraphs serve as introduction into HLL-VMs and are based on the work of Smith in [22] where the core ideas are bundled. Every computer employs some kind of *instruction set architecture* (ISA) and an *operating system* (OS). Every developed program is bound to these two technologies. If a program is developed for a particular pair of ISA/OS it has to be ported to run on a machine with a different pair of ISA/OS. The result of this paradigm is support overhead since every ported version of an application has to be updated individually. It is thus highly impractical to enforce such a development environment for every application program. By developing a HLL-VM this task is posed upon the development of the VM only and all other application programs being developed don't have to focus on platform dependency resulting in a more lean development and update process. Hence the VM delivers an abstraction layer for the ISA/OS couples and the applications running on top of the VM. [22]

How is the VM delivering the abstraction layer? High-level language virtual machines enhanced the concept of early VMs like *P-code* [21] by using virtual instruction set architectures (V-ISA) encompassing code and metadata, like data structures and resource-related information, independently of platforms. The code is simply interpreted the metadata loaded and thus turned into a machine-dependent version by the virtual machines provided emulator. [22] This alone already stands out as a huge accomplishment but HLL-VMs come with even more features.

In today's computer landscape security risks can come from untrusted software run on machines. The HLL-VM, in particular the Java Virtual Machine (JVM), provides a metaphoric sandbox in which the untrusted application can run without making the rest of the system outside of the VM vulnerable. Although it amplifies security there are loopholes to bypass said security measures especially when untrusted software is given explicit permission to go outside of the VM's provided resources. [19, 22] Hence running untrusted software is still not advised but made more secure with a HLL-VM. As said before security is not the only feature HLL-VMs serve. A feature directly improving code production is delivering an environment being beneficial for robust code. The details of the particular environment are subject to the upcoming discussion.

Especially when it comes to large-scale software systems robust code is key. Here the good fit of the object-oriented model and the platform independence make HLL-VMs the top technology in the field. How much a HLL-VM supports robustness is of course depending on the used VM but on the example of the Java Virtual Machine strong type-checking and garbage collection which will be explained later lift a lot of beverage off the programmer's shoulder making the programmer concentrate on the mere implementation and thus making code production more robust automatically because the execution environment is robust within itself. Additional merits of HLL-VMs come from technologies like dynamic linking saving network bandwidth or profiling for performance.[22]

As the start of this subsection already stated a HLL-VM is an abstraction layer with a lot of features making a programmer's life considerably easier. This comes with some costs most notably performance. This cost is then highly reduced by certain techniques like the aforementioned profiling. With the groundwork laid out the next section discusses certain features of the Java Virtual Machine.

### 2.1.2 The Java Virtual Machine

The Java Virtual Machine (JVM) is, as its name states, developed around the language Java. Java is a general-purpose, object-oriented language with strong static types aimed as a production language. [7] Since one of the language's target is simplicity, details of machine representation are abstracted away from the user and not accessible through the language. This is in compliance with the points made in the HLL-VM chapter beforehand as the JVM serves as abstraction layer. First of the programmer does not have to worry about the ISA/OS-couple, secondly the aforementioned

garbage collection abstracts machine representation as well. Further safety measures have been made like automatic storage management and checked array access. The Java sourcecode is usually compiled ahead of time (AOT) to *Java bytecode* which is then run on the JVM. First of, what is java bytecode? Java bytecode is the instruction set of the JVM and is thus an intermediate form. Secondly, how and why is java bytecode created by AOT? Ahead of time compilation means that the programmed sourcecode gets compiled into a machine specific executable form. This form is, in the case of Java, Java bytecode, i.e. a class-file.[7] As stated before the JVM is the virtual sandbox environment in which an AOT-compiled java program, represented by the java bytecode, is executed. The JVM, like an actual machine, has an instruction set and access to various memory areas. With this in mind a JVM can also be directly implemented as a CPU or in various other direct ways. From knowing what the JVM is, the next paragraph dives deeper into the topic and explains how the JVM is constructed with the help of figure 2.1.

As already explained the JVM executes Java bytecode which can be in the form of *.class-files. Before the thesis dives into the structure one note has to be made: The JVM is usually delivered as a Java Runtime Environment (JRE) or Java Development Kit (JDK). These include besides the Java Virtual Machine also the Java Application Programming Interface (API) classes. These are not directly part of the JVM but play a vital role in the simplification process for code production that is undertaken by the Java project. The next paragraph talks about some Java specifics before taking the topic towards JVM details.

When talking about the JVM the mentioning of classes and interfaces is inevitable. In further parts the distinction between class and interface is abbreviated as class. Multiple causes for class creation exist. The class to be created can be referenced by the constant pool of another class or a class's method can be invoked via reflection. The creation at the start of a program works via loading the initial class and initializing it and furthermore invoking the specified method *public static void main(args[])*. The complete execution is driven by this method which, as soon as the program has more than this specific method and starting class, causes loading, linking and initialization of additional classes and invoking additional methods. [19]

### Class Loader Subsystem

When creating a class in execution the JVM transforms the implementation-independent bytecode into an implementation-specific internal representation of the class. Figure 2.1 shows how the sequential algorithm of loading, linking and initialization is conducted in the class loader subsystem. The algorithm is executed step by step meaning that a class has to be completely loaded before linking can be started and it has to be verified and prepared to full extent before initialization. Again as can be seen in Figure 2.1 this process starts the execution process and is tightly knitted to the runtime data areas of the JVM. The next paragraphs discuss the specifics of the three subcomponents and their tasks.

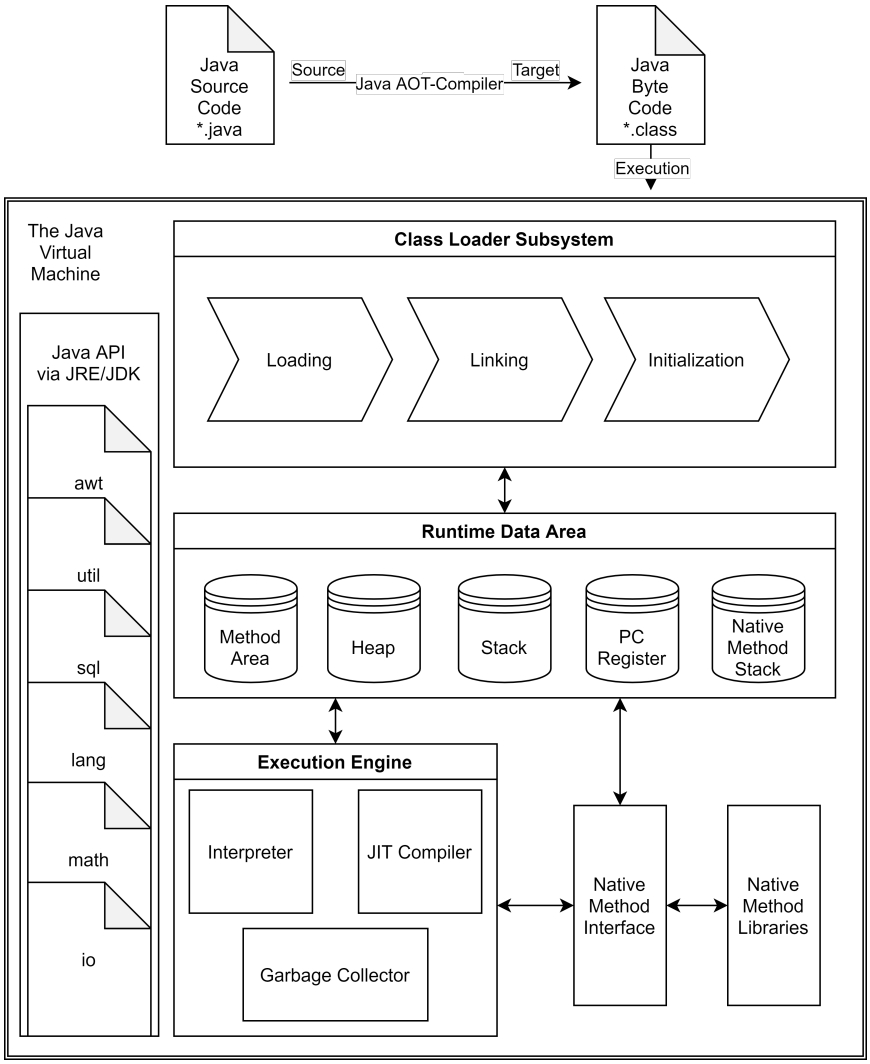


**Figure 2.1:** The Java Virtual Machine simplified structure

The loading is started by the class loader which can be the JVM bootstrap loader or a user-defined one. Before the JVM starts the class loader it checks whether the pair binary class name and class loader already exist in which case the class already exists thus eliminating the necessity of class creation. If creation proves necessary the class loader has to find the class' bytecode representation on the specific platform, usually a file with the class' name in a hierarchical filesystem. After finding, if not a specific error is thrown, the JVM parses the bytecode, which in turn might not produce a valid outcome resulting in different kinds of errors. Optional superclasses may also have to be resolved. After these steps the pair of loader and class are saved by the JVM. Loading usually concerns the class itself however linking concerns its whole ecosystem. [19]

The JVM links a class by preparing the class, superclasses, element type in case of array classes and resolving all symbolic references which as it is a recursive algorithm includes loading and linking these as well. The two strategies for resolution are an eager algorithm, meaning immediate loading

and linking on verification, or loading and linking in a lazy way meaning resources are only loaded and linked when they are about to be used. Both have their respective advantages and disadvantages which won't be discussed in the scope of this thesis. At this point the JVM knows where the binary representation of a class can be found and which other classes are in the direct ecosystem of the class but the class's structure hasn't been checked yet. [19]

Checking the class' structure is the main concern of the verification phase. In this step the class is checked for a static and structural constraints list. These include checks for appropriate type usage, number of arguments, instance initialization before usage, return types and many more. Any caught error in this step lead to ta thrown VerifyError. If no such error occurs the verification is finished and structural correctness of the binary class representation is acknowledged. [19]

When doing the preparation of a class the JVM checks loading constraints on methods overriding those of superclasses and create and initialize static fields to default values. This phase can occur any time between creation and initialization but imperatively before initialization. Resolution in the context of the JVM means the process of dynamically determining conrete values from symbolic references in the run-time constant pool. The target of resolution includes classes, fields, methods, method types, method handles or dynamically computed constants. These different targets require different approaches, requiring class loading, lookup, etc. They can also fail and throw errors or succeed resulting in a resolved dynamic binding in the constant pool. With all these preceding steps the class is now ready for initialization. Conceptually initializing a class is fairly simple as it is the mere execution of the initialization method. This can only be invoked by certain cases or instructions like *new* or a subclass is initialized resulting in the initialization of the superclass. At this point the JVM's multithreading needs to be taken into account since synchronization is imperative. This again is solely handled by the JVM and of no concern for the application programmer. The concrete inner workings are generally spoken handled by object states and unique initialization locks. The Initialization phase is the final stage of the class loader subsystem. Now the different data areas are explained briefly. [19]

#### Runtime Data Area

The JVM run-time data area is subdivided into five different data areas. These fulfill different tasks and are also different in their relation towards processes and threads as there are data areas being defined per thread and others per process. This results in some data areas existing as long as the JVM is running and some areas' existence being linked to their respective thread's existence. Areas that are shared by everything inside the JVM are JVM-defined structures, while thread specific data areas are called thread-defined structures. The data areas are, as can be seen in figure 2.1, the method area, heap, stack, program counter register and the native method stack.

The next paragraphs explain which area is responsible for which task and when they are created and destroyed.

The method area is shared between all JVM threads and is created on JVM start-up. It stores all per-class structures like the constant pool or fields and method data. Those are data every instance of a class shares analogous to each car having four tires.

The heap is a JVM-defined structure, as such is created on JVM start-up, and is shared between all threads. It is an allocation space for class instances and arrays on run-time. Allocation and deallocation in the JVM is automatically handled by the garbage collector which again can be shortly defined as an automatic storage management system. The structure of the heap is not necessarily contiguous and attributes like initial, maximum and minimum size can be defined by the programmer or user with so called command-line arguments that are given to the execution engine to take with the given program.

The stack is a thread-defined structure and is itself subdivided into thread-specific stacks. It does have a fixed size which is defined at stack creation. Regarding the data a stack holds: local variables, partial results and has its share in method invocation and return. The not necessarily contiguous stack is only changed by pushing and popping frames.

The program counter (PC) register holds the current program counters of all running threads and as such is a JVM-defined structure. A thread specific PC register contains the address of the thread-specific currently executed method. The native method stack is an optional data area not supported by all JVMs. If it is supported it is similar to conventional stacks on a thread basis. They can be of fixed or dynamic size. These stacks fulfill a similar task to the regular stacks but are for code not written in Java. With the data areas explained the execution engine is the focus of the next few paragraphs.

### Execution Engine

The execution engine is mainly built up of three different components, the interpreter, the just-in-time (JIT) compiler and the garbage collector as can be looked up in figure 2.1. The interpreter is the part that executes the code directly/live including byte code getting translated to actual implementation-dependent machine code and its execution on the central processing unit (CPU) per instruction. This mechanism causes the interpreter to reinterpret, translate and execute the same code multiple times per instruction. This redundant interpreter pipeline work makes execution slow compared to AOT-compiled execution. When doing so the interpreter of the JVM profiles the run code for example by keeping track of how often a certain piece of code is executed. This profiling information is then used to choose the parts of code that get JIT-compiled hindering too many redundant interpretation steps.

A part of the byte code chosen for JIT-compilation is compiled to machine code and gets thus turned from implementation independent form into implementation dependent form. Afterwards if this part of the program gets executed again the machine code gets executed directly out from the cache where it's stored in the meantime. This method of choosing code hotspots that get executed often for compilation rather than continuous interpretation is called adaptive compilation and is used in the Oracle Hotspot VM. [19]

The last component left out so far is the garbage collector (GC). It takes care of objects without references resulting them to be unreachable by application code. Such an object will be removed by GC and memory will be freed up. The GC is part of inner workings and fully functions without user interaction but the programmer can make a not guaranteed call for the GC.[19] In programming languages like C a programmer would have to free memory explicitly. The obvious advantage is more direct control by the programmer but the disadvantage are possible memory leaks. The GC thus cares for memory and consequently robust code. A disadvantage of the GC is its appearance as a black box which means as a programmer you can't be sure at what point in execution unreachable resources are deleted. This concludes the JVM discussion with a wrap-up in the next paragraph.

This section explained the JVM ecosystem. It is split up between an AOT-compiler that creates Java bytecode out of Java source code and the runtime. From there on the JVM executes the bytecode. Generally speaking the JVM consists of three areas, the class loader subsystem, taking care of loading, linking and initialization, the runtime data area and the execution engine, consisting of the interpreter, the JIT compiler and garbage collector. Additionally to ease the programmer's work the JRE or JDK come with libraries. What hasn't been discussed in detail so far are technologies for making Java execute faster apart from features like hot spot profiling. These technologies are a core feature of GraalVM which is the topic of the next section.

## 2.2 GraalVM

As seen in the last two sections building a virtual machine for executing programming languages is a non-trivial task needing a lot of effort. A lot of this effort has to be redone for different programming languages. The core idea of GraalVM is to reduce this effort via a layered structure where each language implementation is in essence an abstract syntax tree (AST) interpreter via the Truffle API. All low level implementions like JIT compiling, data areas, memory management, optimizations and the like are reused by each language implementation connecting to the GraalVM or any JVM via Truffle. Considering performance these implementations are best run on top of GraalVM. Reasons for that are explained in the next section.

The GraalVM complete ecosystem is comprised of multiple technologies. In the runtime environment the GraalVM reuses most components from the HotSpot VM like the class loader sub-

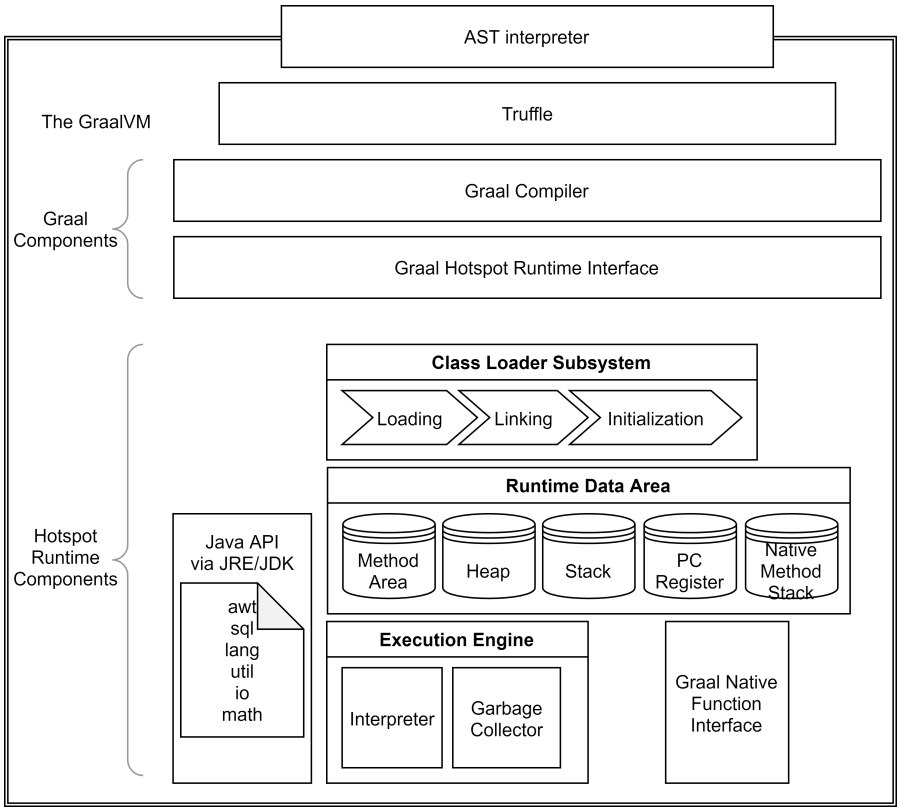


**Figure 2.2:** The Graal Virtual Machine simplified structure

system, the runtime data area and parts of the execution engine, in particular the interpreter and the garbage collector, but replaces the HotSpot JIT compiler with the GraalVM compiler. Since the GraalVM compiler functions differently it also implements an interface between the regular HotSpot components and the GraalVM compiler. Outside of the VM it also has the Truffle framework for different language implementations. Already implemented languages are JavaScript, Ruby, R, python and LLVM-based languages like C, C++ and Rust. Thus the feature is able to end the segregation of programming languages. Another feature is its ability to be embedded in combination with OpenJDK, node.js or inside Oracle Database, while also being able to be run standalone. [6]

### 2.2.1 GraalVM Compiler

The GraalVM Compiler replaces the JIT compiler in a JRE. Other as the regular java JIT compiler it doesn't translate java bytecode directly to machine code but to the Graal IR. [2] The Graal IR is a graph-based high-level intermediate representation. This intermediate representation is key to GraalVM's aggressive optimization techniques based on optimistic assumptions such as specific types and branch prediction. Aggressive optimization also means the compilation is prone to failures which are caught by so called guards. When a guard fails execution is transferred

back to interpretation. This mechanism is called deoptimization. [13] Compiled code gets saved as a VMs internal data structure and subsequent calls to the particular part of code get always directly executed in machine code omitting the unnecessary step of interpreting. Another component the GraalVM project substitutes is the Java Native Interface [19]. It is exchanged with the Graal Native Function Interface (GNFI). [10] It allows for efficient native function calls within Java code.

### 2.2.2 Truffle API

The Truffle language implementation framework enables programmers to write an AST interpreter in Java for any programming language. The biggest advantage is the reuse of JVM runtime services such as automatic memory management with execution by a JVM. But how does the interpreter work?

The interpreter works by modeling language structures as nodes building the abstract syntax tree. Evaluation of the tree works by executing each node. Each node type shares basic specifics for executing their respective semantics. The whole tree gets then recursively evaluated by the recursive execute method calls.

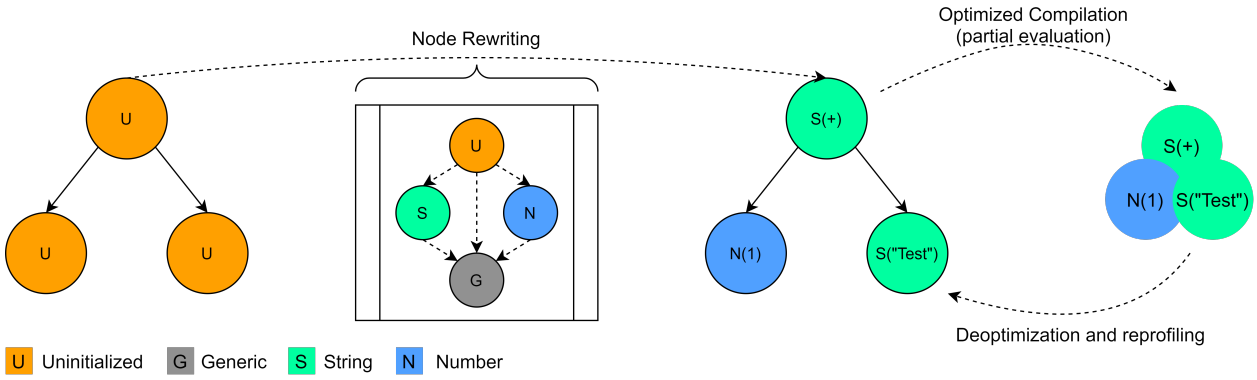


**Figure 2.3:** Simplified Node Rewriting example

Optimizations on AST level mostly work with speculative specialization techniques. On the left side of Figure 2.3 the nodes are in an uninitialized state and write themselves into specialized variants at run time where profiling information is available. [27] At that point execution happens in the interpreter. This specialized tree can then, after certain conditions are met, be compiled into what can be seen at the far right side of Figure **??**. As we have discussed earlier in the GraalVM compiler speculative optimizations, like node specialization, are prone to failure. As soon as the compiled version's execution fails execution goes from the specialized compiled version back to a more generic version being interpreted and the cycle starts again. This cycle process is shown in the right side of Figure 2.3.

What kind of specializations are possible via this approach? Type specialization especially with dynamic languages is an operation reducing code on runtime and thus creating speed up. To discuss this optimization method in more detail the add method with the + operator in the scripting language JavaScript is used. The add method in JavaScript is defined for various types including Strings and Numbers. We reduce the language's complexity to the aforementioned two types in the example. The example code in Listing 2.1 shows how addition in JavaScript works in different settings. The interpreter has to check the types of two operands from left to right before it can decide on which operation to use, an addition for numbers or one for strings. In line 1 a number and a string addition takes place, resulting in a string concatenation. In line 3 the interpreter takes the left plus as an integer plus, resulting in the addition of 1 and 2 to 3, while the second plus is interpreted as a string addition, leading again to a string concatenation. In line 5 the first plus is interpreted as a plus-sign, resulting in the string being parsed into a number which works in this case but not in the case of line 7. The second addition is a numeric one again in both lines.

```
1  var w = 1 + "Test" // results in "1Test"
2  typeof(w) // string
3  var x = 1 + 2 + "Test" // results in "3Test"
4  typeof(x) // string
5  var y = +"1" + 2 // results in 3
6  typeof(y) // number
7  var z = +"Test" + 1 // results in NaN
8  typeof(z) // number
```

**Listing 2.1:** JavaScript addition types minimal example

The node rewriting process shown in Figure 2.3 happens at various states to achieve different goals. In the interpreter node rewriting is used to capture profiling feedback via runtime information such as the rate of node rewrites. If certain thresholds are surpassed the specialized AST gets compiled into machine code. If the execution fails the compiled code gets deoptimized back to the AST interpreter where again nodes are rewritten and now update the existing profile information. After this step the newly specialized AST gets compiled again. [27]

One note on the compilation process: The compiler uses as said aggressive optimization methods. This operation is successful for stable ASTs. Then compilation techniques like method inlining and eliminating interpreter dispatch code can be performed. This technique is called partial evaluation.

In addition to type specialization truffle offers so called polymorphic inline caches. [14] An inline cache itself is simply caching a method from a former lookup at call site. In Truffle inline caches are built by chaining nodes which check for cached target matches which are then executed. At a predefined chain length the chain gets replaced by a polymorphic node handling all operations.

As discussed in the JVM section program loading often requires resolution. An implemented truffle language can cache resolved targets at run time by replacing an unresolved node with its resolved version. Thus further resolutions of this node are prevented which leads to an optimized version of the AST.

This section gave a small introduction into the Truffle API for implementing languages via an AST interpreter for running on top of a JVM and showed the optimizations that are conducted on AST level. After an AST at run time becomes stable with no more rewrites the AST is compiled dynamically to machine code. Truffle uses the JIT compiler of a JVM, in case of GraalVM the GraalVM compiler. If GraalVM is used the nodes' execute methods are inlined producing one compilation unit of the whole tree. In the next step the GraalVM compiler uses its aggressive optimization techniques for the whole inlined tree unit producing efficient machine code. These combined methods are based on the principle of the first Futamura Projection also called partial evaluation. [5] As stated in section 2.2.1 guards or also optimization points need to be implemented to check all the assumptions made before compilation. As soon as it is necessary to rewrite a node during machine code execution the control flow gets transferred back to the interpreted AST to rewrite the node either on profiling information to another specialized version or a more generic version. [13]

In conclusion the GraalVM project aims at simplifying language implementations and application programming while also trying to have a rapid run time. The simplification comes from introducing a framework for implementing languages as AST interpreters which can run on top of any JVM. The performance benefits are especially achieved from running the AST interpreter on top of the GraalVM which uses various techniques to bring run time performance to higher levels.

## 2.3 Graal.js

The Graal.js project is an AST interpreter on top of Truffle. It is in full compliance of the newest ECMAScript 2021 language specification [16, 17] and growing support of the technical committee 39 (TC39) proposals for future ECMAScript language specifications. [12, 18] The runtime can execute JavaScript and Node.js applications with the previous discussed benefits of the GraalVM technology stack. [8] As a recap the most notable ones are performance and polyglot programming support with multithreading support.

## 2.4 ECMAScript & the current state of modules

As stated in the introduction, in Chapter 1, ECMAScript is a script language that is the standard specification for JavaScript. The naming conflict originates from copyright issues with the name JavaScript being trademarked by SUN and ECMA being the organization on standardizing the language. [26] It is maintained by the ecma organization with the formalization process starting in 1996 resulting in the first ECMA-262 (ECMAScript specification) edition in 1997. Current state is the 12th edition with name EcmaScript 2021. [11] The language is a multi-paradigm, dynamically typed, general-purpose scripting language with first-class functions. This means the language supports functional, imperative, reflective and object-oriented programming. The object-orientation is prototype-based meaning existing objects are used to reuse behavior. These design decisions were prone to blistering criticism in the past but still reside in the language. [26] A very specific language feature that hadn't been supported initially is the module.

With ECMAScript 6 the language included modules making it possible to divide code up into multiple files. Although this possibility has been around in JavaScript before, it was implemented by external libraries and not an inherently specified language feature. The ability to import modules as needed came with a rise in source code size in regular JavaScript programs. When the language started out it was not unusual to have a one-line `.js`-(JavaScript)file. As the code size grew the necessity of modules came with it which led to a multitude of concurring solutions. As script concatenation as the basic step meant manual building and testing, this method could only be used in simple projects while more difficult ones started using module loaders. Those in turn could also become complicated for larger code bases. To overcome these problems so called module bundlers, like Babel[1], could be used to generate the needed JavaScript code at build time. Thus the code includes all dependencies in a single concatenated file. The specification of modules in ECMAScript bundled these efforts into one concept and gave implicit module support by the language. With this specification feature performance of module loading was pushed towards engine implementation.

In a short note to avert term mixup. ECMAScript is a language specification for script languages. The actual code written is in JavaScript and this language is then run via engines in browsers on computers. In hard terms the ECMAScript specification does not need to be fully supported by a particular browser and single features sometimes are left out but in general engine implementations follow the ECMAScript specification leading to the specification's features in JavaScript sourcecode.

In a broad sense modules at current state read like regular scripts with the difference of them being in strict-mode only and using imports and exports. The benefits are separate files with self-contained functionality, sharing of those files between different projects, diminishing naming

[1] https://babeljs.io/

conflicts and the robustness that comes with shared open source. But what are modules in the context of JavaScript in detail?

As a module is run in strict mode all declared variables, functions and classes are private and cannot be accessed from the outside. Only those top-level items that are explicitly exported by the keyword *export* can be used from a script importing the module. This is then done by the keyword *import* either the whole module or explicitly named items which can also be aliased.

As modules have to be loaded and are sometimes scattered throughout the web, deferring module loading is a default setting. Another note is the caching of modules. A module is only executed the first time it gets imported. All other scripts importing the module are given the already evaluated exports. This also means that a change in one of the exports is visible to all importing scripts.

This standard unfortunately left out the possibility to add inline modules into a script which results in modules just containing one exported function. This results into unnecessary files which could be avoided by inlining the module into the original script. This exact feature is discussed in Section 2.6.

## 2.5 ECMAScript proposal process

Before coming to module blocks themselves the general proposal process is to discussed in order to give insight at what development step the proposal resides at the time of this thesis. The ECMAScript proposal process [24] is split up in five stages from zero to four. The first notable stage is stage one where the general shape of the proposal has to be outlined with examples, problems and the shape of their solutions all collected in a public repository. At this stage the proposal is expected to undergo major changes. To be uplifted to stage two the proposal has to include an initial spec text with all major semantics and syntax but allowing placeholders. The stage is still expected to undergo certain changes but the general outline should be set. To reach stage three the proposal needs to have a finished and complete spec text that has been reviewed. In simple words the theoretic groundwork is finished and further improvements need empirical knowledge gathered on actual implementations. From this stage on the proposal is expected to undergo minimal change. The final stage before acceptance is stage four. Apart from a finished specification the proposal needs acceptance test in compliance with the rest of the ECMAScript tests and implementations that pass these tests as well as live experience by two VM implementations. [24] This concludes the proposal pipeline discussion which serves as a state overview for the upcoming proposal module blocks.

## 2.6 Proposal: Module blocks

Module blocks are an official TC39 proposal from Daniel Ehrenberg and Surma at stage 2 as of July 2021. [4] As stated ECMAScript does not have the feature of inlining modules. This causes several issues in programming applications, most notably code composition and file count but also CSP (Content Security Policy). In the current specified state if code should be run in parallel an additional file is needed resulting in every Worker (node.js feature) needing a separate file. So module blocks work like a module but are inside the script where they are also imported. One important note to take is the non closure of module blocks following the principle of coding in one place, running it anywhere. It can thus also be imported from any Realm and is code independent of a particular context. So in a sense module blocks are mainly a syntactical addition to ECMAScript as the concept is to provide a syntax for inlined module contents as can be seen by Figure 2.4. [4, 11] These can then afterwards be imported asynchronously. Reason being the ability of module blocks to import other modules as well. As regular modules since ES6 module blocks will be cached in the module map after the first import. One note to take is, that module blocks need to be imported via dynamic `import()` calls rather than import statements. This is due to them not being addressable as specifier strings. Dynamic import has a particular positive impact on performance as module blocks will only be loaded on a base of need.

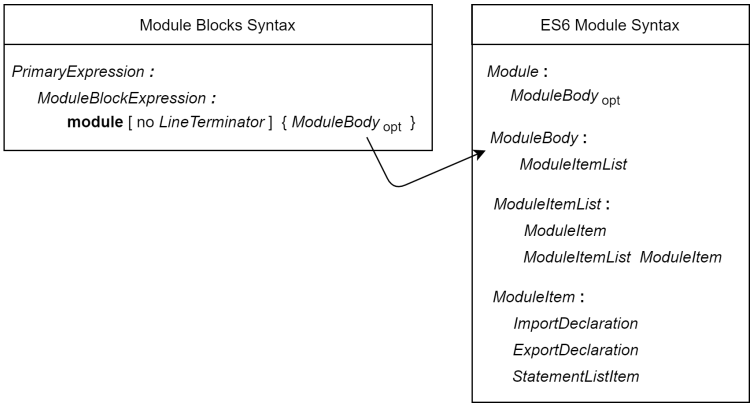


**Figure 2.4:** Syntax of Module Blocks and ES6 Modules

# 3 Implementation

This chapter lays out the implementation by presenting an architectural overview on how and where the proposal can be included in the Graal.js project which is further detailed in the upcoming sections. The implementation explanation is then wrapped up by explaining the interaction with dynamic imports and the serialization / deserialization framework.

## 3.1 General overview

Before thinking about implementing the proposal into the Graal.js project one needs to know how source code is evaluated by Graal.js. The path of source code going through the project is straightforward. Without going too deep into the specifics it follows figure 3.1. The source code is traversed and transformed into a token stream. This token stream is then evaluated by the parser which uses the found token in conjunction with the specified grammar given by ECMAScript [11] to transform the token into an intermediate tree form. This tree contains nodes representing general language concepts without specifying the nodes into specialized forms. As can be seen in Figure 3.1 a module block is represented by a function node in the intermediate form. Later on in the translation pipeline the `GraalJSTranslator` class is called with an intermediate form node and translates it into a specialized form. In the scope of this thesis a `FunctionNode` to a `ModuleBlockNode`. These specialized nodes are then executed when being visited in the tree. On the specifics of interpretation and compilation during execution see Section 2.2.1.

Implementing the module block proposal into the Graal.js project involves several steps. Those are adjustments to the parser to capture the newly introduced syntax, implementation of the prototype, introducing a new respective node for module blocks and lastly adjusting the node for dynamic import-calls. At this point the implementation is limited to the available specification of the proposal. The chapter's outline is firstly the module block direct implementation into the Graal.js project followed by explanations of some necessary adjustments for dynamic import-calls. The direct implementation part is split up in four parts: parser changes, module block node, constructor and the prototype.The specific parser changes are explained in the next section.

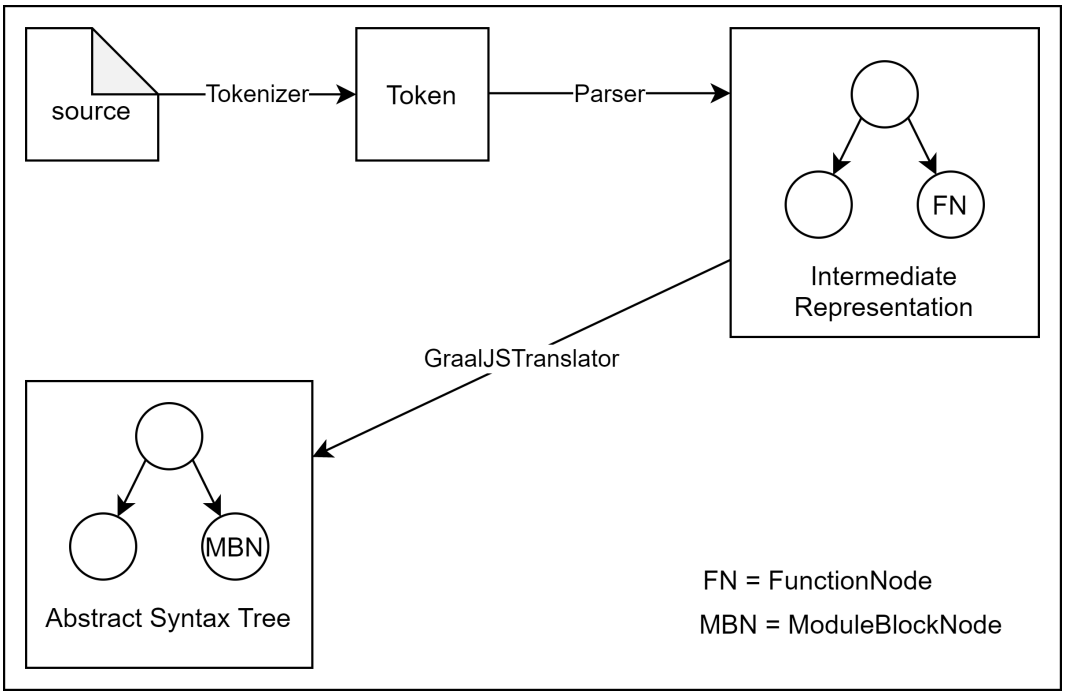


**Figure 3.1:** Code evaluation by Graal.js

## 3.2 Parser

From the specification we can take away that `module` won't be introduced as a keyword in ECMAScript meaning during parsing the word will be treated as an identifier. As both, identifier and module blocks, are classified as so called `PrimaryExpressions` the main change is happening in this explicit parser method inplace of the "ident"-Token branch. Like Listing 3.1 shows `module` can still be used as identifier. Thus recognizing a module block needs a lookahead token and additionally satisfy the no line delimiter condition. This is done by first checking for the next token being a `{` after a `module` identifier , changing from the "ident"-branch to the module block branch and then checking whether there is at least one line delimiter between the `module` and the `{`. A line delimiter occurrence would result in a syntax error. Otherwise the parsing continues with capturing all statements inside the module block similar to a regular module and then putting the result into the same intermediate representation form a regular module would be put in. This caters to the notion of modules and module blocks acting similar in the language specification. The only relevant difference is the end of the parsing which is the end of file in the module case and a closing brace (}) in the module block case. The next section talks about further steps that are taken during translation.

```
1    var module = 42;
2
3    var moduleBlock = module { };
```

**Listing 3.1:** Minimal module identifier example in JavaScript

## 3.3 ModuleBlockNode: Module Blocks' Truffle AST representative

The Graal.js project includes a class called `GraalJsTranslator`. This class is generally speaking for translating the intermediate representation nodes into specialized ones for the resulting AST. During execution of this class the intermediate module block representative gets translated into a module block node. The node is built according to the provisional proposal's specification meaning the hidden properties are set as defined. At the current point the `hostInitializeModuleBlock`-method which is mentioned in the specification is not implemented as it is not included in the specification yet. With the node representation itself finished the next section turns towards the prototype and the constructor.

## 3.4 ModuleBlock prototype and constructor

This part of the implementation is split up into two parts where firstly a discussion is depicted as to why the constructor is specified as is. Secondly the actual implementation is explained. At current state, as of July 2021, it is unwished for by the developers to implement yet another eval-esque structure coming from the desire for less dynamic evaluation. A point for including an actual constructor for module blocks would be the symmetry and consistency in the whole language's specification. This concern can be brought up since structures like `AsyncFunction` or `GeneratorFunction` do have constructors with a code-string parameter. Right now the eval-point outweighs the symmetry-/consistency-point. The constructor when called simply throws a type error. This concludes the rather simple implementation of the constructor with the discussion behind it maybe resulting in a change of the constructor's specification. The next paragraph concludes the implementation of the module block itself by introducing the module block prototype.

The prototype is fairly simple constructed. Its prototype property is the module block Prototype Object with the general attributes writable, enumerable and configurable all set to false as is standard in the ECMAScript specification. The module block Prototype Object is `%ModuleBlock.prototype%`. Its internal prototype slot is `% Object.prototype %` and it is an ordinary Object. It has simply one function which is the toString() method. This method should check whether it is used on a `ModuleBlock` object and as such return the hidden `SourceText` property. On a second note `ModuleBlock` is specified as global. This allows for **??** to work. Again this matter can be subject to change during the proposal's progress.

```
1  var moduleBlock = module { };
2
3  moduleBlock instanceof ModuleBlock; // true
```

**Listing 3.2:** "ModuleBlock working as global in JavaScript"

This concludes the module block implementation itself. The module block's syntax is implemented in the parser, the module blocks runtime semantics are implemented in the specified module block node for the resulting AST and the constructor's and prototype's specification is met accordingly with also registering ModuleBlock as global. In conclusion the module block at this state is implemented as a standalone without much interaction with the rest of the EcmaScript ecosystem. This leads us to the question how does the module block interact with existing language features. The proposal's specification only states one interaction with another language feature: the import()-call. Thus the next paragraph states the necessary changes for integrating module blocks into the dynamic import calls.

## 3.5 Interaction with dynamic import

Hence, only the Graal.js representative, the class ImportCallNode, needs to be changed in this matter. The main difference is the change from using only Strings with the module's URL as identifiers to using the module block node as identifier for the resulting Module Record. To accomplish this, the object needs to be checked and if it does not satisfy the condition of being an object with the internal slot of ModuleBlockBody, a String is then used as identifier, otherwise the ModuleBlockNode. After that the regular HostImportModuleDynamically routine is called as would be for regular modules. This results in a Module Record which is saved in the module map. The module map itself also has to be changed to accept the module block node as key as well as the URLs used for regular modules. During that process the three-step phase of modules is executed. First the ModuleBlock source code gets parsed as it would be a separate module. It is then instantiated and at last evaluated. Through these explicit inner workings the module block preserves the positive effects of modules while eliminating the aforementioned negatives.

The state at July 2021 the dynamic import is the only interaction being inflicted by the module block proposal. With the changes made on the ImportCallNode and the aforementioned changes to the parser, the introduction of the ModuleBlockNode and the registration of the respective prototype and constructor the implementation phase is finished. As already mentioned some specifics might have to be changed in the future as the proposal matures through the stages but the skeleton is led out at this state. The following paragraph and figure 3.2 conclude the implementation on a comparison between modules and module blocks on implementation level behavior.

## 3.6 Serialization framework

This section presents the framework for transferring module block source code from one process to another, i.e. one machine to another. For this to be possible two functions have to be imple-

mented and registered at the global object in JavaScript. Their names exactly match their purpose: `serialize` and `deserialize`. The serialize method is straightforward and is oriented on the `toString`-method by accessing the `SourceText` property and then taking the string gained from the access and turning it into a byte-array. Fortunately, Java itself has a builtin method for strings turning them into a byte array using the executing platform's standard charset. This already concludes the serialization of a module block. The deserialization involves a few more steps. The given byte-array has to be turned back into a string, which again is builtin into Java, as the string class has a constructor taking a byte-array and a charset. Then an internal source object has to be created. The source object is created via the beforehand created string. Then the source code has to be parsed and translated as explained in Sections 3.2 and 3.3.

## 3.7 Wrap up

On regard of parsing the main difference between module blocks and modules is the outside call onto the class when parsing modules but module blocks have to be caught from inside the parser. Hence the parser is told explicitly to parse a module from another class while a module block is caught by the inner class's logic and will from then on be treated similarly to the regular module. Both parsing results are then encapsulated into the same intermediate representation as a `FunctionNode`. When translating this intermediate representation into the final AST representative the path of modules and module blocks splits up again as the module block is transformed into a `ModuleBlockNode` and the module is transformed into a `JSFunctionExpressionNode`. Finally as those constructs are imported, in the module's case also statically, dynamically the final result will always be a `JSModuleRecord` which is the AST's representation of the, in ECMAScript specified [11], Module Records. The description is also envisioned by the subsequent figure 3.2 This brief difference in behavior explanation concludes the implementation chapter which is followed by the evaluation chapter detailing the testing of the as set out above implementation.

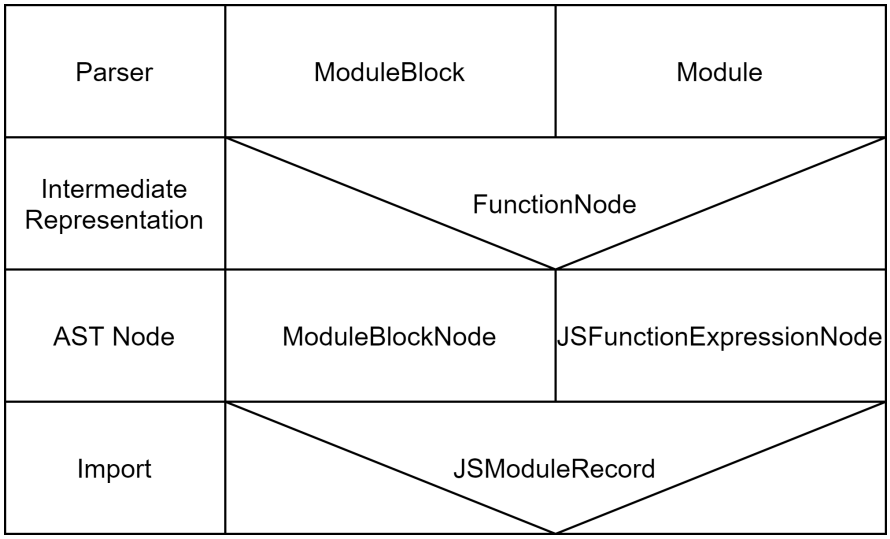


**Figure 3.2:** Simplified GraalJS translation pipeline of module blocks and modules

# 4 Evaluation

The presented module blocks implementation which has been embedded inside the Graal.js project, was evaluated by running the provided tests in the project. Those tests represent the Test262 implementation conformance test suite. [23] Since these tests can't cover module block syntax and semantics additional tests were conducted.

The Test262 suite combines the three ecma standards ECMA-262, ECMAScript Language Specification, ECMA-402, ECMAScript Internationalization API Specification and ECMA-404 the JSON Data Interchange Syntax, also known as ISO/IEC 21778. The suite's test files cover any behavior stated in these specifications and is comprised of around 30.000 individual test files. [3, 23] Although the test environment is extensive full coverage cannot be guaranteed. It's certain to say that the suite doesn't yet contain any tests that cover module block functionality. This shifts the topic towards the tests that were designed and conducted in the course of this thesis.

The main orientation for designing tests regarding the module block proposal came from the available basic specification. While it is not finished yet, the parts of the specification that could be implemented can also be tested. First of, the syntactic part of the specification can already pose different testing scenarios.

```
1  var module = 42;
2
3  var nextModule = module;
4
5  var moduleBlock = module { };
6
7  var failingModuleBlock = module
8      { };
```

**Listing 4.1:** Module block syntax tests

The first syntactic specialty is the fact that "module", while acting as a keyword in the scope of module blocks, won't be added to the keywords group in ECMAScript. The consequences are random variables being used with "module" as identifier are allowed and line delimiters between `module` and the `{` are forbidden. Since the `ModuleBody` part of the module block syntax is optional an empty module like in line three of Listing 4.1. This concludes the three testing cases for the syntax. Next up are tests regarding the prototype and the constructor.

The constructor of module blocks basically doesn't have to exist as it simply should throw a `TypeError`. The decision during implementation was made that it actually is implemented but when called throws the specified error. Still it doesn't change the expected behavior and thus a simple call to the constructor is made and expects the aforementioned error. Testing the prototype requires more elaborate testing since multiple aspects need to be checked. Those regard especially the prototype property of the module block prototype object itself but also the module block prototype property. The described tests are conducted by source code similar to Listing 4.2.

```
1  var constructorModuleBlock = new ModuleBlock(); // TypeError: ModuleBlock
       is not a constructor
2
3  var moduleBlock = module { };
4
5  var instanceTest = moduleBlock instanceof ModuleBlock; // true
6
7  var moduleBlockPrototype = moduleBlock.constructor.prototype; // object
```

**Listing 4.2:** Module block constructor and prototype tests

The following tests mainly regard the module blocks interaction with different ECMAScript constructs inside of the module block, in particular the optional syntax part `ModuleBody`, and its interaction with the dynamic import. The interaction with existing ECMAScript structures is straightforward as all of them have to be used inside a module block as part of the `ModuleBody` to some extent. The following Listing 4.3 list some example cases with a simple number variable, a function and an async function.

```
1  var moduleBlock = module { export var x = 42; }; // export number
2
3  var moduleBlockFunc = module { export function square(x) {
4          return x * x;
5      }
6  }; // export function
7
8  var moduleBlockAsync = module {
9      function resolve() {
10         return new Promise(resolve => {
11                 resolve('resolved inside module block')
12         });
13     }
14
15     async function asyncCall() {
16         return await resolve();
17     }
18
19     export var resolved = asyncCall();
20 }; // export promise
```

**Listing 4.3:** Module block general test example cases

A particular case that should also be noted as it raised interest during the proposal presentation is the ability to bundle module imports into a module block which was also tested extensively with importing modules in the different possible ways and also importing module blocks. The specialties of module block imports are explained in the next bit.

Module blocks can only be imported dynamically with awaits. Since top level awaits are not supported dynamic imports have to be encapsulated in async functions. Hence imported module block module records are encapsulated in a promise and have to be processed in a then-environment. Listing 4.4 shows the exact syntax needed for importing the module block.

```
1        var test = (async function() {
2            var moduleBlock = module { export var x = 42; }; // export number
3
4            return await import(moduleBlock);
5        })();
6
7        test.then( function(module) {
8            console.log("Module block x: " + module.x);  // Module block x: 42
9        });
```

**Listing 4.4:** Module block dynamic import test

In conclusion adding a new feature into an engine for ECMAScript requires extensive testing. First of the general preexisting test suite Test262 has to be applied to the changed semantics and have to pass. Then all intricacies of the proposal's specification have to be tested. Although the proposal is a rather syntactic one it still introduces semantics which have to be tested. Next up after all internals are checked the interaction with preexisting ECMAScript structures needs to take place which furthers the extensiveness of testing.

# 5 Future Work

Future Work on the thesis's matter is tightly tied to progress on the module block proposal. At current state the proposal resides at stage two where neither the specification is finalized nor the details are all cleared. A specific case of this is the hostInitializeModuleBlock-method mentioned in the runtime semantics of the module block specification, as it is merely mentioned but not specified. The work needs to be continued to keep the implementation on par with the specification. Especially as soon as the proposal reaches stage three the final specification is available. Subsequently the implementation has to be adjusted to fit the final version. Furthermore as the proposal progresses official tests will be released and later on embedded into the official Test262 suite. Those tests then have to be applied to the finished implementation. The preceding development description is rather rough, therefore the following paragraphs discuss details on the proposal that are not fixed yet.

A controversial implementation detail up for possible change is the constructor of module blocks. At current development state a constructor call leads to a TypeError. The relevant point for rejecting the ability to create a module block via a constructor is that module blocks would turn into an eval-like structure which is unwanted for as the web community generally rejects the eval-method. Due to security issues this sentiment gained widespread support. On the other hand one can argue that a language's design should be sound and symmetrical. Other structures that were introduced in ECMAScript that share the eval-like constructor discussion do have a working constructor. Thus for reasons of symmetry module blocks should have a constructor as well. The result of the discussion will be visible latest at stage three of the proposal. Right now the simple constructor implementation only throws the error and would have to be fully implemented if a working constructor is introduced in the proposal.

Another feature that might be introduced is an abbriviated form of module blocks. Due to the feature's rather syntactic nature the requirement are parser changes leading then into the same semantics as module blocks. The feature might be included if import-free-single-function module blocks become a frequently recurring pattern. Also not entirely clear is what the import.meta.url of a module block should look like. Although it is save to assume that it's bound to the module's import.meta.url in which it is specified it is frowned upon to be the exact same. A possible setup would be the module's import.meta.url directly followed by something like: "$L\#^{+}C\#^{+}$" where the #s stand for the exact line and column number of the module block. As this is also

not fixed yet the implementation has to follow suit as soon as the import.meta.url semantics are fixed.

In short, the implementation cannot be finalized in the scope of the thesis as the proposal won't be finished by far. A lot of details need to be worked on before the proposal can reach stage three. With stage three reached the implementation can be finished as the specification will be finished more or less. The testing in the scope of the thesis has to be adapted to the final implementation and the tests being released before the proposal can reach stage four. Since the implementation is tied closely to the proposal it can only be done as soon as the proposal is completed and adopted by moving to stage 4 which will earliest be in 2022.

# 6 Conclusions

In the scope of the thesis the ECMAScript TC39 proposal module blocks was implemented into the Graal.js project which resides inside the GraalVM ecosystem. The proposal enhances module usage in ECMAScript and has thus the potential to greatly impact their usage throughout the internet especially by simplifying use of Workers. For this to happen the proposal itself is not enough, at least not in the Graal.js project since code has to be serialized before being sent to another process. The serialization and deserialization is also implemented in order to enhance the practical feasibility. Now, inline JavaScript code can be sent back and forth between processes resulting in lower network footprint as code and results have to be sent but not whole data packages. Another point to keep in mind that early proposal adaptation greatly helps keeping on par with the ECMAScript specification's development but also the rival engines. The implementation passes the preexisting Test262 official ECMAScript test suite and explicitly for the proposal written unit tests on it. Since the proposal is not finished yet the thesis laid the groundwork for implementing the upcoming further developments on the proposal into the Graal.js project.

# List of Figures

# Listings

# Bibliography

[1] Pierre Carbonnelle. URL: https://pypl.github.io/PYPL.html (visited on 04/03/2021).

[2] Gilles Duboscq et al. "Graal IR: An Extensible Declarative Intermediate Representation". In: Feb. 2013.

[3] *ECMA-414*. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-414/ (visited on 06/10/2021).

[4] Daniel Ehrenberg and Surma. *JS Module Blocks*. URL: https://github.com/tc39/proposal-js-module-blocks (visited on 03/31/2021).

[5] Yoshihiko Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12 (Dec. 1999), pp. 381–391. DOI: 10.1023/A:1010095604496.

[6] *Get started with GraalVM*. URL: https://www.graalvm.org/docs/getting-started/ (visited on 06/08/2021).

[7] James Gosling et al. *The Java Language Specification Java SE 16 Edition*. Oracle America, 2021. URL: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf.

[8] *GraalVM JavaScript Implementation*. URL: https://www.graalvm.org/reference-manual/js/ (visited on 03/31/2021).

[9] *GraalVM Native Image*. URL: https://www.graalvm.org/reference-manual/native-image/ (visited on 06/08/2021).

[10] Matthias Grimmer et al. "An efficient native function interface for Java". In: Sept. 2013, pp. 35–44. DOI: 10.1145/2500828.2500832.

[11] Shu-yu Guo, Michael Ficarra, and Kevin Gibbons, eds. *ECMAScript 2022 Language Specification*. URL: https://tc39.es/ecma262/ (visited on 03/31/2021).

[12] Jordan Harband et al. *ECMAScript proposals*. URL: https://github.com/tc39/proposals (visited on 03/31/2021).

[13] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging Optimized Code with Dynamic Deoptimization." In: vol. 27. July 1992, pp. 32–43. DOI: 10.1145/143103.143114.

[14] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches." In: July 1991, pp. 21–38. ISBN: 3-540-54262-0. DOI: 10.1007/BFb0057013.

[15] *Introduction to GraalVM*. URL: https://www.graalvm.org/docs/introduction/ (visited on 06/08/2021).

[16] *JavaScript Compatibility*. URL: https://www.graalvm.org/reference-manual/js%20/JavaScriptComp (visited on 03/31/2021).

[17] Kangax, Webbedspace, and Denis Pushkarev. *ECMAScript compatibility table*. URL: kangax.github.io/compat-table/es2016plus (visited on 03/31/2021).

[18] Kangax, Webbedspace, and Denis Pushkarev. *ECMAScript compatibility table*. URL: kangax.github.io/compat-table/esnext (visited on 03/31/2021).

[19] Tim Lindholm et al. *The Java Virtual Machine Specification Java SE 16 Edition*. Oracle America, 2021. URL: https://docs.oracle.com/javase/specs/jvms/se16/jvms16.pdf.

[20] Maika Möbus. *GraalVM: Clearing up confusion around the term and why Twitter uses it in production*. URL: https://jaxenter.com/graalvm-chris-thalinger-interview-163074.html (visited on 07/01/2021).

[21] *p-System Operating System manual*. URL: http://www.sageandstride.org/html/manuals.html (visited on 07/01/2021).

[22] James E. Smith. *Virtual machines :: versatile platforms for systems and processes*. Amsterdam ; Boston: Morgan Kaufmann Publishers, 2005.

[23] *Test262: ECMAScript Test Suite*. URL: https://github.com/tc39/test262 (visited on 06/10/2021).

[24] *The TC39 Process*. URL: https://tc39.es/process-document/ (visited on 06/08/2021).

[25] *Using Web Workers*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (visited on 07/01/2021).

[26] Allen Wirfs-Brock and Brendan Eich. "JavaScript: The First 20 Years". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386327. URL: https://doi.org/10.1145/3386327.

[27] Thomas Würthinger et al. "Self-optimizing AST interpreters". In: *ACM SIGPLAN Notices* 48 (Jan. 2013), pp. 73–82. DOI: 10.1145/2480360.2384587.