JOHANNES KEPLER
UNIVERSITY LINZ

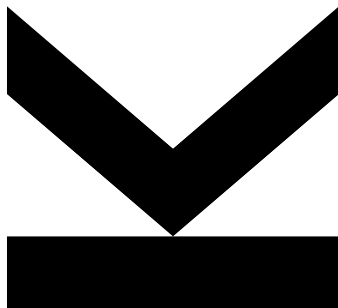
Author
Alexander Stummer

Submission
Institute for System Software

Thesis Supervisor
o.Univ.-Prof. Dr. Hanspeter Mössenböck
Assistant Thesis Supervisor
Dr. Lukas Stadler,
Dr. Christian Wirth

October 2023

ECMASCRIPT'S PIPELINE OPERATOR FOR GRAAL.JS



Master's Thesis
to confer the academic degree of
Master of Science
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at
DVR 0093696

Kurzfassung

JavaScript ist aktuell eine der meistverwendeten Programmiersprachen. Aus diesem Grund werden kontinuierlich neue Features zur Sprache hinzugefügt. Bevor dies passiert, müssen Features einen mehrstufigen Prozess durchlaufen um eine standardisierte JavaScript-version zu gewährleisten. Um es Benutzern zu erlauben, verschiedene Vorschläge schon vorab auszuprobieren, stellt Graal.js experimentelle Features zur Verfügung. Einer dieser neuen Vorschläge ist der Pipeline-Operator, welcher sich im Moment in Stufe 2 des Prozesses befindet. Darüber hinaus ist es eines der meistgewünschten Funktionen in der Community. Graal.js läuft auf der GraalVM und verwendet das Truffle-Framework. Dieses ermöglicht Toolsupport und die Interoperabilität mit anderen Truffle-Sprachen.

Diese Arbeit beschreibt den Ansatz, den Pipeline-Operator als experimentelles Feature in Graal.js zu implementieren mit Hilfe der in der Spezifikation beschriebenen „Topic Bindings“. Die Pipeline wird aufgelöst, um die Kompilierung zu vereinfachen und gute Leistung beizubehalten. Des Weiteren wird der Ansatz mit dem äquivalenten Code ohne Pipeline verglichen und die Interoperabilität mit anderen Truffle-Sprachen wird getestet.

Abstract

JavaScript is currently one of the most used programming languages. Due to this fact, the language is continuously extended by new features. Before those are added, they need to run through a multi-stage proposal process by ECMAScript, to ensure a standardized JavaScript version. To allow users to experiment with and test out different proposals, the Graal.js engine provides experimental features. One of these new proposals is the pipeline operator, which is currently at Stage 2 and one of the most desired features by the community. Graal.js operates on the GraalVM through the Truffle language implementation framework. This enables tooling support and interoperability between several languages.

This thesis describes the approach to add the pipeline operator to Graal.js as an experimental feature by using the topic bindings described in the specification. The pipeline itself is desugared to simplify compilation and maintain good performance. Furthermore, the implementation is compared to equivalent code without pipelines and the interoperability with other Truffle languages is explored.

Table of Contents

1. Introduction 5
   1.1. Motivation 5
   1.2. Task 7
2. Foundations 8
   2.1. JavaScript 8
   2.2. ECMAScript proposal process 10
   2.3. GraalVM 11
   2.4. Truffle 12
   2.5. Graal.js 14
3. Architecture 15
   3.1. Nodes 16
   3.2. Additions to the architecture 16
   3.3. Pipeline operator syntax 17
      3.3.1. Topic reference 18
      3.3.2. Examples 18
4. Implementation 20
   4.1. Approaches 20
      4.1.1. First approach 20
      4.1.2. Second approach 24
      4.1.3. Third approach 27
   4.2. Error handling 31
   4.3. Tests 32
5. Technical Data 34
   5.1. Benchmarks 34
   5.2. Interoperability 41
6. Related Work 43
7. Conclusion 45
8. References 48
9. Picture References 50

1. Introduction

JavaScript is one of the most widely used programming languages in the context of the web. To keep up with the constantly evolving internet technology and the demands of the users working with the programming language, new features are proposed regularly. To maintain a standardized version of JavaScript across different browser environments, an ECMAScript specification is defined as a standard. The new proposals must run through multiple stages, where they are being explored and evaluated by a committee. After this process, features are added to the ECMAScript specification and can then be provided by the different JavaScript engines.

One of these engines is Graal.js, which is a high-performance JavaScript interpreter implemented in Java, using the Truffle AST framework provided by the GraalVM. The interpreter is highly optimizing by using specialization for optimizing the execution. Furthermore, it also allows interoperability with code of other programming languages supported by the GraalVM like Java, Kotlin and Python.

1.1. Motivation

Right now, JavaScript provides two options for performing operation chaining – nesting and chaining.
Nesting means executing one operation and using the result as a parameter in another operation. This is generally applicable to any type and arbitrary sequence of expressions in JavaScript. The main issue is the readability of nested operations, especially when the nesting gets deep and the numbers of parameters for single operations are high. This can lead to confusions as to which parameters belong to which operation. Moreover, the statement, which was nested, must be read from right to left (innermost-to-outermost) as opposed to the natural reading flow of source code. A very simple example of nesting would be:

```
function1(function2(function3(value)))
```

Chaining means calling a function as a method on a value. Arbitrary many methods can be chained to be executed sequentially. This style is of course much more limited as opposed to the nesting, as first, it can only be used with functions and no other expression types. Furthermore, not even all functions are applicable to a value. Only functions which are defined as methods in the class of the value can be called on it. Therefore, this option is very restricted. Nevertheless, it provides very good readability, as the chained expressions are read from left to right and the parameters for single methods are clearly grouped and unambiguously associated. A simple example for chaining would be:

```
value.function1().function2().function3()
```

In the year 2020, a survey was performed asking people what they feel was missing from JavaScript, the fourth highest answer was a Pipeline operator. [6]
This pipeline operator combines the readability of the chaining style with the broad applicability of the nesting to allow the combination of arbitrary JavaScript expression types. The example from above would look the following using a pipeline operator:

```
value |> function1(%) |> function2(%) |> function3(%)
```

In the current proposal, the % token is used as a placeholder to mark where to use the previous result in the next expression. The |> is the pipeline symbol. The code is very easy to read, and it is immediately clear how the expressions are structured, and the flow of execution can also be seen clearly. [1]

In a real-world application, these pipelines can of course be much larger and more complex. An example for this can be seen in the following code snippet from the proposal. [1]

```
jQuery.merge( this, jQuery.parseHTML(
    match[ 1 ],
     context && context.nodeType ? context.ownerDocument || context :
document,
    true
  ) );

context
    |> (% && %.nodeType ? %.ownerDocument || % : document)
    |> jQuery.parseHTML(match[1], %, true)
    |> jQuery.merge(%);
```

```
console.log(
    chalk.dim(
      `$ ${Object.keys(envars)
        .map(envar => `${envar}=${envars[envar]}`)
        .join(' ')}`,
      'node',
      args.join(' ')
    )
  );

Object.keys(envars)
    .map(envar => `${envar}=${envars[envar]}`)
    .join(' ')
    |> `$ ${%}`
    |> chalk.dim(%, 'node', args.join(' '))
    |> console.log(%);
```

The examples show JavaScript code from the jQuery and React libraries. In the upper half of the code snippet you can see the example without the pipeline and in the lower half the examples are shown with the pipeline. In general, the version with pipeline gives the code better structure and a good overview over the specific operations and their parameters.

1.2. Task

The goal of this thesis is the addition of one of the new ECMAScript proposals as an experimental feature to the Graal.js JavaScript engine – the Pipeline operator. This proposal is currently at Stage 2 / 4 of the proposal process, which means that a draft specification already exists, which should be used to fully implement the feature. Additionally, the implementation should be tested and benchmarked, comparing it to equivalent code without using the Pipeline operator to evaluate the work.

The remaining chapters are structured in the following way: Chapter 2 explains the background of the project and provides general information about the technology used. Chapter 3 explains the architecture of Graal.js and the newly added components during this thesis. The fourth chapter explains the implementation of the Pipeline operator and goes into details about the different approaches taken to solve the task. Chapter 5 shows Technical Data like benchmark results and the interoperability exploration, and the sixth chapter will look at related work and finally the thesis will be finished with a conclusion.

2. Foundations

In this chapter, basic concepts related to this thesis' work are explained as well as the existing foundations needed. At first, JavaScript and the ECMAScript proposal process is shortly explained, followed by a short introduction to the GraalVM and the Truffle framework. Lastly, basics of the Graal.js interpreter are introduced.

2.1. JavaScript

JavaScript is the go-to programming language for web-based applications. It is a lightweight, single-threaded, prototype-based, dynamic language, which exists since 1995, when it was released as a proprietary software at Netscape. Later in 1997, the ECMAScript standard has been introduced to get a standardized specification for scripting languages for and ensure interoperability between different web browsers.
Since then, new versions of this standard have been released continuously and currently, the 15th version is the most recent. While it was intended to be a scripting language, it became a general-purpose language due to its variety of usages in all contexts.[18] [19]

At first, JavaScript was a purely interpreted language. In later versions, Just-in-Time compilation was added to improve performance. The language allowed websites to become dynamic, as it could manipulate the DOM (Document Object model).[18]
JavaScript is not self-sufficient. It needs a host-environment to operate in, which provides objects to be manipulated and input and output, as they are not covered in the specification. An example of such a host environment is a web browser for client-side computations. The objects it provides are e.g., windows and text boxes. Furthermore, scripts can be attached to events to be executed once they occur. As those objects differ between various environments, the ECMAScript language is specified independent from them. [19]

Like Ruby or Python, JavaScript is a dynamically typed programming language.
This means that data types of variables are not constant, but instead can change during runtime. It allows reusing existing variables and assigning different values to it, like a String to a variable holding an Integer value, e.g., which is not possible in many other languages. Datatypes defined for JavaScript are Numbers, String, Boolean, Null, Undefined, bigint, Symbol and Object. [20]

Objects are the most important data type in JavaScript, as basically all non-primitive elements are objects (e.g. functions, arrays and so on). They enable users to also use some Object-Oriented paradigms when programming JavaScript. A speciality of JavaScript is the absence of classes. While classes are fundamental in most object-oriented programming languages as abstract models for objects, JavaScript uses prototypes. This way, only a constructor function needs to be specified and via the prototype property each function automatically gets, new fields and methods can be added to the function later in the code and will be inherited by all objects using it. Since ECMAScript 2015, a class syntax has been introduced, however it is mostly based on prototypes as well and mainly focuses on making inheritance and object creation easier syntactically. [21]

To run JavaScript in the browser, a JavaScript engine is necessary to translate the code to machine code. Nowadays, there is a wide variety of different JavaScript engines available, some of the most well-known are the V8 engine by Google, Microsoft's Chakra engine and SpiderMonkey, which was the first JavaScript engine to exist. The focus in this thesis however lies on the Graal.js engine of the GraalVM project.

2.2. ECMAScript proposal process

JavaScript is extended by a proposal process. Every new feature, which should be added to the language eventually, must run through five stages to be accepted as a new addition to the ECMAScript specification. In these stages, the proposals are presented to a committee, which then decides, whether this proposal is worth exploring. The stages are structured in the following way:

- **Stage 0 (Strawperson):**
  Proposals in this stage are either yet to be presented to the committee or they have not fulfilled certain criteria to be promoted to Stage 1.
- **Stage 1 (Proposal):**
  At this stage, the committee wants to develop solutions for this problem and problem and solution descriptions are created and reviewed.
- **Stage 2 (Draft):**
  In this Stage, a formal language specification exists, describing the syntax and semantics of the proposed feature, which should be developed and finally added to the ECMAScript specification.
- **Stage 3 (Candidate):**
  At this stage, the intended solution is complete and external feedback or implementation experience is necessary to proceed.
- **Stage 4 (Finished):**
  The proposal can be added to the standard at any time.

Additionally, there is also an Inactive stage for proposals, that were withdrawn or rejected by the committee. [2]

2.3. GraalVM

The GraalVM is a layered approach to achieve high-performance when executing a wide range of different heterogeneous languages by a common framework, which allows reuse of multiple optimizing concepts.

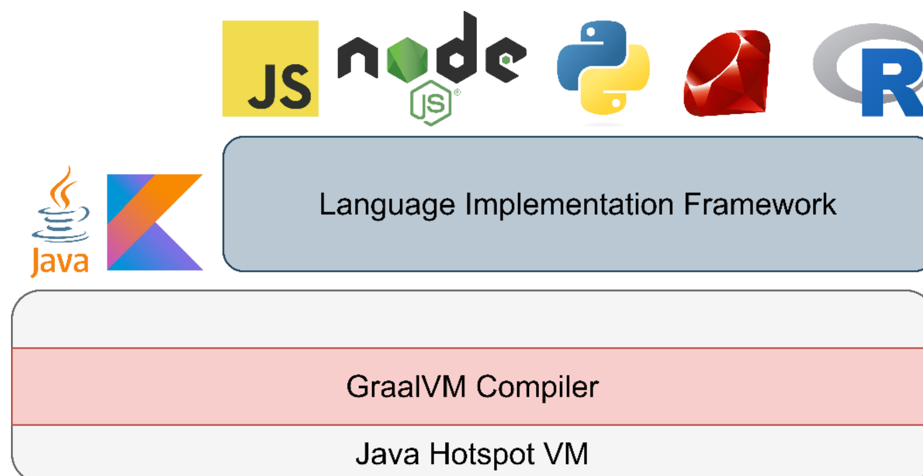


Figure 2-1: GraalVM architecture overview [1]

As visible in Figure 2-1, the Java HotSpot VM is the bottom layer of the architecture. On top of that is the Graal compiler, which is a highly optimizing compiler written in Java. To achieve high performance, it uses partial compilation with method inlining, specializations, assumptions, and other profiling information. This compiler can directly be used by languages like, e.g., Java or Kotlin, which are native to the JVM. To also execute other languages on the GraalVM, another layer in the architecture is needed. This layer is the Language Implementation Framework called Truffle, which enables developers to write their own Abstract Syntax Tree Interpreter of the desired guest language. In Figure 2-1, some of the guest languages are depicted, for which such interpreters already exist. [3]

2.4. Truffle

The Truffle framework[1] is an essential part of the GraalVM. It is an open-source library for implementing programming languages as Abstract Syntax Tree interpreters in Java and provides native performance, the ability to integrate the language with all other Truffle languages and tool support. [4]

This approach uses specialization information like type information when interpreting code to rewrite the AST. After the AST is seen as stable, partial compilation is used to produce optimized machine code for the specialized AST parts. Should some specialization not hold afterwards, deoptimization is performed. In this step, the compiled machine code is invalidated, and execution is transferred back to the interpreter, which can then use the newly gathered information to recompile the code again with partial evaluation. [3]

Truffle allows developers to write their own self-optimizing AST interpreters for any programming language of their choice in Java. Würthinger et al. [10] present the approach, where an Abstract Syntax Tree can be modified during interpretation by replacing a node in the tree with another node. This allows optimizing dynamic language constructs.

The approach uses profiling feedback to specialize nodes. At first, a general node is executed in the AST. After incorporating profiling data, this node is replaced by a more specific node, which performs the operation faster for the current operands. However, this node is not able to handle all cases. Should later profiling prove one of the assumptions taken wrong, the node is again replaced with a generic one performing slower but covering different variants.

[1] https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/ (Last visited: 21st June 2023)

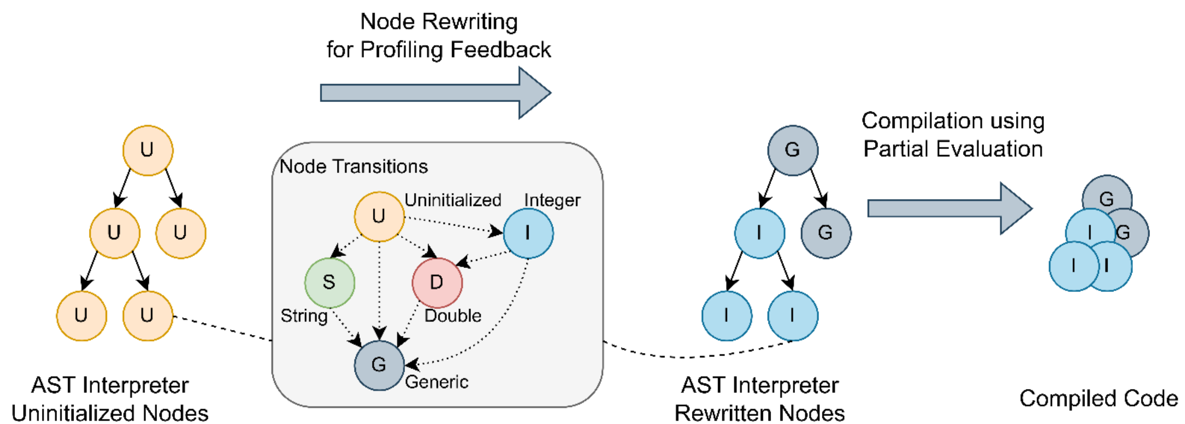


Figure 2-2: Node rewriting and partial evaluation using Truffle [2]

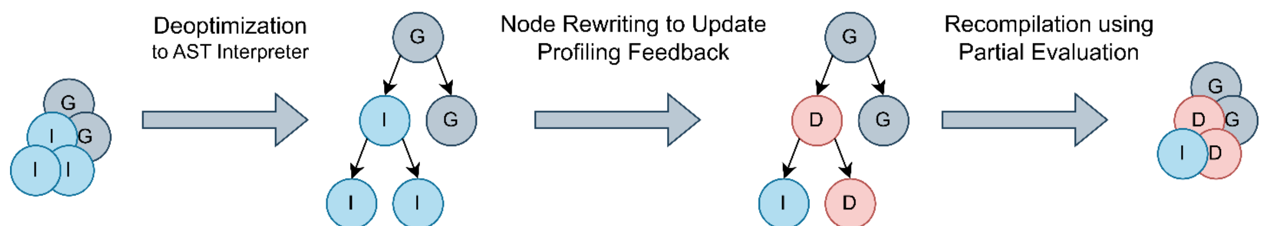


Figure 2-3: Deoptimization, node rewriting and recompilation using Truffle [2]

In the figures above, an example is shown for the Truffle partial compilation and deoptimization parts. For this example, a simple addition is performed. At first, all nodes of the AST are uninitialized. After executing the nodes for the first time, the nodes profile to Integer addition. With this information, the respective tree nodes can be rewritten to Integer nodes and the rest are generic nodes. This tree is then compiled using partial evaluation to optimize the execution for this case. [3]
If one of the assumptions does not hold, the execution is transferred back to the interpreter and the machine code is discarded. Again, profiling is used to rewrite the nodes, which were violating the previous speculations. In the example above, one of the summands is now a double instead of an integer, which means that also the addition itself must be changed to an Addition node for doubles. After the new profiling feedback is incorporated, the code is again compiled aggressively with partial evaluation. [3]

JOHANNES KEPLER
UNIVERSITY LINZ

2.5. Graal.js

Graal.js[2] is a JavaScript implementation written in Java, which uses Truffle and the GraalVM compiler to achieve high performance. It is fully compliant with the ECMAScript standard specification. Due to using the GraalVM architecture, it is also interoperable with all other Truffle languages and supports tooling. [5]

A detailed description about the architecture and components of Graal.js can be read in Chapter 3.

[2] https://github.com/oracle/graaljs/tree/master/graal-js (Last visited: 21st June 2023)

3. Architecture

Graal.js has several components, the work of this thesis was done in two of them: the Parser and the runtime system. Therefore, the focus of this chapter lays in explaining these parts of Graal.js in more detail as well as the Pipeline syntax specifications.

As Graal.js is a very large project with many different components, the focus of this chapter lies on the parts worked on during this thesis. The Parser was the part of the architecture, where most of the work of this thesis happened. The runtime system is the component, which provides the Truffle AST nodes and implementations for all JavaScript language elements.
A diagram of the components and interaction between them is shown in Figure 3-1.

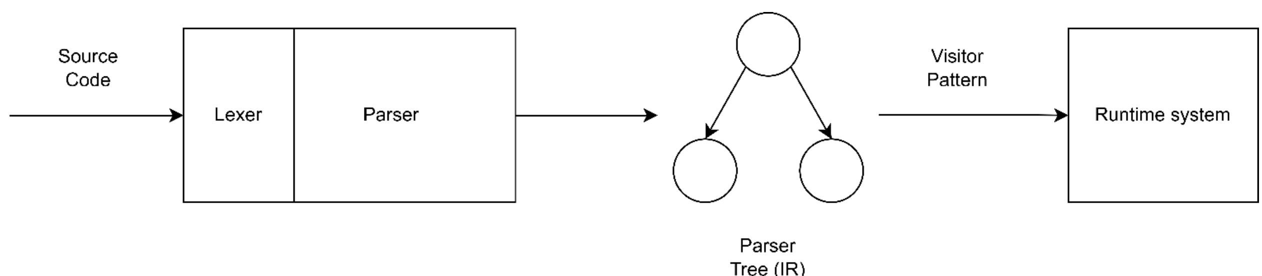


Figure 3-1: Simplified Architecture of Graal.js

As shown in Figure 3-1, the Source code is first processed by a Lexer and a Parser. The Lexer scans the code and creates a Token Stream. This stream is then processed by the Parser to create the Intermediate Representation as well as check the code for errors and validity and establish contextual and semantical dependencies. This Intermediate Representation is a Parser Tree, where dependent nodes are connected. After the parsing process is finished, the Parser tree is traversed by a Visitor. Whenever the visitor visits a node, the node is transformed into a Truffle AST node.

3.1. Nodes

Graal.js uses the Truffle framework to interpret JavaScript code. This framework allows writing AST interpreters for any programming language. In Graal.js, the AST nodes are all specified in the runtime system. The abstract base node, JavaScriptBaseNode, is the root of all other implemented nodes. Starting from this, nodes are implemented for each operation and language element existing in JavaScript. Some examples for Nodes are ArrayCreationNode for creating an array, FunctionCallNode for function calls or nodes for unary and binary operations like the JSNotNode or the JSAddNode and many more.

In each node, execute methods are defined to correctly execute the JavaScript code in Java. Specialization is used to provide more efficient and performant execution.
For this, partial evaluation is performed by speculating on types and values of variables. Using these speculations, the code is compiled and executed. Should a speculation fail, the call transferToInterpreter is performed and execution is continued in the interpreter.
The profiling feedback is then used to update the specializations and the code is compiled again using partial evaluation.

3.2. Additions to the architecture

In the final approach, only additions to the Parser component are made to realize the Pipeline operator as an experimental feature. All other components are left unchanged. In the first approaches, the runtime system was also changed, but these changes were removed again later, as they were not used anymore.

The changes required in the Parser were additions to two of the Parser methods, where the expressions are parsed. Temporal variables are created in the process of parsing a pipeline expression, which are always in the scope of a function. Additionally, the pipeline token had to be added to be scanned correctly by the Lexer.

3.3. Pipeline operator syntax

For indicating the pipeline operator, the token |> is currently used in the specification. This will most likely also be the final choice. The grammar of the pipeline syntax looks as following:

Left expression |> right expression { |> expression }

The pipeline operator (|>) is an infix operator. If the pipeline operator is only used once in a pipeline, the left expression must return a result, otherwise the placeholder token (%) cannot be bound to a value. The right expression on the other hand can be any expression type without restrictions. When chaining multiple pipeline operations, all expressions except the right-most need to return a result.

Expressions, which should be compatible with pipelines are:

- (Unary) function calls
- Method calls
- Arithmetic operations
- Array literals
- Object literals
- Template literals
- Function literals
- Object constructions
- Await expressions
- Yield expressions
- Import statements

For using these expressions in combination with the pipeline syntax, there are some restrictions:

- **await** can only be used inside of an **async** function body.
- **Yield** can only be used inside a generator function.
- **import** can only be used in modules.

All other expression types mentioned above can be used without any restrictions.

JOHANNES KEPLER
UNIVERSITY LINZ

3.3.1. Topic reference

After executing the left expression, the result needs to be stored for using it in the right expression. The place, where a user wants to use the previous result, needs to be marked somehow. For that, a placeholder token (topic reference token) indicates the location. Right now, the specification defines the **%** token as a placeholder for the result, although this may change in later versions as the proposal progresses. If the placeholder token is not used in a right expression, the syntax is not valid and will throw an error message.

At execution time, when the left side execution finishes, the result is bound to the topic reference and when executing the right expression, the topic reference token evaluates to the saved result.

In this approach, the topic binding is implemented using temporal variables, which exist in the background invisible for the user. They are created when the left side is executed and after the pipeline statement has been executed, they are again removed. More on the implementation details can be read in Chapter 4.

3.3.2. Examples

In this section, some very simple and trivial examples explain the usage of the pipeline syntax in different settings.

A basic example of a pipeline expression is the following:

```
5 |> double(%);
```

In this code sample, the left expression is a simple Integer literal (5) and the right side is a function call (double), which takes one parameter and returns the parameter times two as a result. When executing this statement, the left side of the pipeline is evaluated first. The result 5 is then bound to the topic reference (%). Then, the right expression can be executed, where % is resolved with the value 5, which means the result of the whole statement is **double(5)**.

The topic reference token can also be used more than once in a right expression. An example for this:

```
(1 + 4)*3 |> calculateArea(%, %);
```

Here, the left expression is an arithmetic operation adding 4 + 1 and then multiplying the result with 3. The result of this calculation should then be used for calculating the area with the function calculateArea, which takes two parameters for the width and height of an object. In this example, both parameters are the topic reference, which means the actual call at runtime will be **calculateArea(15, 15)**. When using the same topic reference token more than once in the same pipebody, all the placeholders evaluate to the identical result.

Furthermore, chaining arbitrarily many pipeline expressions is possible. A simple example for this is:

```
3 + 4 |> new Rectangle(6, %) |> %.calcArea();
```

In this example, the first expression performs an addition. The result is then used as a parameter for constructing a Rectangle object. In the right most expression, the method **calcArea()** is called on the Rectangle and calculates its area.

For chaining pipelines, the execution order is always form left to right, so **3 + 4** is evaluated first. The result – 7 – is then bound to the topic reference and the next expression, **new Rectangle(6, 7)** is evaluated and the created object is bound to the topic reference and finally **calcArea()** is called on the object.
Of course, more pipeline operators can be added and more operations in a single pipeline performed.

4. Implementation

This chapter explains the implementation details and different approaches taken during this thesis.

4.1. Approaches

To realize the pipeline operator in Graal.js, three different approaches were taken. While in the first two approaches, changes in the Parser as well as the runtime system were necessary, for the third, current and final approach, only additions to the Parser are required.

4.1.1. First approach

In the first approach, the idea was to always treat the pipeline operation as a binary expression.
This expression has two sub-expressions, the left-side expression of the pipeline operator and the right-side expression. If multiple pipelines are chained, the right-side expression is again a binary pipeline expression and if another pipeline occurs, it is again the right child and so on.
To evaluate the pipeline at runtime, the goal was to add a new Truffle AST Pipeline node in the runtime system, which performs the topic binding and execution in its execute method. The topic binding should be resolved by using temporal variables, where the result of the left expression is saved and then inserted into the right expression.

First, the new operator **|>** was added to the TokenType enumeration as a binary operator with the same precedence as the function arrow, the assignment operators, and the generator operators, as described in the specification.
Then code to parse the pipelines was added to the Parser class. The primaryExpression method was altered to parse the % token as an identifier, if it occurs at a position, where it is semantically invalid to be a modulo operator.

In that case, an identifier reference is created with the ident name “%pipeDepth”, where pipeDepth is the nesting depth. For the first pipeline body, it is 1, 2 for the second and so on. When no pipeline is currently parsed, pipeDepth is 0 and an error is thrown if it is attempted to use % as an ident.

The parsing of the pipeline itself is done in the expression method, where all kinds of expressions are parsed in a loop. First, the left side expression is parsed. If the operator is a pipeline token, the pipeDepth variable gets incremented by 1, then the right side is parsed in another loop with a recursive call on the expression function. After both subexpressions are gathered, the pipeDepth is decremented. A check is performed if the topic reference token is used in the right subexpression. For this, a new PipelineContext is introduced, which contains the pipeDepth and a Boolean to indicate, whether the topic reference was parsed for this pipe body. If this is not the case, a syntax error is thrown. Otherwise, the method newBinaryExpression is called to create a Parser node of the pipeline expression.
In this method, an additional clause is added to create a Pipeline node, should the operator be a pipeline. For this to work, a PipelineNode was implemented for the intermediate representation. This node was modelled as a child node of the BinaryNode. It has two children, the left and right subexpression and additionally saves the level of pipeDepth for assigning the topic reference to the correct placeholder identifier.

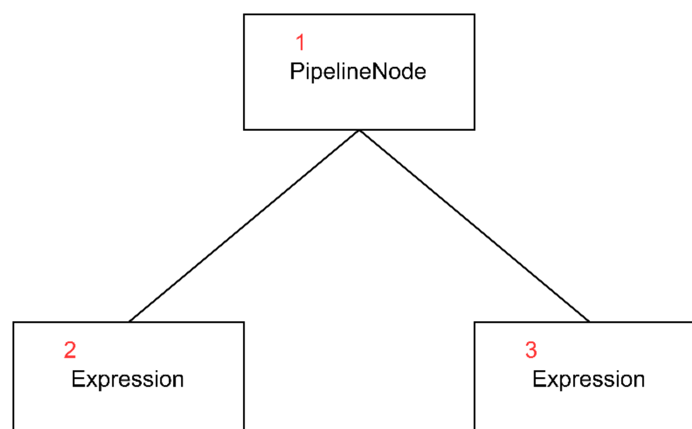


Figure 4-1: AST of a pipeline expression with execution order

In Figure 4-1, the abstract structure of the AST of a pipeline is depicted. The red numbers indicate the order of evaluation/execution. First, the execute method of the PipelineNode is entered. Second, the left sub expression is evaluated and third, the right sub expression is evaluated in this approach.

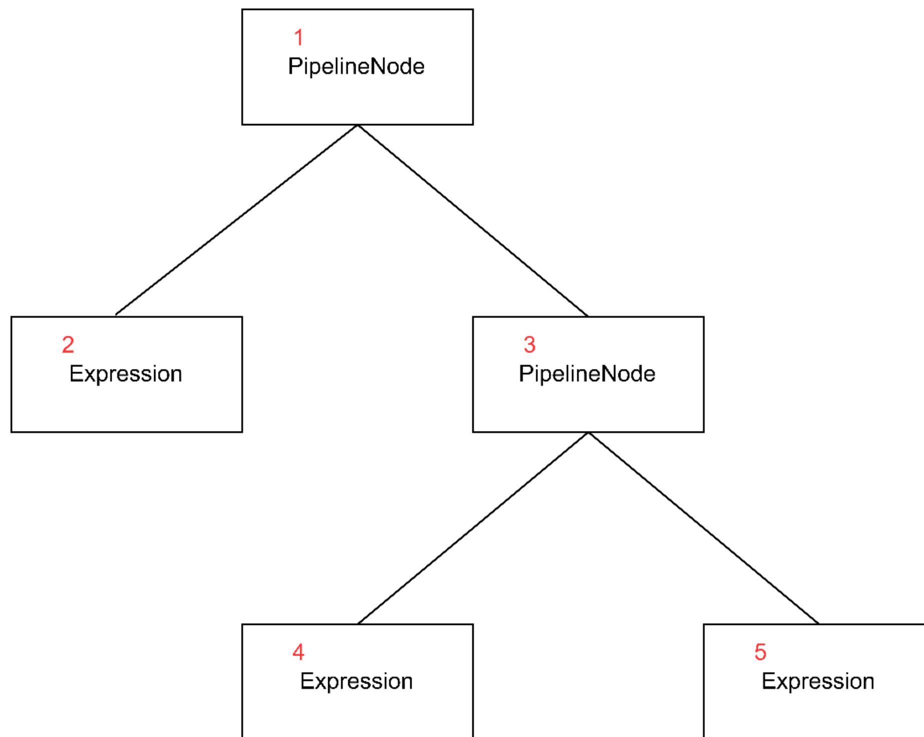


Figure 4-2: AST of a nested pipeline expression

In Figure 4-2, the abstract structure of the AST of a nested pipeline is shown. The order, in which the execute methods of the AST nodes are called is again indicated by the red numbers.

A very simple, concrete example how the Parser tree of a pipeline looks can be seen in Figure 4-3. In this example, the expression **1 + 2 |> % + 4** is parsed. The Pipeline node is the root node, and it has two children, which are both addition nodes. These have again two children, the summands. For the left child, they are two numbers, 1 and 2 and for the right node we have 4 and the topic reference token.

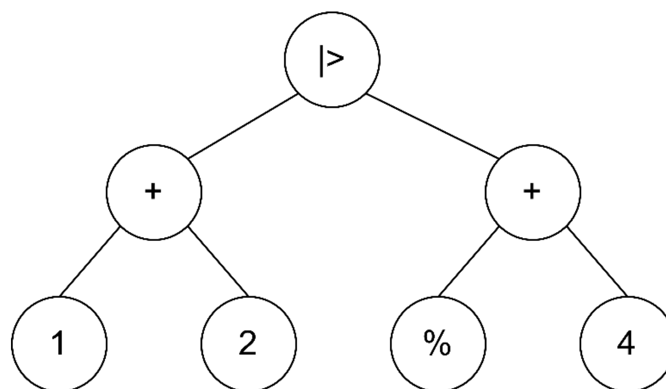


Figure 4-3: Parser Tree of the expression 1 + 2 |> % + 4

Figure 4-4 shows an example Parser tree of a pipeline chain. The expression parsed is **ints[1] |> multiply(%, 3) |> 100 - %.** The first node is an Index node for a simple array access at a position, with two leaf nodes, the array name, and the index to be accessed. This is the left child of a Pipeline node. Next, the multiply(%, 3) function call is processed and a CallNode is created with the function name and its parameters % and 3 as children. Next, another pipeline operator occurs, so the CallNode becomes the left child of a second Pipeline node. In the last step, parsing the final subexpression, 100 - %, creates a Subtraction node with the minuends as child nodes.
As no more pipeline operators occur after that, the Subtraction node is the right child node of the second Pipeline node and this in turn is the right child of the first pipeline.

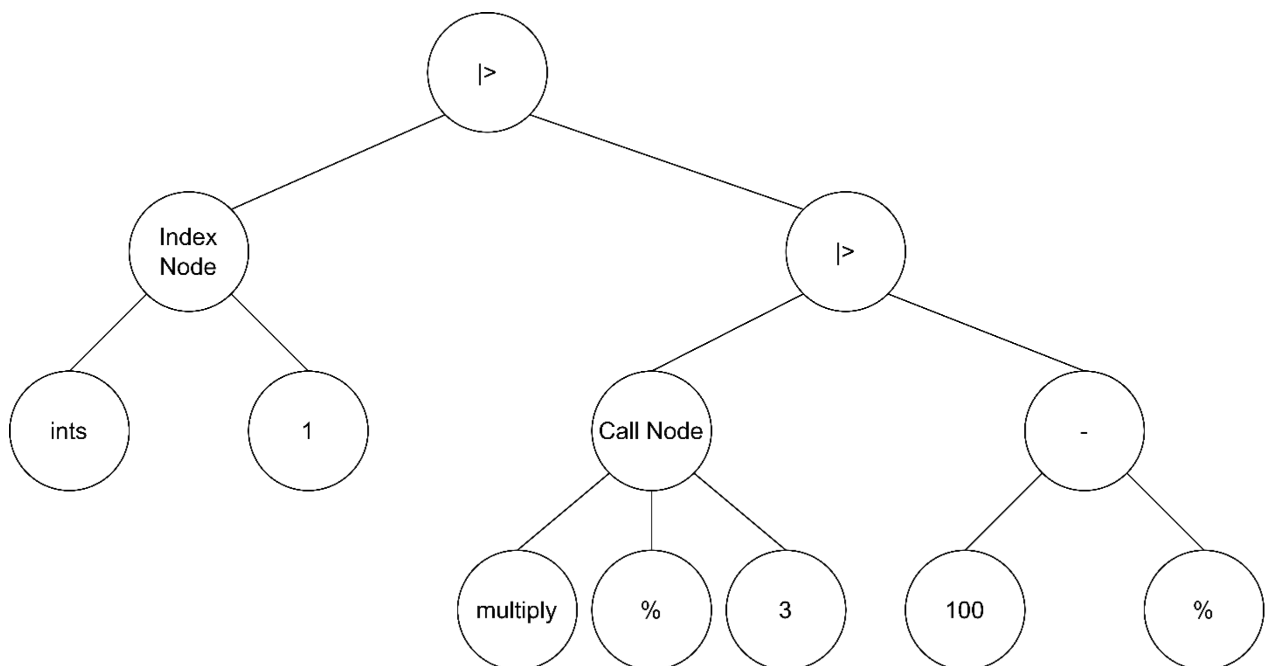


Figure 4-4: Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - %

For this approach, whenever multiple pipeline operators are chained, the one occurring first is always the top node and the next one is the right child of that node. Until no more pipelines occur in the statement, the right child is always another pipeline node.

After the parsing is finished, the Pipeline node of the Parser tree needs to be transformed to a Pipeline AST node once visited by the Visitor. The visitor is the class **GraalJSTranslator**. In this class, the transformations for all existing Parser nodes to Truffle nodes are implemented.

In the case of the PipelineNode, the binary node transformation methods, **enterBinaryNode** and **enterBinaryExpressionNode,** were reused. The only changes necessary were to add the Pipeline operator to the switch statement to enter the second method from above.

After coming to this point, some issues with this approach came to light. The main issue was the order of the Parser nodes. So, the approach was changed slightly, which will be described in the next subchapter.

4.1.2. Second approach

In the second approach, most of the parts from the first approach were kept. The main change was the grouping of the Parser nodes.

For a single Pipeline without chaining multiple pipeline operators, also nothing changes. There is a left subexpression and a right subexpression. The order only changes for pipeline chaining. In this case, where now the Pipeline node always becomes the left child of the next Pipeline node as opposed to before, where the first node was the outermost and following Pipeline nodes became the right children of the previous pipe.

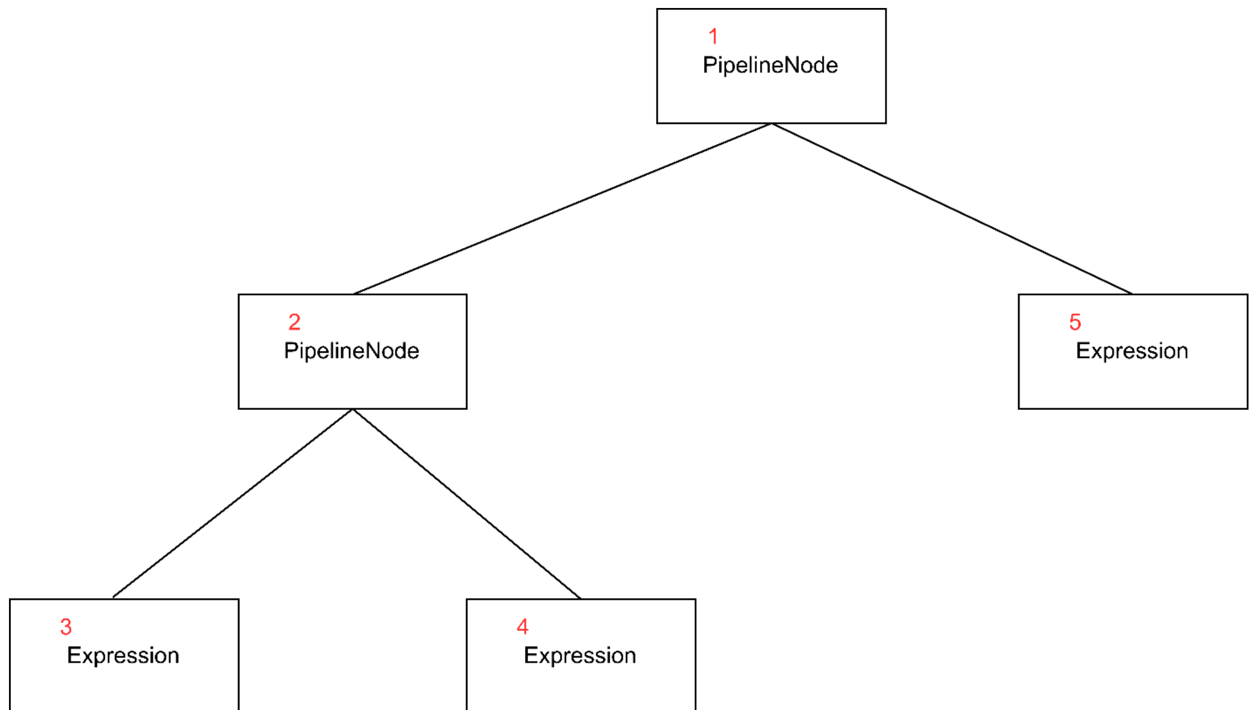


Figure 4-5: Generic AST of a chained pipeline for Approach 2

In Figure 4-5, the abstract structure of the AST of a chained pipeline expression is shown with the order of execution stated in red numbers. The top PipelineNode's execute method is called first. Next, the second PipelineNode's execute method is entered and then the left subexpression of this node is executed, which can again be a PipelineNode. After that, the right subexpression of the leftmost node is executed. Finally, when all the left side is executed, the right subexpression of the topmost PipelineNode is evaluated and the result is returned.

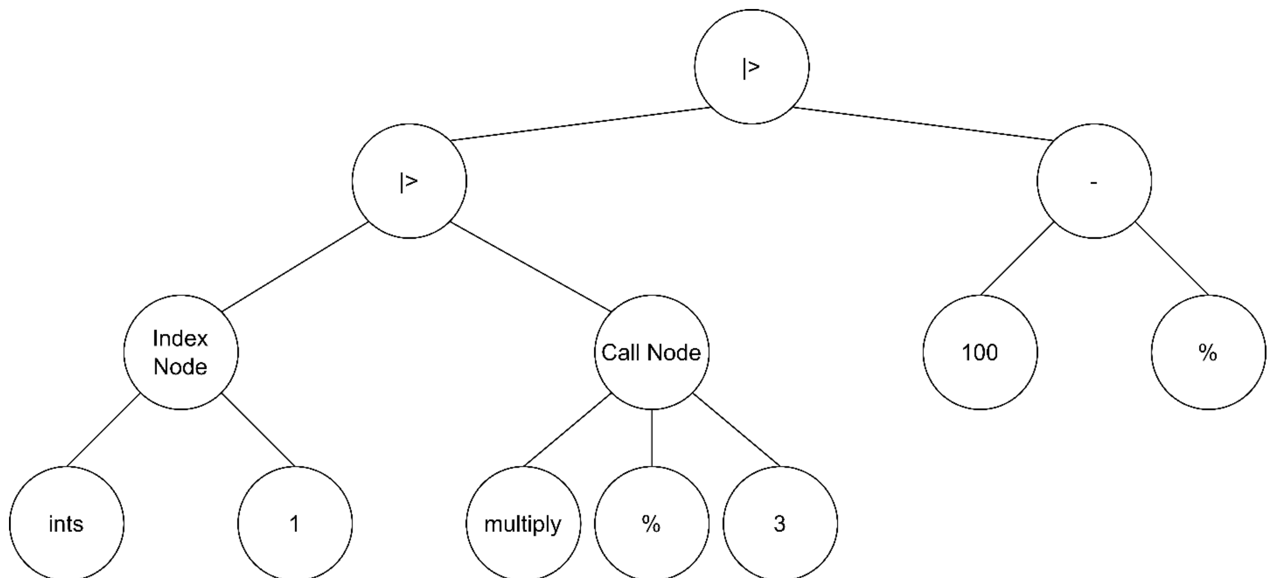


Figure 4-6: Revised Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - %

A concrete example for this can be seen in Figure 4-6. The same example expression **ints[1] |> multiply(%, 3) |> 100 - %** is used as in the previous section 4.1.1. However, in this case, after the IndexNode and the CallNode are parsed, they become the left and right child of the first Pipeline Node. After encountering another Pipeline operator, the first Pipeline node becomes the left child of the second Pipeline node. The Subtraction node is then the right child of this Pipeline node.

After performing this change and checking, whether the parsing process works correctly, the next step was to implement the Pipeline Truffle node. Here the execution of the pipeline at runtime needs to be specified. The basic idea consisted of three steps:

1. Execute the left subexpression and save the result.
2. Create the topic binding by assigning the saved result to a temporal variable.
3. Execute the right-hand side using the topic binding to resolve the usages of the topic reference.

To implement all this, an execute method was used, which all Truffle nodes have. The first step from above is rather straightforward to realize by calling the execute method of the left subexpression and then assigning the result to a local Object variable.

The second part was the one where new issues appeared. After saving the result of the left side, accessing the placeholder identifiers, and replacing them with the result turned out to be a bigger challenge than initially thought. In the end, no feasible solution for this problem was found, which led to the third and final approach of this project.

4.1.3. Third approach

In the third approach, a different angle to solve the task was taken. Instead of trying to add new Truffle AST nodes and Parser nodes to get the desired outcome, existing nodes were reused, and the pipeline syntax was “desugared” to fit into those structures. This resulted in no need to make changes/additions in the runtime system, but only in the Parser. The new nodes created in the previous approaches were removed.

Parsing the pipeline syntax is moved to the method **assignmentExpression**, as the pipeline operator has the same precedence as the assignment operators. A new if-clause is added to parse the pipelines.

The main change is the topic binding now happens in the Parser as opposed to the previous approaches, where it was done at runtime. In the first step, the left side expression is parsed. After that, if a pipeline operator occurs, a BinaryNode assigns this expression to the placeholder identifier. This is an internal assignment not visible to the user only for the purpose of creating the topic binding. Next, the right-side expression is parsed by a recursive call of the **assignmentExpression** method. In the final step, the internal assignment and the right side are combined by creating a **CommaRightNode.** This node takes two expressions and first executes one – here the assignment - and after that the other – the right-side expression.

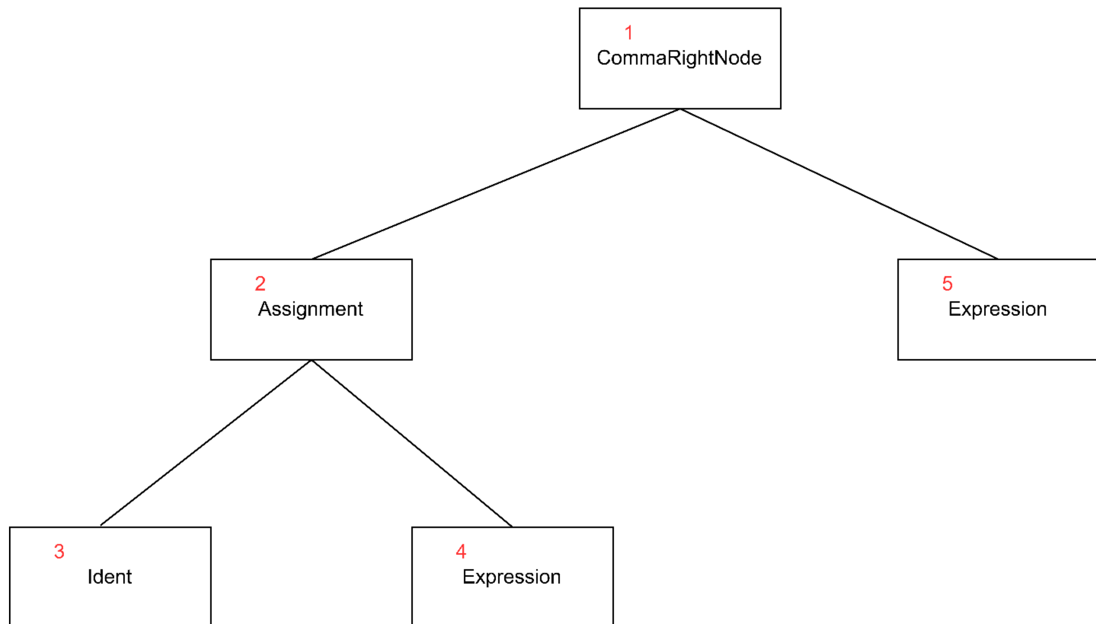


Figure 4-7: AST for the third approach with CommaRightNode

In Figure 4-7, the AST structure of a single pipeline expression is shown. The order of execution in this case again starts with the top-most node of the tree, which is the CommaRightNode. Next, the Assignment node is entered, which executes the IdentNode and the expression, which is the left side of the pipeline. Lastly, the right-side expression is executed. This expression can be of any kind, including another pipeline, in which case the execution order shown in the picture above will be performed again.

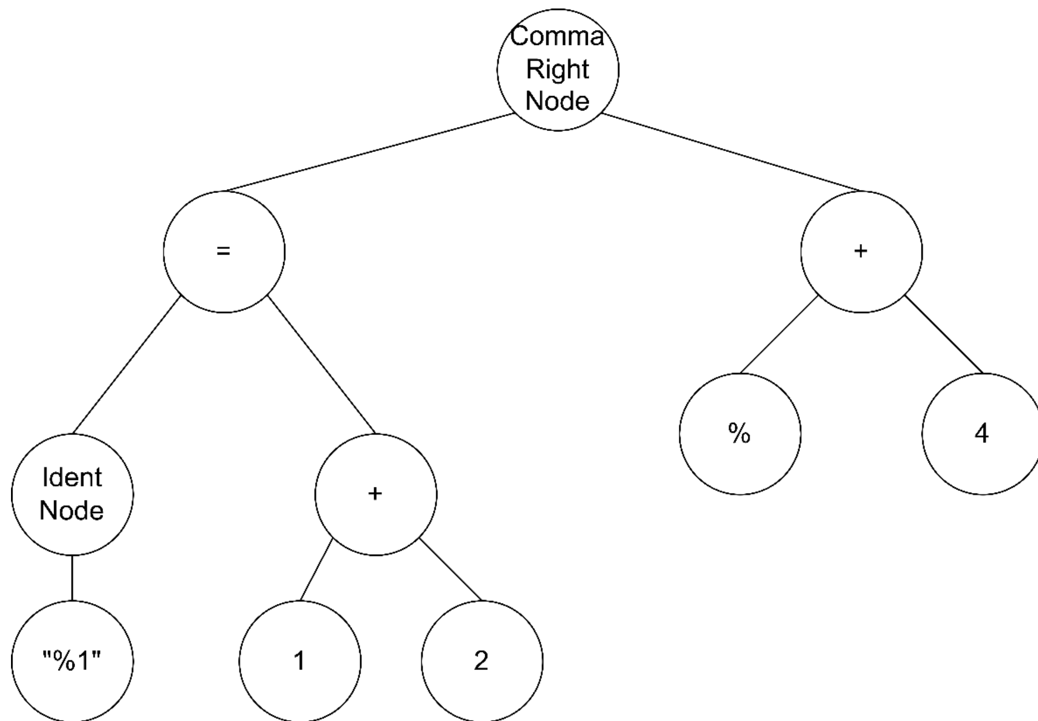


Figure 4-8: Parser Tree of of the expression 1 + 2 |> % + 4 with third approach

In Figure 4-8, the Parser tree of the example from the previous approaches is shown. The expression **1 + 2 |> % + 4** is parsed in the following way:
First, the left subexpression, 1 + 2, is parsed. In the next step, the topic binding assignment is inserted by creating the identifier "%1" and assigning the left expression to it by creating an AssignmentNode. Next, the right expression - % + 4 - is parsed, and finally both the topic binding and the right side are combined in a CommaRightNode.

As can already be seen from this example, the pipeline tokens are completely removed in the Parser and therefore no new transformations have to be specified.

For chained pipelines, all expressions except the right most need to be assigned to a placeholder identifier, which means the pipeline is internally transformed into a chain of assignments. To show this based on a very simple example again:

Before:

```
ints[1] |> multiply(%, 3) |> 100 - %
```

After:

```
%1 = ints[1]; %2 = multiply(%1, 3); 100 - %2
```

This code sample shows the same pipeline chain as in previous sections as a very simple example for pipeline chaining. First, **ints[1]** is parsed and then assigned to %1, which is then used in **multiply(%1, 3).** In turn, this expression is assigned to %2. The final expression, **100 - %2** provides the result of the whole pipeline expression. In Figure 4-9, the Parser tree for this example is shown as well.

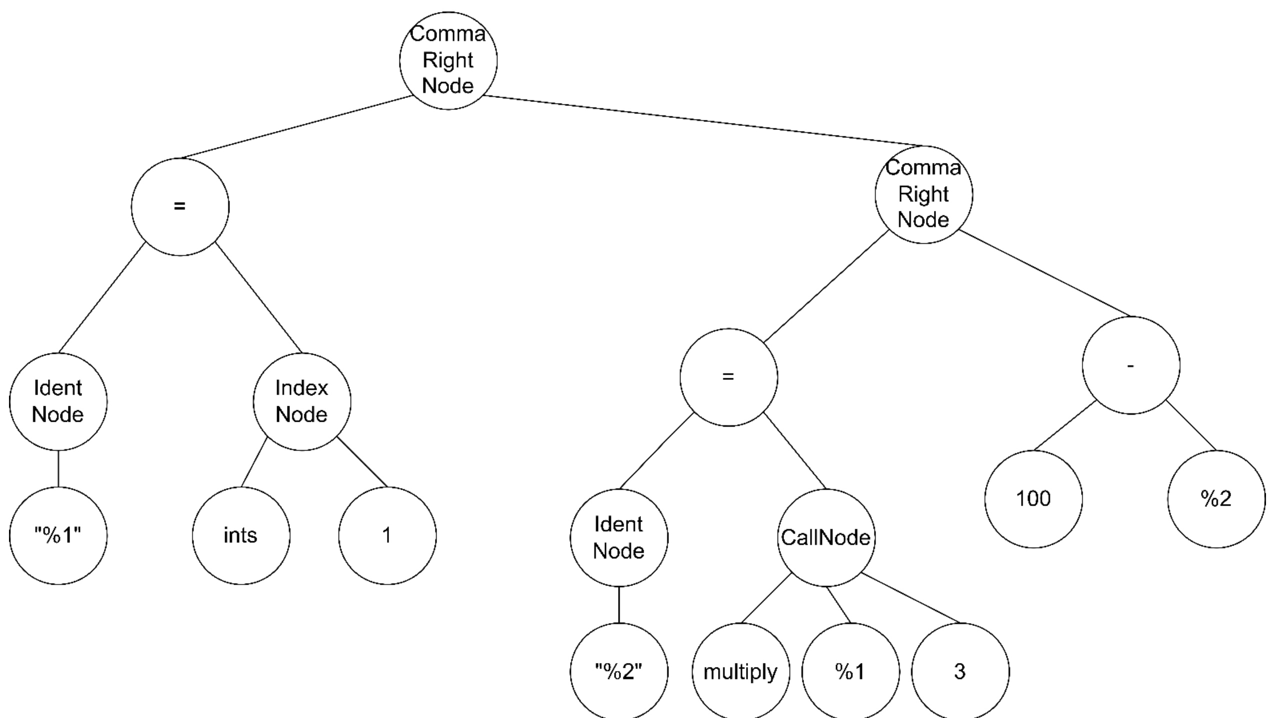


Figure 4-9: Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - % for third approach

Additionally, the pipeDepth variable is moved to a function context object, which means there is one variable per function. This is necessary to ensure that also pipelines inside function literals in a pipeline chain are parsed correctly. Two simple methods are added to increase and decrease the pipeDepth variable as well.

A special case, which occurs through this approach and needs to be handled explicitly, is using the pipeline syntax in JavaScript's strict mode. This mode forces users to write better and more correct code, as mistakes, that would not be thrown as errors from normal JavaScript are treated as such in strict mode. One example of such an error is using a variable without declaring it first.

This is also the reason, why some additional details must be done for this pipeline syntax to work in strict mode, as the current implementation uses undeclared variables as placeholders to save the result of the left-side expression of the pipeline. For this approach to also work in strict mode, an if-clause is added to check if this code should be executed in strict mode. If this is the case, a VarNode needs to be created to declare the placeholder variable with a let-statement. With this, the method declareVar can be called and therefore it is indicated, that the topic reference variable is indeed declared in the current scope. This step is especially necessary for enabling the usage of the pipeline in modules, as they are executed in strict mode per default.

4.2. Error handling

Two new possible syntax errors need to be handled due to introducing the Pipeline.
The first error that can occur is not using the topic reference token (%) in a pipe body.
An example of this error is:

```
7*7 |> square(3);
```

In this example, a syntax error will occur, as the topic reference is never used in the right expression. That is an error, as without the topic reference, the left expression will be executed, but never used afterwards and the result is just thrown away.

Implementing the detection of this error required the introduction of a new Boolean variable in the Parser called **topicReferenceUsed.** This variable is set to false each time the Parser encounters a **|>**-token. It is set to true once the first topic reference token is parsed after a |> token and inside the pipe body. Every further occurrence of the topic reference on the right side is not relevant for this check. Once the right expression is parsed, the variable is checked. If it is still false, the error message “Pipe body must contain the topic reference token(%) at least once” is thrown as a Syntax error.

Furthermore, to also check this error for a chained pipeline, a local variable is introduced to the method, in which the pipeline is parsed. As the right side of the pipeline is parsed recursively, the check always happens, after the right side was processed. In the case of pipeline chaining, each time another pipeline operator occurs, the **topicReferenceUsed** variable is reset to false. To maintain the correct Boolean values for each pipe body, the local variable **prevRef** saves the previous value of the **topicReferenceUsed** Boolean to restore them at the end of the method before returning the parsed expression.

This step is necessary to not miss any potential errors as otherwise the variable will always have the value of the last right side and all other checks will have the same result.

The second error is using the topic reference token outside a pipeline body. This error is checked in the **primaryExpression** method, where the topic reference token is parsed. A simple example for this error kind is:

```
let sum = % + 10;
```

To detect this error, the check pipeDepth <= 0 is performed. If this check evaluates to true, a syntax error with the message “The topic reference cannot be used here!” is thrown.

4.3. Tests

To validate the implementation, the task for me was to write test cases to get a good coverage. For this, test cases for the different use cases of the pipeline were written. The first part tested using a Pipeline in combination with all possible expression types on the right side. On the left side different types of expressions are used to also test their compatibility with the pipeline. Additionally, test cases for pipeline chains of different depth were added to test, that the chains work correctly and some special interesting corner cases.

Furthermore, to test some of the expression types described in Chapter 3.3, it was necessary to implement basic constructs to properly test them. These constructs were a unary function (function with one parameter), a function with two parameters and a class for testing method calls and construction using the pipeline. Moreover, a function returning a Promise as well as an asynchronous function for testing awaiting a Promise and a generator function to test the use of yield in the context of a pipeline.

Most of the expression type test cases are kept very simple by performing a calculation, a function call or an array access on the left side and using the result in the different expression types with the topic reference placeholder. To verify the result of the operation, the actual outcome is compared to the expected for equality.

In the second part of the tests, different sizes of pipeline chains were tested. The number of pipeline operators of the statement is always increased by one by appending some step at the end. Chains from two to 6 sequential pipeline operators were tested in the end. In one of the statements an interesting corner case was tested, where one pipebody statement consists of only topic reference /modulo tokens(%) as in the following example.

```
const example = 15*3 |> % % %;
```

It is necessary to test for this constellation as well even though it is not really a statement that will appear in actual code as the result of this operation will always be one. All the test cases passed in the end, which means that the implementation handles the use cases described above correctly.

5. Technical Data

In this chapter, technical aspects of the implementation of the pipeline operator are shown in detail like benchmark results and Truffle interoperability tests.

5.1. Benchmarks

For benchmarking, there are several aspects, which should be benchmarked during this thesis.

1. Comparing the pipeline syntax to equivalent code without using a pipeline
2. Comparing the pipeline implementation to the implementation of other engines

For point 1, two aspects were compared. The first being the different expression types, which can be used in a pipeline and secondly chaining together multiple pipelines (2 up to 6 chained pipeline operators). It is also distinguished between running Graal.js as a plain Java program and using the Graal compiler for additional optimizations through partial compilation.

For the second point, as only Babel.js currently provides support for the pipeline operator and no other engines does so at the point of this thesis, the implementation in Graal.js was compared to the Babel.js pipeline support, although Babel simply transforms the pipeline to older JavaScript versions.

A small and simple JavaScript file was written for benchmarking, which uses two nested loops. The first loop is used to setup each run and to take a starting and end time and calculate the resulting measured time for each execution batch. It is repeated **x** times. The inner loop simply executes the operation to be benchmarked repeatedly. After this loop is finished, the time is again measured, and the time taken to execute the operation **n** times and an average for a per operation execution time are calculated. The measured times of each loop run are also summed up to calculate an overall average after all runs are completed.

Additionally, a warmup run is performed, whose statistics do not count towards the result. This is done to remove some noise and not have some outliers. For the parameters n and x from above the values 5 million (n) and 10 (x) were chosen.
Overall, each operation is executed 50 million times, and the average execution time is calculated using the time it took to execute these.

In the figure below, the results of the benchmarks for comparing different JavaScript expression kind execution times when using the pipeline versus not using it without the Graal compiler are visible. Instantly, it becomes clear that using the pipeline results in a slight increase of the execution time in all the cases below. The biggest difference in execution time occurs when using the pipeline with yield statements.

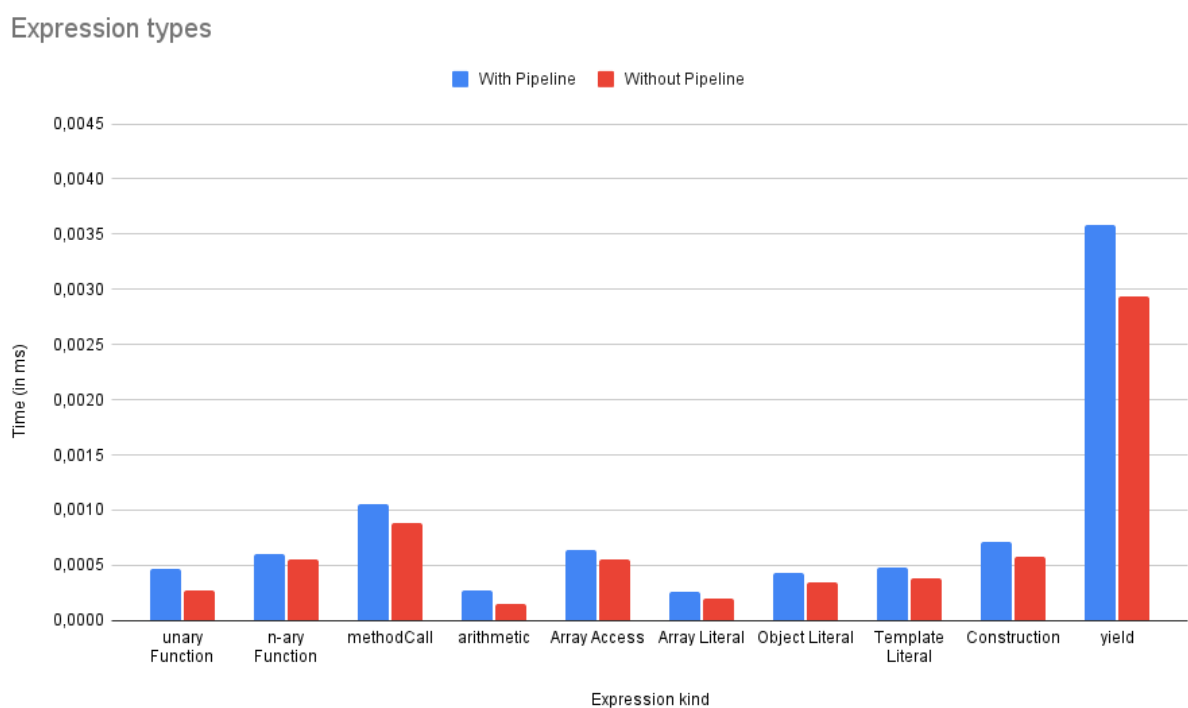


Figure 5-1: Benchmarks comparing execution times of single expressions

In the case of chaining pipelines, the result is similar, but with one exception. When chaining three pipelines and using the special case expression % % %, the pipeline is faster in average execution compared to no pipeline. Moreover, in the graph below, it is visible that the execution time increases for each additional pipeline in the chain, which is the expected behaviour. Moreover, the difference in execution time gets larger for each additional pipeline operator. Also in this case, the pipeline use increases the execution time compared to without a pipeline.

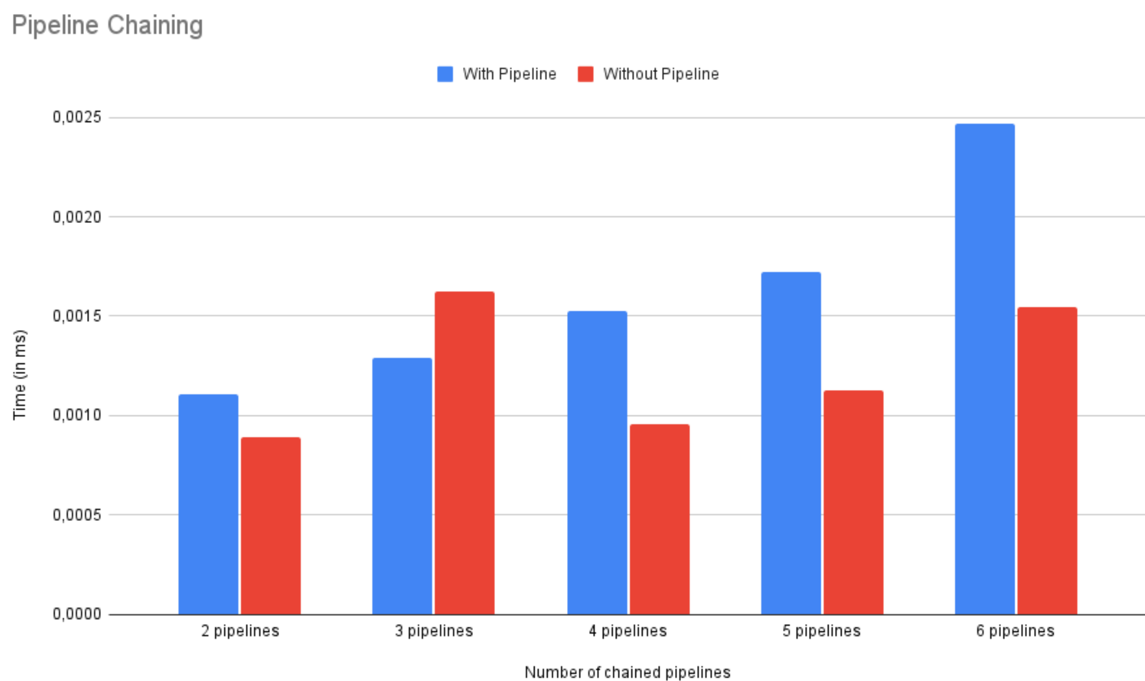


Figure 5-2: Benchmarks comparing execution times of chained expressions

For the second part, the execution times using the Graal compiler were compared. In Figure 5-3 below, the results of the benchmarks for single expressions are visualized. The average execution times are much lower compared to the interpreter results. For most of the expression types compared, the execution times are nearly identical, with the pipeline even outperforming the no-pipeline expression for the object literal and construction benchmark. On the other hand, the pipeline is significantly slower when used in combination with a method call, where the execution time is about three times higher than without a pipeline. Moreover, the highest impact on execution times is noticeable for the array operations and template literals with roughly 25% of additional execution time (on average).

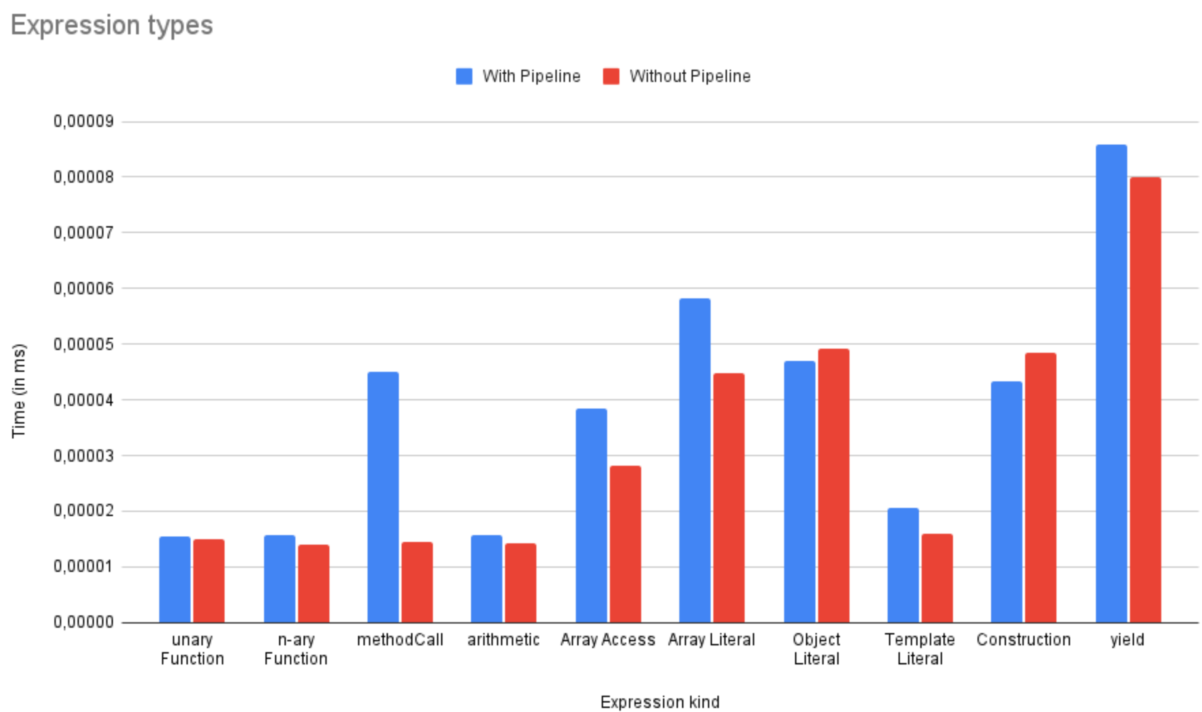


Figure 5-3: Benchmarks comparing execution time of single expressions using the Graal compiler

The chaining benchmarks provided rather different results, as seen in Figure 5-4. The increase in execution time with pipelines is very steep at about 50 up to 75 percent of the execution time. This result is surprising, as well as the fact that the number of pipelines does not really affect the execution time. From two to six chained pipelines, the average execution time of the benchmark remained rather constant and even decreased slightly when adding an additional one, e.g., comparing two and three chained pipelines.

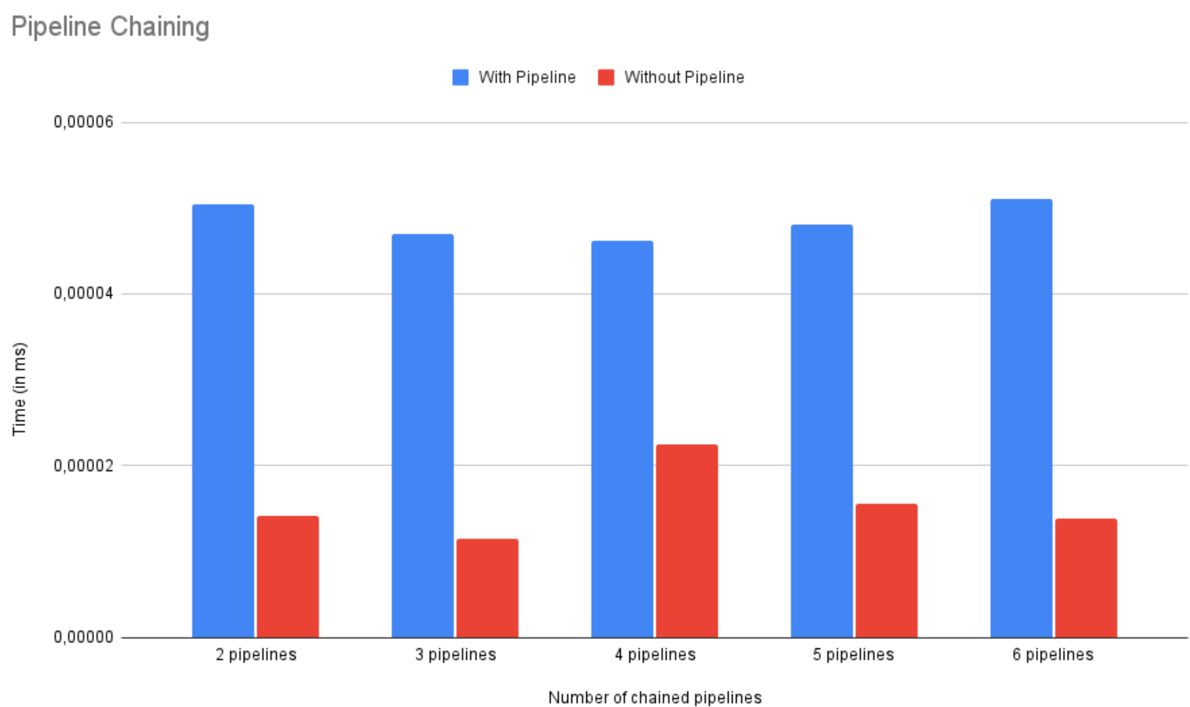


Figure 5-4: Benchmarks comparing execution times of chained expressions with the Graal compiler

All in all, these benchmarking results show, that my implementation of the pipeline operator leads to a slight increase in execution times. As expected, the execution times are many times faster with the Graal compiler. For single expressions, there is only a marginal rise in execution time as the result of using the pipeline operator for most types compared to equivalent expressions without pipelines – regardless of the usage of the Graal compiler. The most significant outlier is the methodCall benchmark with compiler, where the increase is staggering at roughly 70 %.

Chaining cases showed a bit of a different picture. In almost all cases, the benchmarked execution time without the use of the pipeline operator is much faster. Here, you must differentiate between the two sets of benchmarks (with/without compiler).
Without the compiler, the execution time increases steadily for each additional pipeline operator added, and even outperforms the non-pipeline equivalent on one occasion and the increase is at a maximum of 40% for these benchmarks.
With the compiler, the execution time remains almost constant when adding another pipeline to the chain. However, the percentual difference is higher at about 70 to 75% for most of the cases.

It was expected that the execution time would increase when using the pipeline operator, as additional checks and the assignments have to be executed. As the main goal of the pipeline is to increase readability of chained expressions, the decision to use the pipeline probably results in a trade-off: If the goal is to write readable and easily understandable code, the pipeline is the best option. When writing performance-critical code, the pipeline in its current form is most probably not the number one choice, as the equivalent code is faster in most cases.

Lastly, the new Graal.js pipeline implementation was compared to the only other publicly existing implementation, which is in Babel.js.
The results of this comparison can be seen below in Picture 18. The blue bars are representing the average execution times of the pipeline in Graal.js, the red ones in Babel.js. It is visible, that the execution times are very similar between the two, with slightly better performance in Graal.js. For two and six chained pipelines, Babel.js is faster by around 2 %. The biggest performance difference appears to be in the 5-pipeline chain, where the Graal implementation outperforms Babel.js by 20%.

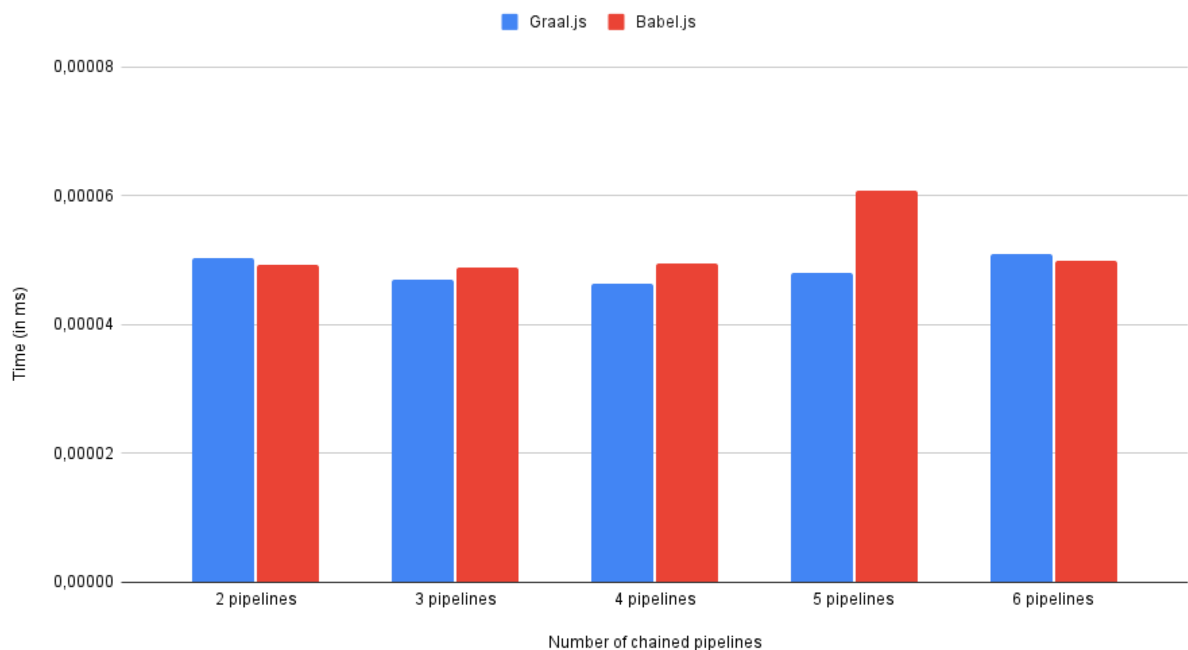


Figure 5-5: Benchmarks comparing execution times of chained expressions with the Graal compiler and Babel.js.

5.2. Interoperability

Another task of this thesis is to explore the interoperability of the implemented pipeline operator with other Truffle-based languages. Language interoperability is one of the core features of GraalVM. It allows the user to take advantage of elements from different programming languages and combine them. In this part, mainly the interoperability with Java was explored and some possibilities of combining the implementation of the pipeline operator with Java code were tried out.

Java interoperability is provided directly when using Graal.js. Classes, construct objects, call methods and many other Java language concepts can be constructed in JavaScript code. Special functions are provided to access Java types. [11]

The interoperability was explored by taking some use cases from Java, where a pipeline operator would be useful and test their compatibility with the JavaScript pipeline using the Truffle feature.

The first example is about simple function chaining and uses String functions, as chaining occurs frequently in this context. First, a Java string with the value **“Java”** was created by using the Java.type function. Next, the pipeline was used and called the method **concat** to concatenate the String with another one and then the **toUpperCase** and **trim** methods were called.

```
var javaString = new (Java.type('java.lang.String'))("Java") |>
%.concat(" used in JavaScript with a pipeline    ") |> %.toUpperCase()
|> %.trim();
```

This example was executed, and the result was the expected String **JAVA USED IN JAVASCRIPT WITH A PIPELINE.** This shows that the new pipeline operator for JavaScript is interoperable with Java, and you can use it for any method chains to make them more readable.

Another benefit of the interoperability between JavaScript and Java in Graal.js is the ability to use Java standard libraries in JavaScript code like the Math library. This library provides helpful functions for calculations like, Math.min()/max(), Math.floor()/ceil(), Math.sqrt(), Math.round() and also constants like Math.Pi. Again, a simple example using several of the methods above was created and they were combined with a pipeline operator as visible in the code snippet below.

```
var result = Math.pow(2,16) |> % + Math.floor(Math.PI)*Math.max(%, 36*146);
```

This leads to the conclusion that the JavaScript pipeline implementation of this thesis for Graal.js is fully interoperable with Java, as long as the main requirement – topic reference is used in the right-side expression at least once – is fulfilled. The same is also valid for the other Truffle languages, which allow interoperability with JavaScript. Users should be able to use the pipeline combined with variables, arrays, objects, and other constructs without any issues, provided the syntax is valid according to the descriptions in the previous chapters.

6. Related Work

As the Pipeline operator is currently only at Stage 2 out of 4 of the ECMAScript proposal process, the feature is not (publicly) integrated in most other JavaScript engines. One engine that supports the pipeline already is Babel.js, where you can experiment with the pipeline syntax by installing a corresponding plugin. Babel.js is a JavaScript compiler, which performs syntax transformations to convert new JavaScript syntax to old, browser compatible code. This allows users to use any novel language features like the Pipeline and transforms it to equivalent code of the current JavaScript language. [12]
The implementation of the pipeline proposal plugin for Babel is written in TypeScript and the approach taken is the same as in this thesis. Pipeline expressions are resolved into chains of assignments with generated variables, which ensure the correct bindings. This emerges from the Babel.js Github repository.[3]

The RxJS framework also provides a pipeline functionality. RxJS is a JavaScript library for reactive functional programming. One feature of this framework is the pipe() function, which is a pure function. It can be called on an Observable and creates another one as output. The parameters of the function are the operators that should be performed sequentially on the Observable. Examples for operators are map(), filter() and many more functions in this context. [13] This approach is implemented in TypeScript using the functional paradigm passing parameters from one function to the next to create the pipe.[4]

The Pipeline operator is also used in other programming languages. Microsoft provides a pipe operator in F#. It has the same token as pipeline operator (|>) and is used to pipe results between functions. Like this, functional pipelines can be used. F# also allows to pipe two or three different arguments with the operators ||> for two parameters and |||> for three. To enable that, the parameters are combined to a tuple and then piped to the next function. Furthermore, it is possible to perform backward piping by simply reversing the operators in the following way: <|, <|| and <|||. This way, piping from left to right can also be used, which is called backward pipeline by the developers. [7]

[3]https://github.com/babel/babel/tree/b5d6c3c820af3c049b476df6e885fef33fa953f1/packages/babel-plugin-proposal-pipeline-operator/src (Last visited: 4th October 2023)
[4]https://github.com/ReactiveX/rxjs/blob/0bd47eab10dec89f245b888f1f26e03cb36d2a78/packages/rxjs/src/internal/util/pipe.ts#L4 (Last visited: 8th October 2023)

The F# version of the pipeline was also proposed as a baseline for the JavaScript proposal; however, this approach was rejected by the committee. The implementation resolves the pipeline operators using inline functions. With that, the parameters and functions are desugared in this approach.[5]

Hack is an object-oriented programming language developed by Meta especially used for developing websites, which also introduces a variant of a pipeline operator. [8] The Hack pipeline operator uses the placeholder token - $$ in this case – to store the result of the left expression and uses this placeholder in some position in the right expression. [9] The Hack languages pipeline operator is the blueprint for the JavaScript proposal. [1]
The implementation is also very similar to the approach of this thesis, as a special pipe variable is defined, which the left side expressions of the pipeline are assigned to before the right side gets executed with the variable in place.[6]

There are also other languages, where developers can use a pipeline operator. Languages that support the use of pipelines are e.g., R [14] and Elixir [15]. There is also a library in Typescript using the pipeline.[16] The most well-known usage of pipes is probably in the Linux shell. Here pipes are also used to perform sequential operations on a dataset. [17] However, all these approaches are different to the implemented pipeline in this thesis, as they do not use a topic reference token.

---

[5] https://github.com/dotnet/fsharp/blob/main/src/FSharp.Core/prim-types.fs#L4315-4315(Last visited: 8th October 2023)

[6] https://github.com/facebook/hhvm/tree/c46470d9e78fa32604115c59ed91aa55b7b101e7/hphp/hack/src/parser(Last visited: 8th October 2023)

7. Conclusion

In this thesis, the ECMAScript proposal for a new pipeline operator for JavaScript was implemented as an experimental feature to Graal.js. The implementation was based on the current draft specification provided alongside the proposal, which is currently at Stage 2 of the proposal process. Test cases were written to properly test the implementation and the performance was compared to equivalent code without pipelines using simple benchmarks.

The test cases cover most of the use cases. Tests include pipes into all the compatible expression types. Furthermore, also pipeline chains with increasing depth were tested and edge cases and errors were thoroughly implemented and tested.

Regarding performance, the pipeline operator's average execution times are a bit slower than equivalent code using chaining or nesting instead. These differences are mostly since some additional operations need to be performed in the background to create the topic binding. As the main objective of the pipeline is increase the readability of chained expressions, these small differences are acceptable. Compared to the only other existing solution by Babel.js, the pipeline operator implemented in this thesis performs slightly better.

Regarding the interoperability with other Truffle languages, the pipeline implementation is fully interoperable with Java. The pipeline can be used combined with any Java code, provided the constraints for its use are complied with. This is also valid for all other Truffle languages.

The implementation is already in the process of being integrated into Graal.js, a pull request[7] has been created in the Github repository and is currently under review. The implementation is complete under the current specification, although changes may be necessary once the proposal progresses into the next stages of the process, as some details of the syntax may be adapted.

[7] https://github.com/oracle/graaljs/pull/761 (Last visited: 28th September 2023)

Acknowledgements

I would like to thank my supervisors, Lukas Stadler and Christian Wirth, for their continuous support in all aspects over the course of this thesis. Furthermore, I would like to thank Andreas Wöß for his technical feedback, advice, and suggestions. I would also like to thank my primary supervisor, Prof. Mössenböck for his feedback on my work.
Finally, I want to especially thank my family and friends for their unconditional support.

List of Figures

Figure 2-1: GraalVM architecture overview 11
Figure 2-2: Node rewriting and partial evaluation using Truffle 13
Figure 2-3: Deoptimization, node rewriting and recompilation using Truffle 13
Figure 3-1: Simplified Architecture of Graal.js 15
Figure 4-1: AST of a pipeline expression with execution order 21
Figure 4-2: AST of a nested pipeline expression 22
Figure 4-3: Parser Tree of the expression 1 + 2 |> % + 4 22
Figure 4-4: Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - % 23
Figure 4-5: Generic AST of a chained pipeline for Approach 2 25
Figure 4-6: Revised Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - %.. 26
Figure 4-7: AST for the third approach with CommaRightNode 28
Figure 4-8: Parser Tree of of the expression 1 + 2 |> % + 4 with third approach 29
Figure 4-9: Parser Tree of the expression ints[1] |> multiply(%, 3) |> 100 - % for third approach 30
Figure 5-1: Benchmarks comparing execution times of single expressions 35
Figure 5-2: Benchmarks comparing execution times of chained expressions 36
Figure 5-3: Benchmarks comparing execution time of single expressions using the Graal compiler 37
Figure 5-4: Benchmarks comparing execution times of chained expressions with the Graal compiler 38
Figure 5-5: Benchmarks comparing execution times of chained expressions with the Graal compiler and Babel.js. 40

8. References

[1] Pipe Operator (|>) for JavaScript proposal. URL: https://github.com/tc39/proposal-pipeline-operator (Last visited: 21st June 2023)

[2] ECMAScript proposal stages. URL: https://www.proposals.es/stages (Last visited: 21st June 2023)

[3] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013). Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[4] Truffle Tutorial. URL: https://www.graalvm.org/latest/community/publications/ (Last visited: 21st June 2023)

[5] GraalVM JavaScript Implementation. URL: https://www.graalvm.org/latest/reference-manual/js/ (Last visited: 21st June 2023)

[6] What do you feel is currently missing from JavaScript? https://2020.stateofjs.com/en-US/opinions/#missing_from_js (Last visited: June 21st 2023)

[7] F# Symbol and Operator reference. https://learn.microsoft.com/de-de/dotnet/fsharp/language-reference/symbol-and-operator-reference/ (Last visited: 11th September 2023)

[8] Hack programming language. https://hacklang.org/ (Last visited: 11th September 2023)

[9] Hack documentation, Expressions and Operators: Pipe. https://docs.hhvm.com/hack/expressions-and-operators/pipe (Last visited: 11th September 2023)

[10] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In Proceedings of the 8th symposium on Dynamic languages (DLS '12). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

JOHANNES KEPLER UNIVERSITY LINZ

[11] Graal.js Java Interoperability. https://www.graalvm.org/latest/reference-manual/js/JavaInteroperability/ (Last visited: 28th September 2023)

[12] What is Babel? https://babeljs.io/docs/ (Last visited: 28th September 2023)

[13] RxJS operators. https://rxjs.dev/guide/operators (Last visited: 28th September 2023)

[14] Der %>% Operator: Pipes in R. https://statistikguru.de/r/pipes-in-r.html (Last visited: 28th September 2023)

[15] Elixir Pipe Operator. https://elixirschool.com/de/lessons/basics/pipe_operator (Last visited: 28th September 2023)

[16] Functional Programming in TypeScript using the fp-ts library: Pipe and Flow Operator. https://www.thisdot.co/blog/functional-programming-in-typescript-using-the-fp-ts-library-pipe-and-flow/ (Last visited: 28th September 2023)

[17] Linux-Pipes erklärt. https://www.ionos.de/digitalguide/server/konfiguration/linux-pipes/ (Last visited: 28th September 2023)

[18] Introduction to JavaScript. https://www.geeksforgeeks.org/introduction-to-javascript/ (Last visited: 10th October 2023)

[19] ECMAScript 2024 Language Specification. https://tc39.es/ecma262/ (Last visited: 10th October 2023)

[20] Variables and Datatypes in JavaScript. https://www.geeksforgeeks.org/variables-datatypes-javascript/ (Last visited: 10th October 2023)

[21] Introduction to Object Oriented Programming in Javascript. https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/ (Last visited: 10th October 2023)

9. Picture References

[1] Introduction to GraalVM. https://www.graalvm.org/22.3/docs/introduction/ (Last visited: 21st June 2023)

[2] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013). Association for Computing Machinery, New York, NY, USA, 189. https://doi.org/10.1145/2509578.2509581

SWORN DECLARATION

I hereby declare under oath that the submitted Master's 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.

Traun, October 23, 2023

Alexander Stummer