FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	A translator of DET scripting language into Java
Student:	Daniil Grankin
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2020/21

Instructions

Describe the syntax of the DET scripting language and its existing parser and translator into the Java language. Both the scripting language and its translator are proprietary; the translator has an ad-hoc design with no grammar. Design a grammar of the DET language and implement a new grammar-driven translator from DET scripts into Java. It will produce Java source code optimised for both speed and the minimal number of temporary objects. Upon agreement with the supervisor, choose features of the DET language to be implemented in the parser. The parser is considered to be a prototype. Perform tests and compare the new parser with the old one; the testing and comparison will show both validity and speed of the new parser.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 1, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Translator of DET scripting language into Java

Daniil Grankin

Department of Computer Science
Supervisor: Ing. Ondrej Guth, Ph.D.

May 14, 2019

Acknowledgements

I would like to thank my parents for giving me all the opportunities, my supervisor and colleagues for guiding me all the way and my friends and life partner for supporting me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 14, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Grankin, Daniil. *Translator of DET scripting language into Java.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Cílem této práce je návrh a implementace vylepšeného skriptovacího enginu, řešení problémů a návrh nových vlastností existující proprietární implementace tohoto enginu. Skriptovací engin se používá ke zpracování skriptů napsaných v jazyce založeném na Javě. Hlavní nedokonalost existujícího skriptovacího enginu DET je použití výhradně regulárních výrazů na zpracování skriptu. Toto je řešeno v prototypu, který je předmětem této práce. Důraz je kladen na běžně používanou techniku konstrukce překladačů: skriptovací jazyk je popsán gramatikou, dále se pracuje s abstraktním syntaktickým stromem.

Klíčová slova ANTLR, grammar, AST, Sripting language.

Abstract

The goal of this thesis is to design and develop an improved scripting engine solution, addressing the problems and revisiting the features of the existing proprietary Scripting Engine implementation. The Scripting Engine is used to process the scripts, written in Java-like syntax with custom language extension into the methods of java classes, which could be compiled, and referenced from the platform. The main imperfection of the DET Scripting Engine is that it is relying on the regular expressions as on the script alternation tool. This

flaw is approached in the prototype project, which is the subject of this thesis. The emphasis is given to the common compiler-construction technique. The scripting language is described with well-defined grammar. The parsing of the scripts provides an abstract syntax tree which could be then traversed with the predefined algorithms.

Keywords ANTLR, grammar, AST, Sripting language.

Contents

List of Figures

List of Tables

Introduction

In the modern world, international real-time exchange of information related to the securities transactions and markets is made possible by using the various messaging protocols. One of such protocols is FIX, which stands for Financial Information eXchange and provides the format of the messages, which are used to exchange the data. The message is used as a data container for various message fields like price, quantity, and others. Each message fields has a specific number, called tag number, which serves as an index for a particular field.

Low latency trading is based on algorithms reacting to various market events and aimed at doing operations faster than competitors to increase the profitability of trades. The crucial part for low latency trading product is to maintain and process the constant flow of the data as any latency above the threshold can result in the loss of profitability.

The Dynamic Electronic Trading platform provides messaging and routing management tools which allow designing complex business routing solutions for low latency electronic trading.[2] The main parts of the Platform do not create objects on the heap since the garbage collector has adverse effects causing jitter.

One of the most significant parts of the platform is the scripting engine. The scripting engine is an environment for developing code components. Code components are mainly used to define the routing logic in terms of matching, transformation and other actions over the messages. The scripts are written by the user and transformed to the java code for further execution in the routing core.

The engine allows the definition of custom functions that can be used in the scripts. Users can use the extensive list of available built in functions, explicitly designed for electronic trading purposes. The functions are specifically annotated static java methods on the backend. Those functions can accept variable number of parameters and have a return type. Apart from regular custom functions, there are so-called runtime functions. Runtime function has

the following features:

- It can be written by the user in the same way as any code component
- The same set of features as in a regular custom function is presented, that would be an abstract number of parameters, return type, and platform-wide accessibility
- It can be utilized in the scripts as well as in the code using the request to the runtime functions provider

All the scripts written by the user are processed during the runtime. Therefore, it is not required to restart the platform on the script addition or modification.

The workflow of this project can be represented as a waterfall of the subprojects, divided into the chapters:

- Current State: the analysis of the existing version of the engine, describing the functionality and the problems of the implementation
- Analysis and Design: addressing the problems of the old engine and designing the solution
- Realisation: description of the developed prototype implementation
- Testing and Validation: comparing the implemented prototype capabilities with the old version of the engine using the benchmarking and the output validation.

Chapter 1

The goal of the work

The main focus of the first version of DET scripting engine is the flexibility of the script language and performance of outcome java code. The scripting language special non-java expressions are transformed to java expression by the engine, and the resulting scripts meet the requirement of minimal temporary objects and overall performance, which is achieved by using various optimizations.

The engine does not have formally defined syntax but instead uses regular expressions to replace matched parts of the code. The algorithms which are used to optimize the code are not efficient in terms of complexity. Many features were written ad-hoc as workaround resulting in non-optimal design and maintainability. The used libraries do not allow proper redesigning of the engine. The user-friendliness suffers from lack of adequate error feedback and tremendous verification and application time due to the non-organized system of handling the cache and high complexity of the code transformations.

In this work, all the downsides of the script engine are addressed to deliver more pleasant script-development. The performance requirements of the resulting scripts remain the same. The work is mainly focused on redesigning the engine to have enough flexibility for elegant extension of the feature set and yet be capable of the previous functionality. The goal is to improve user experiences, such as error feedback, intuitive semantics, fewer restrictions in custom expressions and the building time. Nevertheless, the development experience should be enhanced as well. By providing the more advanced design, integration of new features would be effortless. The achieved enhancements would lead to quick delivering process and convenient overall usage of the platform.

Chapter 2

Theoretical Background

2.1 FIX Protocol

The Financial Information eXchange protocol is an ASCII message-based protocol, created in 1992 as a communication framework which was used between the counterparties [6]. The FIX 4.4 possesses 956 tags which are used to build the messages. The tags are represented by the integers. [3] The message type could be identified by tag number 35, which will contain one of 92 types of messages. [4]

2.2 FIX Engine

The FIX engine is an implementation of the FIX protocol, which manages the messages exchange between counterparties by establishing the connection and handling the requests. [9] There are many publicly available implementations, while the electronic trading companies mainly use their proprietary version.

2.2.1 QuickFIX

QuickFIX is an open source FIX engine based on Apache license. While the core implementation is written in C++, the API is available in many languages, such as Java, C++, .Net and others. [10]

2.3 Parser

A parser is a component that processes the input into the smaller elements, which are easier to translate to another language. The processing steps include the following stages:

- Lexical Analysis: splitting the input string into the tokens.

- Syntactic Analysis: validation of the tokens to form the defined expressions referring to a context-free grammar. An abstract syntax tree is generated during this stage
- Semantic Parsing: the validation of the implications of the expression and taking the appropriate actions.

Parsers are widely used in programming languages and exchange protocols. [12]

2.4 ANTLR

ANother Tool for Language Recognition is a parser generation tool written in Java and published by the Terence Parr under the BSD License. It is widely used in many parsing tasks and generally supports any language definable by the LL(*) grammar. The appliance of the ANTLR could be found in the Twitter search engine, used to parse the queries. [1]

Chapter 3

Current State

The engine can be described as a set of seemingly independent components, which exchange the data to achieve the common goal – build the script and apply it to the system.

3.1 Input

The dedicated script editor window is used to capture the user‘s input. After the user finished development of the script, there is an option to build configuration, which passes the code to the backend system for further examination and appliance on successful validation.

3.2 Replacement

The received code is passed to the replacement component to transform all the custom expressions into compilable java code. Code replacement component performs the search using regular expressions which matches the patterns derived from protocol dictionaries, custom functions, and custom variables.

The regular expression replacements are mainly based on several common replacements. The following list of the replacements includes the description of the functionality as well as the example of usage of the particular replacement. The structure of the list entity would be the following:

- Name
- Description
- Input pattern
- Output pattern
- Example of the input expression

- Resulting output expression

All the examples are based on the FIX protocol version 4.4. The list of the replacements follows:

- **Field name to tag replacement**
  The field does not represent an enum but serves as the alias for the index in the array of the fields.

  {message_variable_name}.{field_name}
  {message_variable_name}[{field_tag}]

  ```
  INFO($M.MsgType);

  INFO($M[35]);
  ```

- **Field tag to method call replacement**
  As the message variable is not an array, but rather an array wrapper, the fields are accessed with the method, receiving the index of the field.

  {message_variable_name}[{field_tag}]
  {message_variable_name}.getValue({field_tag})

  ```
  INFO($M[35]);

  INFO($M.getValue(35));
  ```

- **Special variables replacement**
  The short named alias for the variable should be translated to the actual variable name or to the method which provides the variable.

  ${special_variable_name}
  {variable_name} OR {method_name}

  ```
  String x = $M.getValue(35);
  Message y = $A;

  String x = m.getValue(35);
  Message y = ArrayFunctions.getCurrentMessage();
  ```

- **Constants replacement**
  The constants are enums, but given that they are not provided as the enum class by the fix engine, the enum to the value transformation should be performed by the engine.

  {field_name}.{enum_of_the_field_value}
  {field_value}

```
if(m.getValue(35) == MsgType.NewOrderSingle){
    INFO(m.getValue(1741) == UpfrontPriceType.Percentage);
}

if(m.getValue(35) == "D"){
    INFO(m.getValue(1741) == 1);
}
```

- **Custom method replacement**
  The custom method aliases should be replaced with the actual methods.
  {custom_method_alias}
  {actual_method}

```
if(m.getValue(35) == MsgType.NewOrderSingle){
    INFO(m.getValue(1741) == UpfrontPriceType.Percentage);
}

if(m.getValue(35) == "D"){
    INFO(m.getValue(1741) == 1);
}
```

After all the regex replacements were done, the code does not contain any custom expressions and can be described as pure java code. However, it is not ready to be compiled yet, as the Java code has to be written in the method of the class. There are predefined templates with different arguments, return values and method names to fulfill code components requirements. The template for runtime functions does not contain the method to provide flexibility to define the parameters and return type in the runtime. The code is inserted into the template in order to obtain defined Java class with a known method name. At this step code is ready to be compiled.

3.3 Byte code cache

The compilation of the scripts is a long process in terms of the real-time functioning system. Byte code cache of every script is stored to minimize the time needed to execute the scripts. The component responsible for byte code cache access possesses a map, where the key is the java non-optimized code, and the value is the byte code, which was created by the compilation of the optimized java code. The string containing the java class is used to determine the existence of an already compiled java code. In case the code was already compiled, byte code can be loaded as the class and used immediately. In another situation, if the code was not compiled yet, it is passed to the optimization component, and upon successful optimization, it gets to the compilation component.

When the non-runtime function script is changed, the byte code cache component invalidates the cache of that script and treats it as a newly created script.

The runtime functions can be used in other code components. The byte code cache does not keep track of the usages and invalidate all the byte code cache of the code-components in case the runtime function is changed. That leads to the necessary rebuilding of all the scripts upon any change in runtime function.

3.4 Optimizations

The optimizations component does replacements on certain kinds of expressions. The strategy is to find the code which is to be replaced, replace it and repeat the process until there is no code to replace. The way those expressions are detected is parsing the code, building an abstract syntax tree and visiting of the tree, which will perform optimization search on specific nodes of the tree. There is a possibility to provide modifiers for expression statements; all other optimizations are predefined in visitor itself.

The predefined optimizations are described in the following sections.

3.4.1 Auto-cast

Given a variable is initialized with the initial value of the result of the getValue method, then the type of a variable is determined, and the cast is added concerning the variable type. It has to be mentioned, that the auto-cast optimization is not capable of determining the type of a variable, declared before the assignment.

The following example expression

```
public boolean match(Message m){
    int a = m.getValue(53);
    return a == 3;
}
```

would be transformed to

```
public boolean match(Message m){
    int a = (int) m.getValue(53);
    return a == 3;
}
```

3.4.2 Custom Value Getters and Setters

If getValue method's result is casted to some type, the optimization tool will replace the cast with the special method which will retrieve the value with the

desired type. The replacement of the simple cast with special method is done due to the technical specifications of the internal Message implementation.

The following example expression

```
public boolean match(Message m){
    int a = (int) m.getValue(53);
    return a == 3;
}
```

is transformed to

```
public boolean match(Message m){
    int a = HighPerf.getInt(m, 53);
    return a == 3;
}
```

3.4.3 Custom Comparison Methods

If m.getValue is a part of a comparison expression, the whole comparison expression is replaced with a custom comparison function, which receives the message object and corresponding field number as the left operand. The optimization works when m.getValue is right operand as well in the same manner. Custom comparison function determines the type of value stored under the given index. Appropriate comparison methods are selected based on the type of values.

Chapter 4

Analysis and Design

The previous approach to process the scripts using a set of regular expressions fails in flexibility and performance. The better approach would be to use a common compiler-construction technique. The parsing was already done in the previous implementation. However, the parsing, in that case, would only take place, when the script is inserted to the template of the class, as have been discussed in Section 3.4. The parsing and validation of the template is not a necessary thing to do since it would consume the time needed to parse the class structure every time the processing of the script takes place. Therefore the improvement would be to create specific script grammar and the parser based on that grammar.

The feature set should be capable of backward compatibility with the first generation scripts. Therefore the features, which were supported in the previous version, should be supported in the new one. Moreover, some of the features should be improved, and others need to be redesigned due to the changed structure of the engine. The new features should be added as well to satisfy the needs of the script developer.

The engine is fed with the data, such as the fields aliases, the constants, the custom methods, and variables. The source of that data is an environment of the engine. The prototype version of the environment would be implemented to feed the prototype engine with the data.

The prototype version of the engine would not be capable of compiling the scripts and managing the saved bytecode. The project aims to transform the scripts into the Java code. This project would be late utilized in the platform to process and transform the scripts.

The more detailed analysis of these aspects would be described in the following sections.

4.1 Grammar

The regular expressions are too difficult to maintain and lack the needed flexibility for extending the engine‘s feature set. The specifically designed grammar would serve a better role in that aspect as it would be well modifiable and much more flexible and faster than the regular expressions. ANTLR provides the tools for grammar development, as well as auto-generation of the parser and a helpful API for AST processing.

The Java 8 grammar for ANTLR could be found on the ANTLR GitHub repository. [8] It describes the syntax of the java quite well, meeting requirements of the DET scripting language syntax. However, grammar still contains the rules that are not needed in the engine and should be eliminated to minimize the grammar and hence the complexity of the processing.

The grammar would support custom DET expressions, such as the custom message variable, the custom field access, and other expressions could be supported and added on demand.

4.2 Abstract Syntax Tree Processing

The speed of the process of script transformation is not obliged to improve, as it is not a problem of the existing scripting engine. The main enhancement of the new scripting engine implementation is the improved flexibility of the tool at the same time keeping it simple enough to decrease the threshold for developers to maintain the engine.

The ANTLR parses the scripts and outputs the abstract syntax tree which is to be processed by the engine. The processing can be divided into two parts.

4.2.1 Type Assignment

The type of the necessary nodes, such as the expression node and left-hand side node, should be discovered and saved for later use in the next steps of the processing.

The types should cover all the following aspects:

- Basic primitive types
- Arbitrary reference type and special reference types (such as String)
- Custom types to represent the message field access and the constant enum
- Method arguments type to handle correctly and cast or transform the arguments into the required type
- The type to combine the types of the subexpressions.

The types would be required in the next processing step, where the types of nodes would result in different processing ways. The following example would demonstrate how the processing of the expression should change based on the types of a subexpression.

4.2.2 String Extraction of Expressions

The string extraction would be then performed on the abstract syntax tree using the type-information to perform decisions on the extraction of the expressions.

The types of expressions are necessary to determine the kind of the main expression, where the main expression is the set of the subexpressions. Particularly, for example, the message field assignment should be treated differently from the primitive variable assignment, as in case of message field access, the access itself should be performed with the specific method dedicated to the insertion of the data in the message fields, while the primitive variable case would not require any modifications. The similar logic applies to the variable initialization and relation expressions. Apart from modifying the scripts, the purpose of the string extraction would also lay in dismissing the unnecessary symbols and separating the symbols with the predefined separators, as the grammar would be designed to skip the whitespaces, tabulators and the newlines.

4.2.3 JAVA Blocks

Old scripting engine offers an ability to process and execute pure java code as if it was in public static method of an empty public class. For the sake of minimalization, not all of the expressions would be supported in the prototype. Complex statements, such as generics and lambdas, are omitted, but the ability to use those constraints would remain with the introduction of the dedicated JAVA block.

Code under the JAVA block would not be modified nor optimized during the processing of the script and would be directly inserted into the resulting Java code as a block.

The JAVA block would not be intended to be as a commonly used option. If the developer would lack the functionality of the scripting language, it would still be possible to implement the logic as part of the plugin creation. The plugins are the modules, which could be injected as part of some bigger module of the platform. The requirements for the plugin implementation would be provided to the clients on demand alongside with the instructions on how to make a custom function as part of the plugin, which would be injected into the set of all custom functions and would be available to the scripting engine.

Java Language allows the usage of blocks inside the method. That gives an opportunity to treat JAVA block as a simple block as the variables declared

inside the block can only be accessed in that same block or its subblocks. The variables declared before the JAVA block could be referenced inside the block.

4.3 Features

4.3.1 Code Optimizations

The code optimizations remain one of the most significant aspects of the scripting engine. The optimizations should be performed in a known amount of phases with the intention of decreasing processing time.

While all the functionality should remain the same, some of the features need to be redesigned, as they heavily count on specific patterns in the strings.

The type related optimizations should possess the types of every declared variable with the purpose of performing the type optimizations in every other possible place and not only in the declaration expression, where the type is easily accessible.

The control of the types gives an opportunity to declare IMessage variables, which would be treated in the same way as the special variables.

4.3.2 Error Position Detection System

There can be a various number of optimizations and modifications on the code passing it to the engine. Java compiler feedback is not informative enough for the user to locate the position of the error in original source code. Java compiler diagnosis includes the position of the diagnostic and the message itself. All the optimizations provide the information of the code replacement, mainly the position and length of both input and output code. That is enough for the engine to trace the error back to the original source code and provide an informative error message to the user.

4.4 Environment

Building the Scripting Engine 2.0 under the old environment would affect the development process with the inability to launch the scripting engine separately without starting the whole platform. Therefore the new trivial environment should be designed. The environment should provide the following data:

- The FIX Data is needed to validate the engine against the existing script, which contains FIX constant expressions and FIX field names in message field access.
- As the tests would expect the engine to produce the result as close to original engine‘s result as possible, the custom functions and variables

metadata should duplicate the existing ones in terms of the package, class and method name.

Chapter 5

Realisation

5.1 Enviroment

As was presented earlier, necessary environmental work is required to emulate the space of the platform and feed the engine with the required data.

5.1.1 FIX Data

The required data for the scripting engine is the message fields, enumerability of the fields and the enums themselves.

The FIX engine would take the role of the FIX data provider. FIX engine is a framework, which is generally the implementation of the FIX protocol, which is used to handle the FIX messages. [9]

The current version of the scripting engine uses a proprietary version of the FIX engine, which should not be used with the prototype. Therefore an external FIX engine should be used.

QuickFIX is a free and open source implementation of the FIX protocol. [10] It provides the FIX data dictionary version 4.4, which would be sufficient for the prototype. Alongside with the dictionaries, it provides the Java library with the API to handle FIX messages. [11] The API for the scripting engine required features has the private modifier, but as the QuickFIX permits to modify the source code, all the code can be copied to the editable class and the methods providing public message fields and other field related metadata could be marked public. That would take place in the prototype, while the further integration of the scripting engine prototype into the platform would require the usage of proprietary FIX engine.

5.1.2 Custom Functions and Variables Metadata

The Java ReïňĆections API provides the necessary functionality to search for the annotated classes and methods and derive the metadata such as annotation

ïňĄelds and method names, which would be needed to provide the names of custom functions to the engine and ability to convert them to the Java method invocation. The implementation of the custom functions and variables metadata provider would use the reflections to scan the classpath for classes, annotated with the custom annotation, which would indicate the presence of custom methods and variables inside the class. The class would be then scanned for the methods annotated with the annotations designed to represent the custom function or variable. The methods would be saved as well as the arguments of the annotations, as they contain the name of the alias for the custom function or the variable.

5.2 Grammar

The DET scripts are made executable by creating a class with the static method containing the script. That would mean the grammar should not include any rules other than those used in the Java method blocks. The entry point of the grammar is represented with the following ANTLR rule, which could be seen on Figure 5.1

```
1 scriptRoot
      :   (blockStatements)* returnStatement? endOfFile
3     ;
```

Figure 5.1: ANTLR grammar rule representing the DET script syntax

The rule states that there could be the arbitrary number of the block statements, as well as no block statements at all. The block statement part should be followed by the optional return statement and required end of the file.

As the scripts are not used to implement complex algorithms requiring non-trivial expressions, many features of Java 8 can be omitted in favor of having a more minimalistic and maintainable framework. The excluded expressions are the following:

- Lambdas
- Generics
- Arrays
- Try-catch blocks and exceptions (try-finally is still relevant)
- Method references
- Variable modifiers

- Assert statement
- Synchronized blocks
- Bit operators (shift, bit-or, bit-and, bit-xor, etc.)

The additional set of rules is added to handle the message field access expression. The message field access can be represented as a cartesian product of the message expression representation set and field-access expression representation set.

5.2.1 Message Expression Syntax

The message can be represented as one of the following:

- $SPECIAL_VARIABLE
- $SPECIAL_VARIABLE(SPECIAL_VARIABLE_ARGUMENT)
- SPECIAL_METHOD()
- VARIABLE

```
customVariableName
    :   customVariablePrefix Identifier ('(' integer ')')?
    ;
directMethodInvocation
    :   methodName '(' argumentList? ')'
    ;
messageVariable
    :   Identifier
    ;
```

Figure 5.2: ANTLR grammar rules representing the Message Expression syntax

ANTLR rules for these representations could be seen on Figure 5.2 The message syntax representation does not assure the engine of the actual type of the expression. Any message expression should be checked against the IMessage type.

5.2.2 Field Access Expression Syntax

The field access representations:

- .FIELD
- [FIELD_TAG]

```
1 fieldAccess_after_primary
      :     '.' Identifier
3     ;

5 fieldArrayAccess_after_primary
      :    '[' integer ']'
7     ;
```

Figure 5.3: ANTLR grammar rules representing the Field Access Expression syntax

The ANTLR rules could be seen on the Figure 5.3.

The ‘fieldAccess_after_primary‘ rule does not necessarily capture the message field access. Thus it should be checked if it is the valid field of the message.

5.3 Abstract Syntax Tree Processing

As was stated in the design Section 4.2, the ANTLR‘s parsing process results in the abstract syntax tree, which is to be processed by the engine.

The processing could be done by using the visitor or listener API, which was generated alongside the parser to fulfill the needs of the tree processing. Both implementations are appropriate for the processing the abstract syntax tree. However, the visitor API is more compact and flexible. The main difference is that during visitor tree traversal, the children nodes must be explicitly visited by the visit call, whereas listener methods are called independently by the walker objects, provided by the ANTLR. That would mean that nodes which are not called during visitor processing, would not be processed, unlike during the listener processing. [7] This feature would be later used to skip unnecessary tokens.

The visitor API allows specifying the behavior for the processing of any possible node of the tree. The generated base visitor does no other work rather than passing control to the children of the current node, so any visitor which extends generated base visitor functionality would have that default behavior. The return value of the visitor’s methods could be parametrized via the template. The representation of the base visitor code could be found on Figure 5.4

The base visitor could be represented as follows:

The two primary processing steps defined in the design could be implemented by a single visitor. However, the benchmark results which can be seen on the Figure 5.5 shows no significant speed improvement over the implementation, where the processing is distributed over two visitors. The double visitor implementation exceeds the processing time by 4 microseconds, which

```
public class DETBaseVisitor<T>
    extends AbstractParseTreeVisitor<T>
    implements DETVisitor<T> {

        @Override
        public T visitNode(DETParser.NodeContext ctx) {
            return visitChildren(ctx);
        }
        ...
}
```

Figure 5.4: Generated base visitor functionality

is less than one percent of the processing time of the old scripting engine. Thus it would not make any significant difference. Comparing to the time needed to compile the script, the double visitor added processing time would be less than 0.001 % of total time.

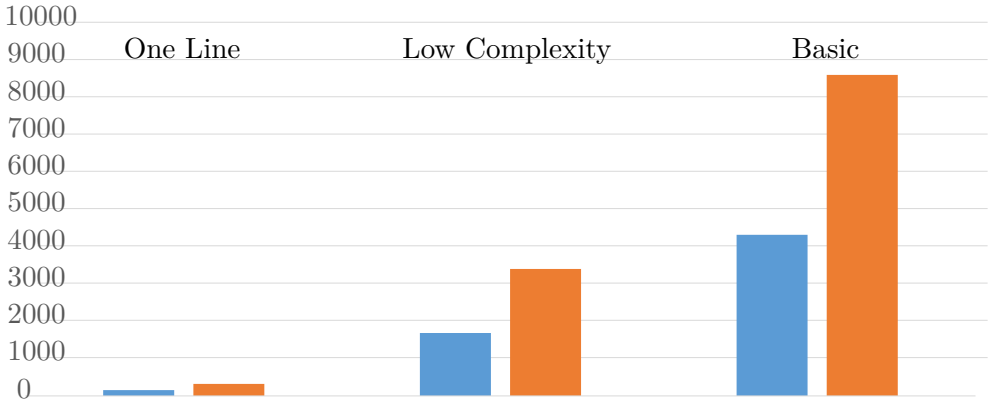

Figure 5.5: The average parsing time of the different implementations. The blue and orange columns represent single and double visitor processing time respectively. The more detailed information could be found in the Section 6.1.1

The two visitors approach could seem preferable in terms of the workflow and responsibilities distribution. The parts of the code which stand for type discovering would be separated from the string extraction code, which in return would come handy as of simplifying massive logic into smaller parts increases understandability of the code.

The two visitors approach would include the type-assignment visitor and

string extraction visitor.

5.3.1 Type Assignment Visitor

The type visitor would assign the specifically designed types to the nodes of the tree for further usage of that information to correctly choose the way of the string extraction. The types would not only represent a special kind of the expressions but would also contain the logic of string extraction of a particular type. The types implemented in the prototype would have a common base interface to maintain specific API, convenient for future development. Moreover, the common parent interface would allow storing all the types in the single map. The base interface would be named IType and would have the following methods:

- getNodeType(): the method would be available across all the types, as some of the string extraction logic would require to know the type of the node. The node type could be one of those included in the list:
  - INTEGER: the integer as a number itself, the declared primitive variable of type int or a variable of the reference type Integer
  - DOUBLE: the double as a number itself, the declared primitive variable of type double or a variable of the reference type Double
  - CHARACTER: the character as a character itself, the declared primitive variable of type char or a variable of the reference type Character
  - STRING: the string as the sequence of characters surrounded by the double brackets, or a variable of the reference type String
  - UNKNOWN_REFERENCE: this type would represent the reference types, which are not String nor Object
  - MESSAGE_FIELD_ACCESS: the expression of the field access of the message
  - CONSTANT_FIELD_ACCESS: the expression of the access of the imported constant enum
  - MESSAGE: the expression, representing the reference to some message variable.
    Examples:
    - "$M"
    - "CUSTOM_METHOD()" considering that this method would return a message
    - "m" considering that this variable was declared before with the message type

  - READABLE_MESSAGE: the expression representing the reference to some read-only message variable
  - CONSTANT_ENUM: the expression representing the reference to some of the imported enum constants.
    Example: ".NewOrderSingle"
  - FIELD_ACCESS: the expression representing the field access.
    Example: ".fieldName"
  - SPECIAL_FIELD_ACCESS: the same expression as the FIELD_ACCESS, but verified to be the actual field of the message.
    Examples:
    * ".FIXFieldName"
    * "[30]"
  - EXPRESSION: is the type, used to define the node, having two or more children
  - ARGUMENTS: is the type, used to aggregate the types of the arguments of the method

- getSymbolPosition(): gets the position of the symbol during the string extraction. The symbol position is stored relative to the actual position in the resulting string and could be later referenced by the error provided by the compiler. The detailed description of what would be the symbol position can be found in Section 5.3.2.2

- setRequiredType(type): sets the required type to the node and its children, if the node type is Expression. The stored required type would be used during the generation of the output expression, to appropriately cast the expression to the desired type.

- translate(visitor): extracts the string from the current type using the visitor to store the symbol position information and process sub expressions if necessary.

The type assignment visitor would require environmental data, mainly fix data and custom functions metadata. These data would be used to determine the validity and provide the replacements of the custom functions and variables, and fix constraints.

The type visitor would assign the type to each node of the Message Field Access tree member. The tree can be viewed on the tree diagram (Figure 5.2).

The Message Field Access node would receive one of the following types:

- If the children would have the types of Message/ReadableMessage and the Special Field Access, then the node would be assigned to the Message Field Access type.

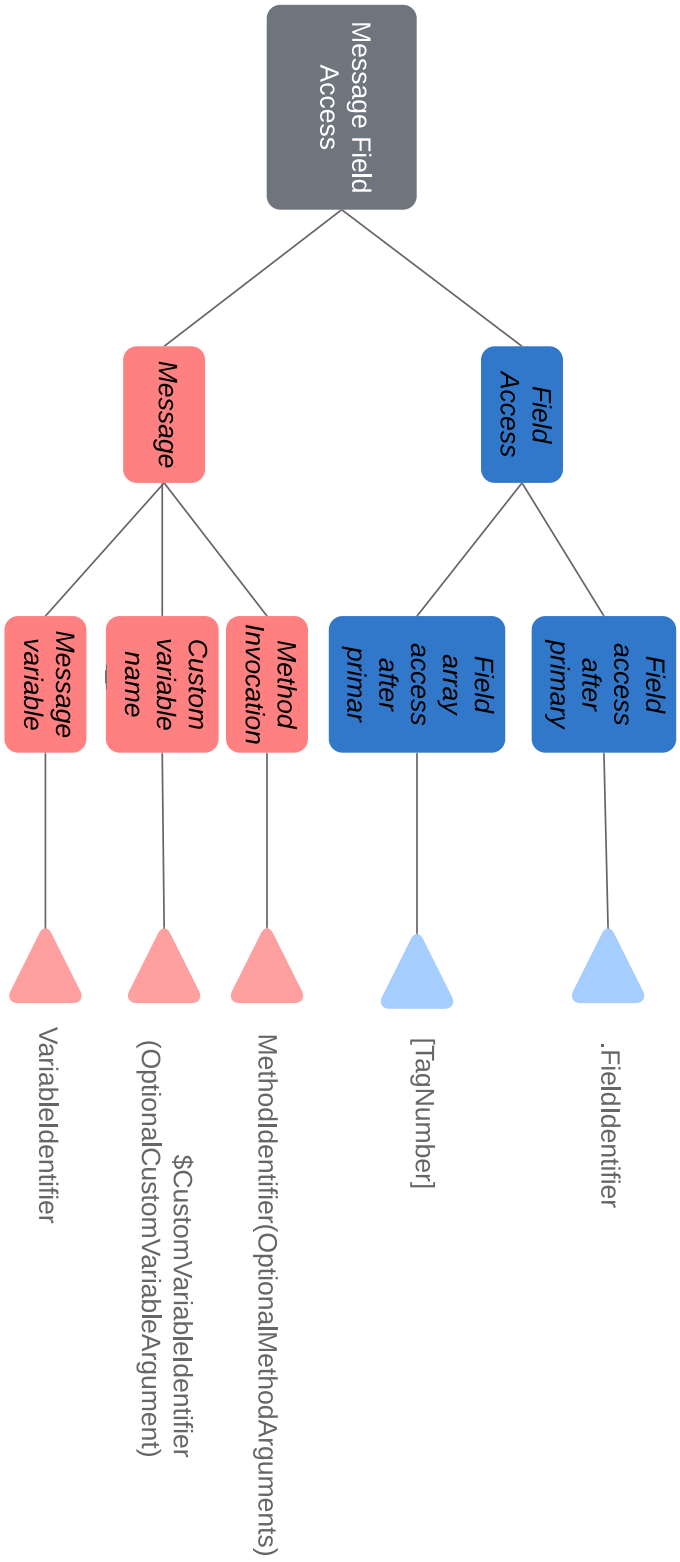

Figure 5.6: The tree of the added custom rules and their relation to each other

- If the children would have the types of Constant Enum and the Field Access, then the node would be assigned to the Constant Field Access type. The type contains a replacement for the constant enum value alias and the position of the alias. The translate method would save the position and return the replacement.
- In case the first child is not a Message, ReadableMessage nor Constant Enum, the node will be assigned to the Simple Expression type, where the translate method would pass the control of string extraction to its children.

In any other case, the input would be considered being invalid, and there would be generated an appropriate error, describing what the expected input was and why the engine failed to process the script.

The parent of the Message Field Access node would receive the type according to the following logic:

- If the Message Field Access node is the only child and has no other siblings, the type of the node will be adopted by the parent
- If there are several children, the parent would take an Expression type regardless of the types of the children

The Message Field Access, alongside with other types, are complex types, which would require more in-depth description. The descriptions would be in separate sections, providing the information about main types and their subtypes.

5.3.1.1 Message Field Access Type

The Message Field Access type contains information about the message and the field which is being accessed. This type has extended API to extract the string, as there are different types of the expressions, where message field access could take place. The API would be extended with the following methods, which would be used instead of the translate() method, defined in the base IType interface:

- tanslateGet(visitor): this would process current message field access getter into the string. This method would use the saved required type to select appropriate getter. If the required type is undefined, the method will generate a general getValue call, with the return type Object.
- translateSet(visitor, argument): would generate the setter method choosing according to the argument type.
- translateSet(visitor, messageFieldAccess):
would generate a special method to copy the field of one message to another.

The setter methods would report the error if the message is read-only since the fields of such a message could not be changed.

5.3.1.2 Message Expression

The Message Expression is an abstract type, which would be set to either verified Message or read-only Message expression. The Message expression is a variable or the return value of the method with type of the message, representing the FIX message in case of the prototype, which was described in Section 2.1. This class defines a new method to pass the string extraction control to the subclass which contains information about the message expression. The Message Expression itself sets the cast to the expression if the required type was set before, saves the position and calls the child to finish the string extraction. The subtype would only need to move the current position and return the replacement for the message expression, as the symbol position would be already saved.

The subtypes are the following:

- Special Message Variable: the message reference by using custom syntax with the dollar sign (Example: $M). This type contains replacement which was derived from the container of custom expressions.
- Special Message Variable With Argument: same custom message reference, but with the parameter. The parameter would be an index of the message in the special message array. The type contains replacement and argument string to build the method appropriately.
- Message Variable: the variable which was declared in the script with the message type. The type contains only the name of the variable and does not change it.
- Custom Method: the reference of the custom function. The custom function could return any type, but in case of message field access, it would be checked that the type is either a message or read-only message. This type contains the replacement method which was derived from the container of custom expressions and the arguments to be inserted into the method.

5.3.1.3 Field Access Expression

The Field Access Expression would be assigned with one of the two types: Field Access type or the Special Field Access.

The Field Access type represents the regular field access which is not using the alias for the message field. It contains the name of the field and the symbol position. It would either not be changed at all or would be used for the Constant Field Access.

The Special Field Access type represents the field access in its usual form, where the field name represents the field of the message, or in array field access style. The tag of the field is retrieved from the container of custom expressions and stored for the string extraction, where the getter would be generated. The tag could also be used by the Message Field Access type, to extract the information and build type-specific methods according to the context.

5.3.1.4 Arguments Type

The Arguments type is used to pass the type of the arguments of the custom method to the actual argument expression and its children.

5.3.1.5 Reference and Primitive Types

The Reference and Primitive types are used to define the types of simple expressions and pass information, such as required type to the other types. Furthermore, some of the types have unique methods to generate getter for message field access. Those types include:

- CharType
- DoubleType
- IntType
- StringType

All of the reference and primitive types have the method to retrieve the cast expression, which could be placed before any expression on request. They have predefined string which they would give as a cast expression, and the UnknownType stores the type name, thus the cast expression is dynamic for this type.

5.3.2 String Extraction Visitor

The tree is processed once more using the information obtained from type assignment visitor. The main functionality of the String Extraction Visitor would be the assembling of the compilable Java code by altering the expressions according to their type and position. Thus the return value for the visitor‘s base methods would be the string. The aggregator should be used to combine the string results from the children nodes. The aggregator would be a method of the visitor, which contains aggregation logic for the three arguments being:

- The previous aggregated strings.

- The new string.
- Separator

The aggregation logic would only vary in a way which characters to use as a separator. There are two possible candidates for the separator:

- The whitespace is used in any other case to separate the tokens from each other

5.3.2.1 String Extraction

There is the default behavior for visitor‘s methods, which is to visit all the children and aggregate the results using a special method. This behavior is suitable for most of the nodes, as no changes should be performed in the context of non-custom code.

There are several patterns the overridden method‘s algorithm could follow:

- Select different aggregation approach: by default, the separator would be whitespace, but in case of block statements, the statements should be separated by the newline
- Skip the translation of certain nodes: the java blocks are made possible by having a specific rule in the grammar so that every block preceded with JAVA keyword is a so-called Java Statement block. The keyword JAVA is omitted by the String Extractor Visitor, and the block stays untouched. As it would be still a block, java compiler would notice if any of the variables declared in the block are used outside of it and that would generate a user-friendly error message. The custom variable prefix should be omitted as well as it does not have any useful information since the script was already parsed. The end of the file has to be handled with the special ANTLR symbol as well, so this node should be skipped in the same manner as the JAVA keyword and custom variable prefix.
- Getting the type from the node and using a type to extract the string: as was stated before, the types assigned to the nodes would possess enough data to transform the expressions to the compilable Java expressions. For some of the nodes, there should certainly be the saved type, and since every type can handle the translation, the control can be passed to that type (e.x.: the Expression node, Message Field Access node).
- Building optimized binary operator method based on context and type of the children: there are several nodes, which require a bit more context than just the syntax context. The Equality Expression node would check if the equality or non-equality expression takes place and if it is, the method will pass the control over to the special class, responsible for building optimized operators. That class would take the nodes and

their types and based on the types and operator; it would decide which operator method is suitable for that case. Mainly the types are needed to check if there is a Message Field Access on either of the sides of the binary expression, as in that case the method would be changed to fulfill the optimal extraction of the field, or both field if the Message Field Access is on both sides. The same strategy goes for the Relational Expression node, where such binary operators as less than, greater than, less or equal, greater or equal are used.

- Building the setters and getters during the assignment context: the assignment algorithm is close to the binary relational expressions approach, but in assignment context, if the left-hand side is the Message Field Access, the special setter should be generated, taking either the argument of arbitrary type or the Message Field Access. If the left-hand side is anything but the Message Field Access, the control goes to the children and the string extraction would be performed as if they would be in separate expressions.

5.3.2.2 Symbol Position Storing

The string extraction in many cases would move the tokens from their initial position and even transform the sequence of tokens to one or other way around, so the track of the modifications takes place in string extraction visitor. For these reasons, the term of Symbol Position is introduced. The Symbol Position is an information of the original position of the token or the sequence of the characters in the script. The position would include the following information:

- Line number
- Position of the first character of the sequence in the specified line
- The absolute start and stop positions

The visitor would store the map where key and value would be the final absolute position and the Symbol Position associated with that position.

To implement the Symbol Position, the extension class for the org.antlr.v4.runtime.Token class would be created, as the provided class lacks the functionality of combining two or more tokens into one. The designed class would take the ANTLR Token class as a constructor parameter and extract all the necessary information about the token which could potentially be used later during the tracing the tokens back to the origin.

The stored Symbol Position information would be later used for building the compilation feedback messages in case any compilation issue would be detected. The compilation feedback would be represented in java like format.

- The first line of the error would inform the user of which script failed to be compiled, on which line the error has occurred and the message of what seems to be wrong. The messages would be provided by either the java compiler or the engine in case of internal errors such as incorrect custom variable alias, nonexistent field alias or the assignment of the read-only message field
- The second line will provide the unchanged line extracted from the original script
- The third line is responsible for underlining the problematic symbols according to the error

Chapter 6

Testing and validation

The testing and validation consist of two parts, where the testing would be benchmarking and comparing the performances and validation would be verifying the output of the new scripting engine implementation and comparing functionality and validity of the resulting scripts between the old scripting engine implementation and the prototype.

6.1 Benchmarks

The benchmarking is necessary to check the performance of the new implementation and assure that the time needed to process the scripts do not pass the threshold of the pleasant user-experience. The benchmarking could also be used in the future improvements of the engine in terms of improving the processing speed.

The benchmarks should cover the processing of the script as it would take place in day to day use. That would be every step of the processing including the parsing of the script stream by ANTLR, analyzing the types and string extraction.

All the benchmarks would be performed on the same computer with the following specifications:

- Processor: Intel Core i7-7700K @ 4.20GHz (8 CPUs)
- Memory: 16384MB DDR4 2400 MHz
- Operating system: Windows 10 Enterprise version 1809 build 17763.437
- JRE: 1.8.0_152-release-1248-b22 amd64

The framework for benchmarking was chosen to be the JMH, as it is a tool to build and run the benchmarks, which can provide results with nanoseconds

precision. The benchmark is capable of measuring the average time per operation, where the operation in this particular case would be the processing of the single script by the ANTLR, engine or the compiler. [5]

The results would be represented as a data table with the three columns:

- Name: the name of the executed benchmark
- Score: the average time to process a single script. The measurements Measured in nanoseconds. The measurement is time-based, which means the operations would run only during the specified period. The period was set to 10 minutes per iteration, where iteration is set of the executed operations. There would be 5 warmup iterations and 5 main iterations.
- Error: the maximum deviation from the average score, which shows the overall steadiness of the benchmark. Measured in nanoseconds.

6.1.1 ANTLR Benchmark

The results of parsing performance were previously mentioned in the realization section of abstract syntax tree processing to show that the single visitor approach would not benefit the overall speed. The ANTLR's parsing speed depends on many factors, such as the design of the grammar and the complexity of the script. The benchmark was performed on three scripts:

- Basic script representing the usual parsing complexity script. The complex expressions take places, such as a switch, the loop, and if-else statement. The additional custom DET grammar rules are used in this script.
- Low parsing complexity script written in pure java.
- One line script: a single expression of return.

The results can be seen on the Table 6.1.

Benchmark Description	Score	Error
Basic script, single visitor	4324.098	54.233
Basic script, couple of visitors	8616.695	96.891
Low complexity script, single visitor	1689.801	17.749
Low complexity script, couple of visitors	3407.846	37.178
One line script, single visitor	156.811	5.410
One line script, couple of visitors	321.881	6.490

Table 6.1: The result of the benchmarks of the processing of the scripts by the generated base visitor.

The error margin is within 1.5 % of the overall score. Thus the benchmark could be considered to be in a steady state. As can be seen from the

Benchmark	Score	Error
Basic script, new engine	289898.036	5843.316
Basic script, old engine	498067.283	7139.693
Low complexity script, new engine	198566.901	2671.938
Low complexity script, old engine	131451.030	2219.262
One line script, new engine	83755.539	1873.524
One line script, old engine	71897.885	1938.929

Table 6.2: The result of the benchmarks of the processing of the scripts by the different versions of engine.

results, the double visitor approach takes two times more time compared to the single visitor. Whereas the double amount of time could be an argument to improve the solution using the single visitor approach, the absolute value of the marginal time increase is quite low compared to the rest of the processing.

6.1.2 Overall Processing

The old engine implementation uses over a hundred of different preprocessors and modifiers which are not natively supported in the new implementation. That makes comparing the different implementation rather estimated when using all the functionality of the old scripting engine, giving a significant advantage to the new version.

To eliminate the possibility of misjudging the results in favor of the improved engine, the benchmarking will be performed with as few base preprocessors and modifiers used in old implementation as would be needed to transform the benchmarking scripts. That set of the preprocessors and modifiers are logically included in the semantic parsing of the prototype.

In that model, the priority goes to the old implementation, as most of the old preprocessors base logic is implemented in the visitors of the new engine and hence are the part of the engine which cannot be turned off or removed.

The benchmarks were performed on both of the engine implementations with the same scripts used in the ANTLR benchmarks. The results are following:

It could be seen from the results, that in case of the low complexity script and the short one line script, the new implementation is slower up to 50 %. The reason why the old implementation did so well in the scripts where custom expressions are absent is that the engine did not call any of the functionality which is responsible for the custom logic, apart from the matching of a few regular expressions against the script, to check if there is any custom code. The optimizations are not performed so that the optimization would end after the first phase. On the other hand, the processing of the script with the custom code took 60 % more time for the old engine.

The deceleration is related to the invocation of the methods responsible for processing the custom code and increased amount of optimization iterations. The new implementation‘s complexity is mainly dependant on the length of the script, rather than the content, however as there are few optimizations for the custom expressions, the presence of that expression in the script would slightly decrease the processing time. That could be seen from the benchmark result on Table 6.3, where the basic complexity script is used without the custom code.

Benchmark	Score	Error
basicProcessingNoCustom	310527.923	4513.265
basicProcessing	289898.036	5843.316

Table 6.3: The result of the benchmarks of the processing of the scripts with and without custom code by the prototype.

To sum up the processing benchmark results, the time consumption of the script processing of the old scripting engine increases with the script's length and the more of the custom code there is in the script, the more processing steps would be performed resulting in the relatively long processing. On the side of the prototype, the complexity of the parsing would increase with the script length as well. However, the presence of the custom code would shorten the evaluation time overall.

The influence of the script length to the processing time could be reviewed in the following benchmarking model. The old scripting engine would perform the processing on the pure Java code. The modifiers and preprocessors would be absent from the engine. Thus the only alternations would happen in code optimizations stage only. The prototype would be tested with the same scripts and without any data from the environment, such as custom functions, custom variable and FIX constraints. The scripts which would be used in that benchmarks would be of similar complexity and length as the ones, described in Table 6.2, but modified to be without any custom code. The long script would be added to show the relation of the length to processing time. The results could be seen on Table 6.4. From the benchmark is could be seen

Benchmark	Score	Error
Long script, new engine	378314.138	6193.141
Long script, old engine	820613.283	5893.914
Basic script, new engine	241701.036	5843.316
Basic script, old engine	412580.591	4810.353
Low complexity script, new engine	179203.193	3097.632
Low complexity script, old engine	96930.583	1984.317
One line script, new engine	80348.671	1459.959
One line script, old engine	50659.418	1253.076

Table 6.4: The result of the benchmarks of the processing of the scripts without custom code by the different engines.

While for the large script the average time increased by 56% compared to the basic script, the increase for the old engine was 99%. That results in the conclusion, where the prototype processing complexity is less than the one in the old engine.

6.2 Validation

The validation would be performed automatically as well as manually. The automatic validation would take two inputs, both representing the processing result of the old or new engine, eliminate the separators, such as tabulators, whitespaces, and newline and compare the resulting data. Out of 241 scripts, 189 were identical to each other. The rest of the scripts comparison results contained nonsignificant differences, such as :

- The different path to the custom method, as not all of the used custom methods were mimicked in the testing module of the new scripting engine
- The missing comments in the new scripting engine output, as the grammar completely omits the comments

That proves the validity of the scripting engine and the compatibility with the previous generation scripts.

More distinct differences were found in the manual testing while using the next generation scripts to be processed by the old scripting engine.

The introduced functionality of the engine in the prototype provides intelligent type handling of the message fields. Relevant getter of the field would be built based on the context of the required type. The following code would be a perfect example of the type handling capabilities:

```
int a;
a = $M[30];
```

Provided the $M custom variable is a message, the prototype scripting engine would detect the type of the variable “a” and would request the message field access to place the getter with the return value integer. In that case, the code would be valid and no compilation time issues would be met. While processing the same code with the old scripting engine implementation, the engine would generate common getter with return type Object, which would result in compilation time error, as the types are incompatible. That would make the scripts using the new generation feature set unusable with the old generation engine.

Conclusion

The scripting engine prototype project carried the functionality of the existing implementation and brought up a modern solution to the script parsing and processing problem.

The analysis of the transition from the original version of the engine to the one presented in this thesis shows the importance of well-defined grammar describing the functional syntax of the scripts, which are being processed.

The usage of the proprietary regular expressions as the language recognition tool was analyzed and compared against the grammar-driven parser solution, highlighting the flaws of the original scripting engine and introducing improvements with the prototype solution. Developing the scripting engine environment became more manageable and viable.

Future Work

As the prototype would be integrated into the platform, new possibilities would be opened to create a more user-friendly script development environment. With the existing grammar and contextual data, the following aspects of improving user experience on the front-end part could take place in future work:

- Proper syntax highlighting
- Syntax errors could be visualized to provide immediate feedback to the developer
- Intelligent code completion could cut down the complexity of the script development and, as a result, decrease the time needed to implement the business logic.

Appendix A

Acronyms

FIX Financial Information eXchange

DET Dynamic Electronic Trading

ANTLR Another Tool For Language Recognition

AST Another Tool For Language Recognition

Bibliography

[1] *About The ANTLR Parser Generator*. URL: https://www.antlr.org/about.html. (accessed: 12.05.2019).

[2] *DET Technologies*. URL: http://det-tech.com. (accessed: 10.05.2019).

[3] *Fields By Tah - FIX 4.4 Dictionary*. URL: https://btobits.com/fixopaedia/fixdic44/fields_by_tag_.html. (accessed: 11.05.2019).

[4] *FIX 4.4 : Messages by MsgType*. URL: https://www.onixs.biz/fix-dictionary/4.4/msgs_by_msg_type.html. (accessed: 12.05.2019).

[5] *Java harness for building, running, and analysing nano/micro/milli/-macro benchmarks written in Java and other languages targetting the JVM*. URL: https://openjdk.java.net/projects/code-tools/jmh/. (accessed: 11.05.2019).

[6] Lee Oliver. *Trading protocols: More Fix in FX*. URL: https://www.euromoney.com/article/b1321zg719lq20/trading-protocols-more-fix-in-fx. (accessed: 10.05.2019).

[7] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999.

[8] Terence Parr and Sam Harwell. *A Java 8 grammar for ANTLR 4 derived from the Java Language Specification chapter 19*. URL: https://github.com/antlr/grammars-v4/blob/master/java8/Java8.g4. (accessed: 11.05.2019).

[9] Javin Paul. *What is FIX Engine in FIX Protocol*. URL: https://javarevisited.blogspot.com/2012/01/what-is-fix-engine-in-fix-protocol.html. (accessed: 11.05.2019).

[10] *QuickFIX*. URL: http://www.quickfixengine.org/. (accessed: 11.05.2019).

[11] *QuickFIX/J Java Open Source FIX Engine*. URL: http://www.quickfixj.org/. (accessed: 11.05.2019).

[12] *What is Parser?* URL: https://www.techopedia.com/definition/3854/parser. (accessed: 12.05.2019).

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format