Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Generating Reflexive Parsers for Matching Syntactic Patterns

Supervisors:
Dávid J. Németh
PhD Student

Simon Thompson
Visiting Professor

Author:
Szilárd Horváth
Computer Science BSc

Budapest, 2020

Table of contents

1. Introduction

Developers often encounter the need for code transformations. A common code transformation is code refactoring. Code refactoring means that the programmer changes his or her already existing code in a manner that its behaviour does not change. Doing this manually over large bodies of code is exhausting and mistakes easily can be made. There are already existing refactoring tools. To use these tools the programmer needs an enormous knowledge about its internal structure. To become efficient in this, is time consuming and hard. It is easier if we use patterns. On the other hand, in this case the pattern must be parsed too. The parsing of the patterns is hard because it can contain metavariables, and its grammar can differ from the basic grammar of the language of the program. However, recent work [1] provides a solution that generates a parser that can parse the input program and input pattern too. The publication offers an approach where we can generate a parser from the .grammar of the language without huge modifications in it and just a metavariable token This means a pattern can be written by knowing the syntax of the language and some knowledge about metavariables. The syntax of the language should already be well known to the programmer and the knowledge about metavariables can be learned quickly [1]. This makes pattern parsing much easier.

The software implemented in my thesis is based on the publication’s syntactic pattern matching approach with some changes. One of the reasons for my own implementation is despite the fact that the publication offers an implementation, but it is part of another more complex project, therefore it is harder to customize and comprehend. The other reason is that an ongoing project in ELTE called High-Assurance Refactoring Project in short HARP, is working on trustworthy refactoring. A kind of software that deals with pattern matching based on syntax could come in handy for the project HARP, because they are working on a language that describes trustworthy refactoring [2][3]. All in all, to meet the needs and to simplify the usage I decided to implement a software that is based on the ideas in the publication but with some differences, that are required to fit in to the HARP project. The differences will be highlighted later in Section 3.

The main goal of the implemented software is to be able to create a parser that can parse metavariable containing patterns. This parser is generated based on the context-free grammar of the language of the program and the pattern. The pattern can contain metavariables that causes holes in the CST. These holes will be filled with the corresponding subtrees of the input program. This way the metavariable gets its actual substitution. The ability of using metavariables in the pattern is important, therefore we want to use them as easily as possible and so with the least modification in the given grammar. This approach also handles this well, because it is enough to define a unique symbol for the metavariable and the parser will take care for the rest.

The inputs of the software are the context-free grammar and a metavariable token which is the symbol that indicates the place of a metavariable. The result of these two is a constructed parser that can be given a sentence of the basic grammar and a pattern containing metavariables. Then the parser can give us a result stating whether the pattern is matching the sentence and if so, it provides us all the metavariable – subsentence assignments.

The software is created in the C++ language with the help of the Qt framework. The reason behind the choice is that C++ is a general-purpose programming language with strong static type safety. Qt is an application development framework based on C++ which can provide an easy to use user interface and a reliable code library.

In the next Section I will talk about how to use the software and how to add the inputs properly. Then in Section 3 I will talk about the internal structure and the different components of the program in details. This section also covers the opportunities of the future works.

2. User documentation

The main task of the program is to create a parser that can parse not just an input program but a metavariable containing pattern from just the grammar of the language and a metavariable token. A patten can contain metavariables that can symbolise an expressions or subprograms basically anything that is valid in the given language. If a match or matches are found the software determines the representative CSTs that contains the information about the substitution of the metavariables. To achieve this some preparations must be made. The program must know the grammar of the language and the unique symbol of the metavariable. If these are given by the user, the software can generate a parser, more precisely the parser table that will be used by the parser. With the parser table the software parses the input program and the input pattern resulting in their representative CSTs. Then these CSTs are compared to determine the result of the match. There can be more than one results therefore the meaning of the metavariable can be different in every different result (figure 1).

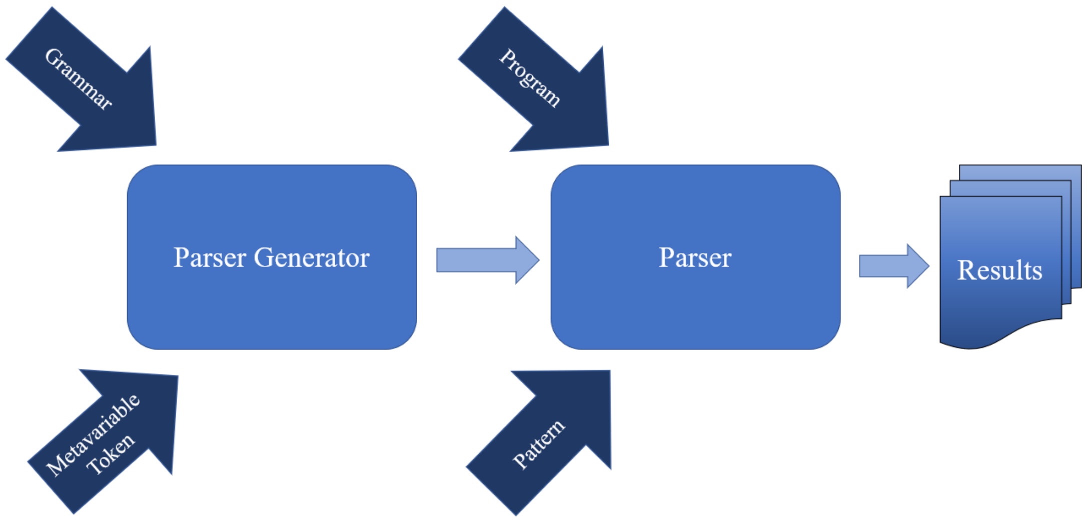


Figure 1: Demonstration of the inputs[1]

This chapter provides a guidance about the proper usage of the software including the minimum requirements to properly run the program, the inputs and their format like the grammar, program, pattern, regular expressions and metavariables. The result and the possible errors of the input also going to be discussed here.

[1] Source: made by own

2.1 Area of application

The software can be used in code transformations. A code transformation engine parses the pattern, finds all its occurrences in the source code and transforms them by a transformation rule. However, to parse these patterns needs not just the language that describes the program, but a modified grammar too that describes the pattern. To create such grammars, is time consuming, inefficient and error prone.

However, the software described in this thesis is based on a solution called Reflexive Parsing, that offers a pattern matching engine that works on syntactic patterns in which metavariables can appear anywhere. More specifically the software generates a parser that can parse the input program and the input pattern and decides if they are matching to each other and if so, it provides all the matching concrete syntax trees. These patterns can be defined easily, because they are based on the syntax of the programming language that is already well known to a programmer and to provide the required grammar, no major modifications are needed in the grammar[1].

The HARP project, during its work creates many and complex patterns, but to use them efficiently, a pattern parser is needed that is generated with the least modification in the grammar of the used language. This software is aiming to fulfil that need.

2.2 Environment

The software was created on Windows 10 32bit. The program does not have any specific requirements. Thereby the minimal requirement is an operating system of Windows 10 or newer.

2.3 Example:

Here I will present a basic example about the usage of the software. For this I will use the following inputs:

Grammar:

S->*pkg* Clss
Clss->*class name* Classes
Classes->*class name* Classes
Classes->

NonTerminals:

S, Clss, Classes

Terminals:

class, pkg, name

Regular expressions (in the form of regular expression – terminal pairs)**:**

class	*class*
[A-Za-z]+	*pkg*
_[A-Za-z]+	*name*

Metavariable specifier:

@

Pattern:

```
myPackage
class _myClass
@ metaV
```

Program:

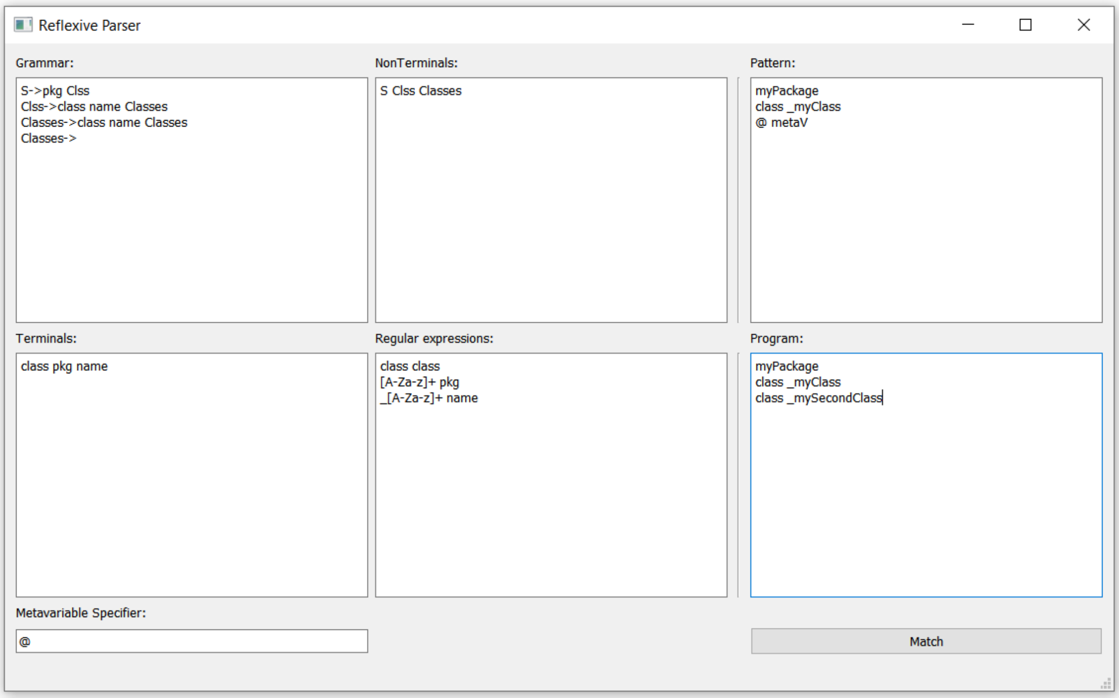
```
myPackage
class _myClass
class _mySecondClass
```

The grammar here describes a basic language. By this language you can draw up programs that must start with a package name, then continues with one or more class declarations. The terminals and nonterminals are defined by the grammar. However, the terminals have regular expressions describing their possible concrete form of occurrence in the program and the pattern. It is important that the names of the regular expressions and the names of the corresponding terminals matches, otherwise the lexical parser cannot process the input program and pattern. In the example the "@"

symbol is used as metavariable specifier because it differs from all of the terminals and nonterminals of the grammar.

In this case the pattern describes a program that has a specific starter package called: “myPackage”, and a specific class declaration: “class _myClass”, then continues with a metavariable. The metavariable here can match to various things. In this specific example the meta variable is expected to be any valid continuity of the code. It could be one class declaration or two or more, even zero, because the language expects at least one package name and a class declaration that is already given before the metavariable. The program that we want to match is a valid program that has a package name and two class declarations.

After we start the program the main window appears labelled Reflexive Parser. Here we can add the Inputs to their appropriate places as can be seen in the following figure (figure 2).



Grammar:
```
S->pkg Clss
Clss->class name Classes
Classes->class name Classes
Classes->
```

NonTerminals:
```
S Clss Classes
```

Pattern:
```
myPackage
class _myClass
@ metaV
```

Terminals:
```
class pkg name
```

Regular expressions:
```
class class
[A-Za-z]+ pkg
_[A-Za-z]+ name
```

Program:
```
myPackage
class _myClass
class _mySecondClass
```

Metavariable Specifier:
```
@
```

Figure 2: main window example[2]

After the inputs were filled correctly, we can start the matching. To begin the process, we click the “Match” button at the bottom-right corner of the window. After the matching is finished the result window will appear. Note that when the result window appears the main window will be blocked until the result window is closed.

[2] Source: made by own

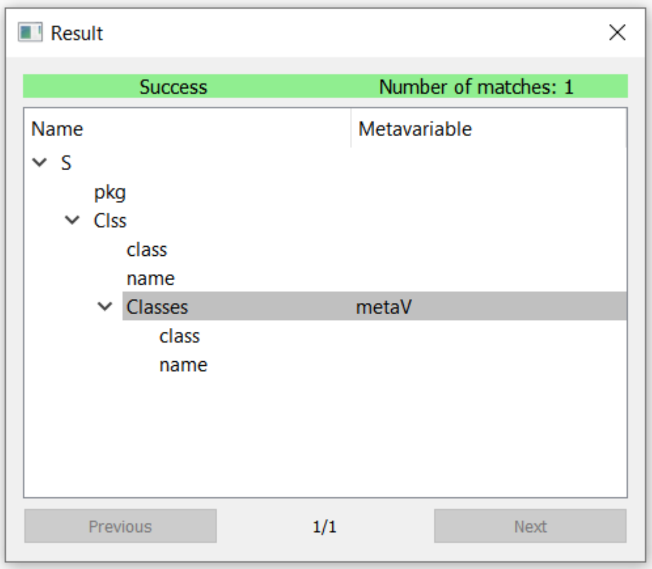


Figure 3: result window of the example[3]

The figure above shows the result screen (figure 3). There are various informations displayed. First at the top of the window with green background colour the result of the match: “Success” and the number of different matches. Here we have only one match, that means we cannot use the Next and Previous buttons at the bottom side of the window, normally these are used to cycle through the different matches if there are more than one. In the middle of the window we can see the matched CST. At the top left of the main widget we can see the root: “S” and on the right from the root, its children. As seen in the figure some nodes have an arrow mark next to them, this means that its children can be collapsed or extended for better transparency. We can see on the figure a highlighted row, that is the metavariable, that has been substituted, in the second column the metavariables name is shown. So, the metavariable has been substituted by the subtree that starts with the Classes token. The representation of the result CST can be imagined as an ordinary CST rotated to left (figure 4).

[3] Source: made by own

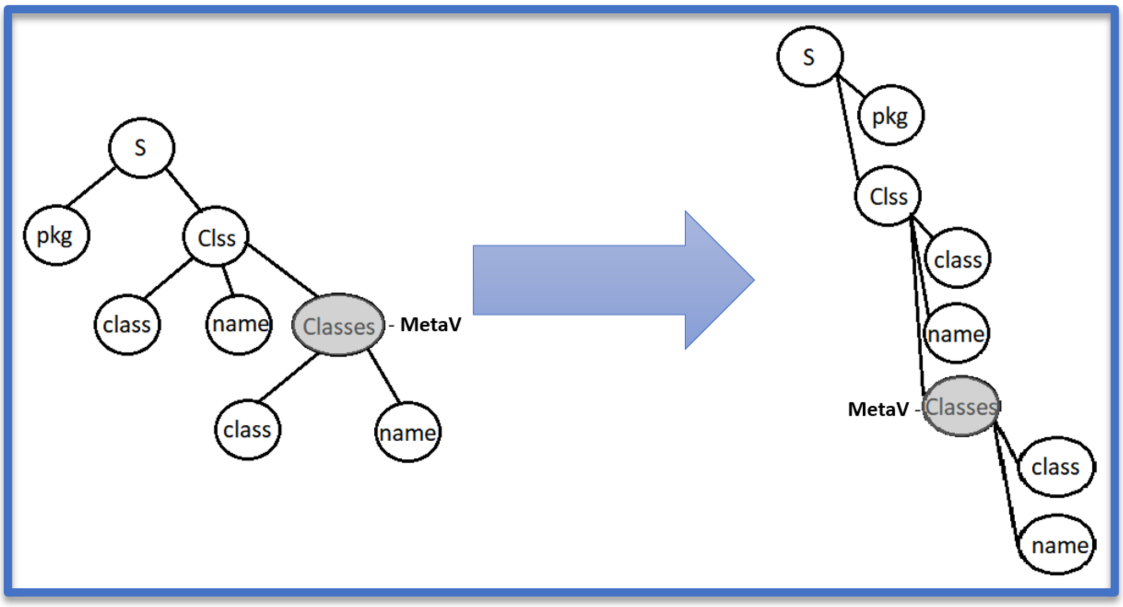


Figure 4: Representation of the Result CST in the result window[4]

2.4 Usage:

The user should fill all the necessary fields properly. The program will warn the user if the inputs are not in the right format or there are missing fields. Then press the Match button to begin the processing. The result window will appear where the user can navigate with the next or previous buttons if there were more than one result mathes. The result window can be closed by the “X” mark at the top right corner. The main window can be closed with the “X” mark at its own top right corner.

2.4.1 The start-up:

The software can be started by the execution of the ReflexiveParser.exe file. It can be found under SzakdolgozatHorvathSzilard\FuttathatóÁllomány\ReflexiveParser.exe.

2.4.2 The input:

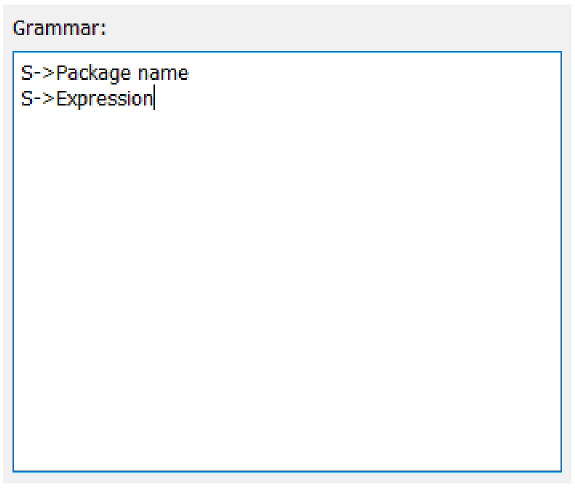
The software has a user interface. The input data should be given by the user through the user interface in the corresponding fields in the window appearing first after the start of the software. The user must provide a context-free grammar and its terminals

[4] Source: made by own

and nonterminals. Furthermore, the user must give the pattern and the program text, the regular expressions and the token of the metavariable. The metavariable token is a unique token that must differ from all the terminals and nonterminals of the grammar.

The grammar:

The grammar must be a context-free grammar. This means the left side of a rule can only contain one nonterminal. In the program the rules of the grammar must be added in the Grammar field. Each line can contain one rule, the left and right side of the rules should be separated by an arrow: "->" (S->Begin). If the right side of the rule contains more than one token, than they must be separated by one spacebar (Program -> Begin Main End). The epsilon rule can only be given as an empty right side of a rule (S->). If a rule has more different right sides each one must be given in new lines with the left side of the rule (figure 5).



```
S->Package name
S->Expression
```

Figure 5: Example rules in the main window[5]

Terminals/nonterminals:

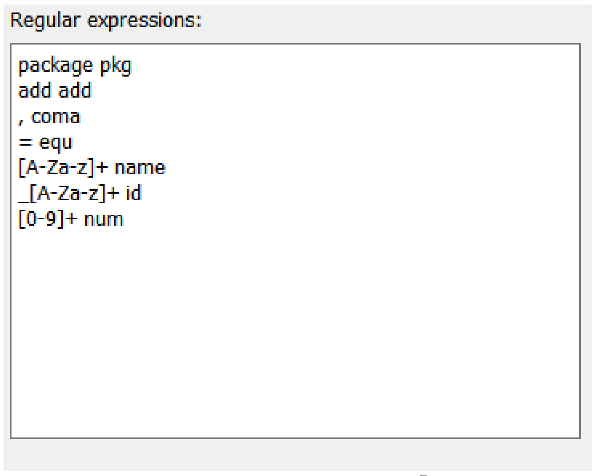
The terminal and nonterminal symbols must be separated by enters or space bars or both. They must be given in their specified field in the program after the grammar. The terminal symbols must on sync with the names of the regular expressions. The symbols used as terminals or nonterminals can be anything but white space and "new

[5] Source: made by own

line”, because these symbols are used as separators in the input fields of the terminals and nonterminals.

Regular expressions:

Each line can contain only one regular expression and name pair separated by a spacebar ([a-z]+ variable) (figure 6). If special characters wanted to be used without their special meaning in regular expressions, a “\” character must be inserted before them to escape their special meaning (\+ add, \\ backslash).The ordinary symbols of regular expressions can be used to specify a regular expression: “+”, “*”, ”[]”,”.”, etc. The regular expressions will only match on full matches, so the “^” and “$” symbols are not required in the expressions. To find out more about the regular expression you can visit the Qt official documentation[6][6].



```
package pkg
add add
, coma
= equ
[A-Za-z]+ name
_[A-Za-z]+ id
[0-9]+ num
```

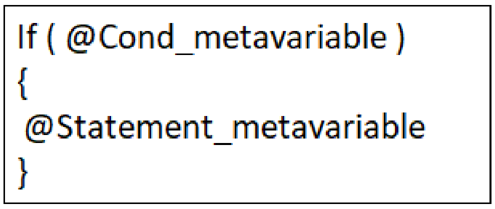
Figure 6: Regular expressions example[7]

Program/pattern text:

The program and pattern texts must be added to their corresponding fields. These must be recognizable by the grammar otherwise the matching will fail. The only exceptions are the metavariables in the pattern. The program should not contain any metavariables. The pattern can contain any number of metavariables. However, two metavariables cannot have the same name. Metavariables can have different names if so, they are treated as different variables and in the result, they can have different substitutions.

[6] https://doc.qt.io/qt-5/qregularexpression.html#details
[7] Source: made by own

In the pattern the metavariable token must be followed by a name identifier: “$ MetaName”.This name identifier must be matching on the following regular expression: "[_a-zA-Z][_a-zA-Z0-9]*". In the example just above the “$” is the metavariable token and “MetaName” is the name of the metavariable. If more metavariables needed in the pattern it is enough to specify different names for them in the pattern.



```
If ( @Cond_metavariable )
{
 @Statement_metavariable
}
```

Figure 7: Metavariable example[8]

On the figure above there are two different metavariables specified, this means that they can match on different substitutions on the input program (figure 7).

Metavariable specifier:

A metavariable token must be specified for the correct functioning. The token must be unique, so the grammar must not contain any terminal nor nonterminal symbol that is the same as the metavariable token It is important that here only the specific symbol must be added not the names of the different metavariables. Only one symbol can be defined as a metavariable token because this symbol specifies if the concrete metavariable and that the next token will be the name of the metavariable. However, the token can consist more than one characters (“My_Meta_Token”).

2.4.3 The output:

The output can be viewed on a new window. This window is called result window and appears automatically after the matching. On the top of the result widow, it can be seen if a result was found, if so, the indicator becomes green and shows Success, it also can be seen that how many different matches has been found (figure 8). The different matching CSTs can be viewed by using the Next and Previous buttons. The CSTs also

[8] Source: made by own

highlights the metavariables substitutions. If there were no matches the indicator becomes red and shows Fail (figure 9).

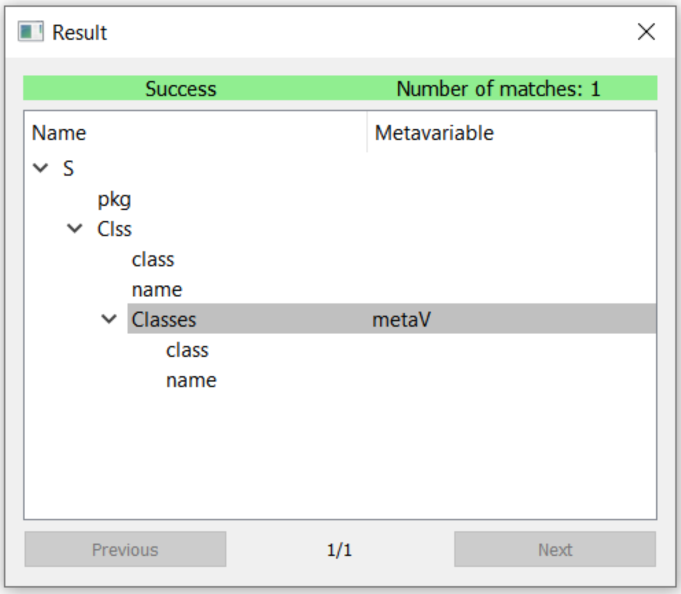


Figure 8: Successful result window[9]

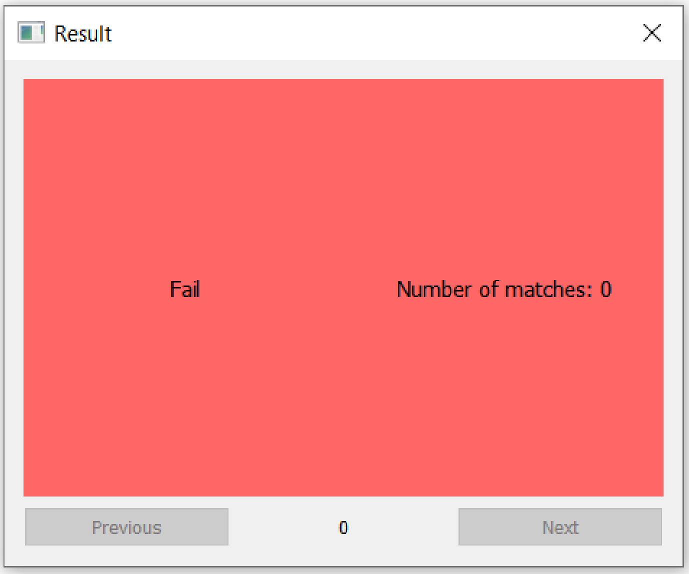


Figure 9: Failed result window[10]

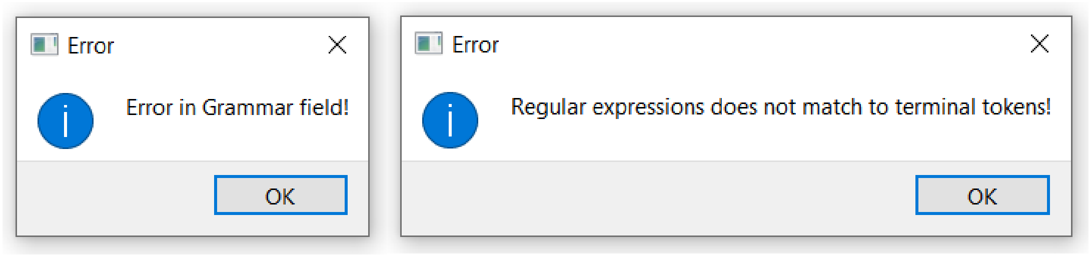
The result CST can be viewed in a tree model representation, where the leftmost symbol is the root, then its child or children on its right etc. For the easier transparency, the children can be collapsed so the same level symbols can be easily identified. The

[9] Source: made by own
[10] Source: made by own

metavariables column is highlighted, and its name can be seen in the second row named: Metavariable. This highlighted node is the root of the sub tree that has substituted the metavariable in the matching.

2.4.4 Errors in the input:

If any of the inputs differ from the mentioned above the program will not start the matching. In this case the software will send information about where the problem can be located (figure 10). The program does not start the matching if any of the fields are empty. The most important rules to look out for: in the grammar field you can add one rule in one line, if you want to add more, you must write it in a new line. The rules left and right side must be separated with an arrow symbol "->". Regular expressions are similarly must be given one per line. The concrete regular expression and its name must be separated by one spacebar, and the name must follow the expression. It is also important that the input program must be recognisable by the given grammar otherwise the matching will fail. The pattern must be recognisable too by the grammar with the exception of the meta variables. It is worth mentioning that the metavariable token used in the pattern must be the same that is specified in the metavariable specifier field. While defining the regular expressions pay attention for that the regular expressions names are same as the terminals of the grammar and covers all of them.



Error in Grammar field!

Regular expressions does not match to terminal tokens!

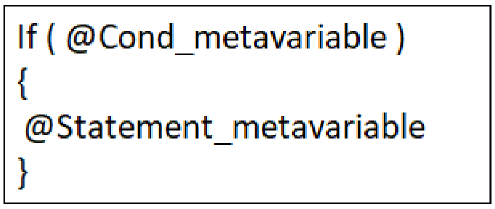
Figure 10: Error windows upon wrong input[11]

[11] Source: made by own

3. Developer documentation

In this section I will present the main difficulties, and their solutions and implementations that helps to understand the actual code. Here will be presented the main building bricks of the software and their logic and implementation. The testing of the program will also be discussed in this section.

The first building block of the program is the parser generator. It is based on an LR parser table. It is generated purely from the context-free input grammar. The problem with LR parser tables that they can contain conflicts in the case of more complex or ambiguous grammars. LR parsing is a great technique, however it cannot handle any conflicts in the parser table as reduce/step, reduce/reduce and even step/step conflicts. To solve this problem, a special type of LR parsing is used, called Generalized LR parsing in short GLR [4]. The strength behind GLR parsing that it avoids all the conflicts by a forking mechanism, although this can result multiply deductions. When the mechanism comes across a conflict during the parsing, it forks into every possible continuation. This technique can not only solve the problem of ambiguous grammars but provides us the opportunity to be able to parse patterns with metavariables [1]. Metavariables are special symbols that can replace single expressions or even whole subprograms in the pattern [1]. Through this we can easily define patterns that describe loops or conditionals (figure 11) or expressions or even more complex subprograms.



```
If ( @Cond_metavariable )
{
 @Statement_metavariable
}
```

11.Figure: Metavariable example[12]

The next step is to use the generated parser. The user should provide a program and a pattern. These are parsed with the generated parser and it outputs the corresponding CSTs. The publication is using potential type annotated syntax trees instead, but in this approach, I am using basic CSTs because we do not need type annotation functions and it also simplifies the implementation. While parsing a pattern we come across the fist issue: what do we do when we find a metavariable? The answer

[12] Source: made by own

to this question is that we treat the metavariable like it is a ambiguity in the parser table and we take advantage of the GLR forking mechanism so that we fork the parsing into every possible type of the metavariable. This solution will create a forest of CSTs that contains all the possible substitutions of the metavariables [1].

Another major issue is the starting point problem. This comes from the fact that we not only want to match whole programs with whole patterns but subprograms with subpatterns. To overcome this issue, I modified the parsing phase in a way that it will parse the inputs from every appropriate starting point. This can result more ambiguity in the program and in the pattern too. However, if the pattern contains metavariables the corresponding CSTs will contain holes int the position of the metavariables [1].

The last major challenge is to match the program CSTs to the pattern CSTs, this is called unification. We compare all the program CSTs with all the pattern CSTs to find all the matching ones. If we come across a metavariable – a hole – in this process, we bound the metavariable to the programs corresponding subtree. This way we know the concrete substitution of the metavariable [1].

3.1 Development environment

The software was developed in the Qt framework version 5.14.2. Qt is a development framework written in C++ that supports cross-platform application development. Qt can be downloaded from the Qt download page[13][6]. There are different versions including free trial version, opensource developer version. When installing Qt, the Maintenance tool appears, here you can select the required tools you need for the development, make sure to select qt 5.14.2. After the installation is complete, open the MainProject.pro in Qt Creator. The MainProject.pro file can be found under the SzakdolgozatHorvathSzilard\Forráskód\MainProject\MainProject.pro.

3.2 Analysis

To parse a pattern, we need a modified grammar that can recognize metavariables. The production of this modified grammar is tedious and error prone. A publication [1] however provides an alternative approach. This publication offers a solution where the

[13] https://www.qt.io/download?

patterns are based on the syntax of the language, that is already known to the programmer therefore making it easier and more effective. Not to mention that it discusses a parser generator that generates a parser that can parse metavariable containing patterns and for the generation it is enough a slightly modified grammar and a metavariable token. This approach fits into a recently ongoing project in ELTE called High-Assurance Refactoring Project. This project focuses on thrust worthy refactoring. The reason behind my own implementation that differs at some point from the publications is that the publication provides an implementation however it is part of a much bigger whole.

The technique presented by the publication [1] is the reflexive and automated syntactic pattern matcher. The main difficulties that the publication focuses on are the table-driven parsing, the parsing of the syntactic pattern, the starting point problem, forking the parser and the unification [1]. To use table driven parsing, we need a parser that can recognise the given language, the publication suggests supplementing an already existing GLR parser generator with the handling of metavariables. However, in my implementation I created my own GLR parser and is generator. To parse the syntactic pattern, we need a parser that can recognise not only the language itself but also the metavariable token in the pattern, because we do not want to supplement the grammar with metavariable rules. The starting point problem about a general problem. While parsing the pattern the parser needs a starter token that is usually the left symbol of the first rule. In this case the pattern we describe can only start at the beginning of the program. However, patterns can describe subprograms, subexpressions, so need a way to parse these patterns too [1].

When we have a parser that can parse the required language, we need a solution that handles the metavariables occurring in the parsing phrase. The solution is to look at the metavariables as conflicts in the parser table. This way we can create a forking mechanism that forks the parser at the required points. The unification is about the concrete matching of the program and the pattern. The publications parser creates ASTs from the processed input program and pattern [1], but in my implementation the parser creates CSTs. In the unification phrase the CSTs of the input program and the input pattern are compared. The CSTs of the pattern can contain holes where the metavariables were. The comparison starts at the root of the CSTs. If a metavariable is found than it gets substituted by the concrete sub tree of the program. Otherwise the comparison is a normal tree comparison examining all the nodes of the trees. This way we can determinate the matches [1].

3.3 Main challenges

3.3.1 Parser generator

To match a program to a pattern the first thing to do is to parse them. To parse these inputs, we need a parser based on the grammar that can recognize the program and the pattern. Therefore, the first problem to solve is the parser generation. In this section of the program a parser table is created based on the input grammar. The grammar must be a context-free grammar. The table is a GLR parser table. Parser tables can hold information about the states that a program can be in. Each state has a specific action that is executed based on the next token read from the input that the software parses. These actions can be the same as in an LR parser table: step when the read token is consumed and the parser steps over into a new state; reduce when the state stack pops as many states as the size of the current reduce rules right side and the next token to process becomes the left side of the rule. In case of more complex or ambiguous grammars these state token cells in the table can content more than one action. This leads to conflicts in the parsing phase. These conflicts can be step/reduce, reduce/reduce and even step/step conflicts. However, the GLR has a unique mechanism that provides a solution to resolve these conflicts. It is a forking mechanism that can work pretty well with metavariables too [1]. The parser thanks to this mechanism, when comes across a conflict forks into every possible action. This causes more parsing branches where can be dead end branches as well, if the branch comes to be a dead end the parser discards it. The process with metavariables is very similar, when the parser come across a metavariable it forks into every possible interpretation of the metavariable [1]. The metavariable token is not present in the parser table, it is another attribute of the parser, and it is handled independently in the parsing process. The result of the parser generator is the representation of the parser table. It is important that it creates the table dynamically and not generates a code that than must be compiled. This implementation is based on the publications table driven parsing approach [1]. To fill up properly the parsing table we need to create the canonical sets of the grammar to do that we need the implement the closure and read methods too. These will work on LR1 elements. It is enough to work with LR1 elements because most of the knowledgeable languages are near LR1 languages. The ones that are LR k languages can be traced back to LR1, but this causes conflicts in the parser table. However, it is not a problem because the parser uses the GLR forking mechanism that solves the problems caused by conflicts [4].

3.3.2 GLR parsing

Parsing the program and the pattern is the next critical step to achieve the goal of the software. Because the input program and the input pattern are based on the same grammar both of them can be parsed with the same parser [1]. The program parses the input program and the input pattern, and the results are their representative CSTs. However, the pattern can contain metavariables that makes the parsing harder, because their meaning can be various depending on their position in the pattern. As mentioned earlier the GLR parser can provide a reasonable solution to this problem. The parser considers the metavariable as it were a conflict in the parser table. When it finds a metavariable, it forks into every possible interpretations of the metavariable [1]. As a metavariable can mean many different things, there are possibly more than one correct CSTs to one patten. This means that parsing the pattern results a forest of CSTs. The parsing process is based on the publication [1].

The parsing process consumes the tokens of the parsed input one by one. The first token in the stack of the parser is a pair of the bottom of the stack token and a state where the parsing starts. The parser obtains the executable action based on the current state and the next token in the text. If there are more actions at once, the fork mechanism of the GLR forks the parsing and creates a branch for every action. To fork the parser we need the actual Stack, the actual state , the index that shows that which token is the actual process token in the input text, and the number of the Action that should be executed in the branch.

An Acton can be Step, Reduce, or Error. If it is Error the parsing of the branch fails and does not continue. If the Action is Step the Actual token and the state, the Step Action specified is pushed into the stack. If the Action is Reduce, then the actions specifies a rule. The Stack pops out as many token-state pair, as many tokens consist the right side of the rule. Then the left side of the rule I saved in a variable, and we indicate that the last action was a Reduction. In the next iteration of the parsing if the last Action was a Reduction we do not take the next token from the input, but we take the saved token that is the left side of the reduction rule used last. This most likely will cause that the next action will be Step. This way the parser pushes that nonterminal token into the stack with the appropriate state.

The forking mechanism is also based on the publications forking the parser algorithm. However, there are some differences between the algorithm and the actual code. The main difference is that the publication and its algorithm can handle a user given type, this type is used to tell the algorithm that a metavariable must be matched on this specific type. However, in my implementation the user cannot define these kinds of types, because the target need does not require this kind of feature. Also, the algorithm in the publication distinguishes the symbols if they are composite or elementary symbols. In my implementation these are not distinguished. The forking mechanism is implemented in a sequential manner as all new branches of the parser are pushed back into a sequence. If the forking mechanism comes across a metavariable first it specifies the possible tokens that can be substituted to its place. Then it examines one by one all the possible outcome of the actions by the substituted tokens. If the action is a Step, then the step action is executed, and the result is pushed as a new branch. If it is Reduce, then the reduce action is executed and the result is pushed as a new branch.

The parsing ends when all the branches in the sequence have been processed. This process is also responsible to create the corresponding CSTs for the input pattern and the input program too. This is achieved by recording the rules that were used during a reduce action. This method takes the last rule that was recorded and continues backward in the sequence. It creates a root node from the last rules left side and add its right side as children. Then it searches for the next rules left sides rightmost occurrence that is not a metavariable and is a leaf on the tree, meaning it has no children. A metavariable after a successful reduction will become a nonterminal and that nonterminal is marked with the metavariables name later this marked nonterminal will be part of a reduction. The marked nonterminal cannot have any children because it is a metavariable that have not been matched to anything yet. A metavariable containing pattern will become a holey tree. The hole in the patterns CST could be substituted after a match were found. When the rightmost occurrence of the left side of the next rule was found it gets the rules right side as child or children depending on how many tokens consist of the rules right side. This goes on until the sequence ends. The process ends after all the CSTs were created. This process can only be executed after a parser table was generated.

3.3.3 Starting point problem

While parsing the input program and the input pattern the first question is where we should start the parsing. Normally the parsing starts at the starting symbol that is usually the left symbol of the first rule in the grammar. This question arises because we assume that a pattern cannot start only at the beginning of a program. The ability to match a subprogram onto a pattern that describes a subprogram is important. This way the user can specify patters beginning at a function or expression or even a statement. Based on the publication [1] the previously generated parser table is a strong tool to define the states that are good starting candidates. Good starting candidates are those that contains at least one step action, because this way the parser will consume a token first, otherwise it could not be able to reduce pattern into a valid node so, those stares that only reduce are not good starting candidates [1]. Since we can determinate the good starting candidates through the parser table the next step is the actual parsing. We create a parsing branch for every good starting candidate. The branches failing the parsing are discarded. This process crates all the possible root nodes for the parsed input [1]. This feature is part of the parsing process as an addition. In the beginning of the function the program adds the appropriate parsing branches to the parsing queue. This process is very similar to the forking mechanism the difference is that it does not forks on metavariables possible interpretations but on the possible good starting candidates. In my implementation this process is executed on the input pattern and the input program too. The reason behind this decision is that my implementation of the matching. It matches the full program CST to the full patterns CST. If a pattern only expresses a sub part of the input program no match will be found. For example, if the input program is a sequence of an if statement and a while loop, however the pattern only describes a while loop that is a sub part of the input program, the match will fail. After all the branches were added to the queue the parsing phrase starts. It parses all the branches, discards the invalid ones, and builds CSTs for the valid ones as normally.

3.3.4 Unification

The unification is responsible for the comparison between the input program CSTs and the input pattern CSTs [1]. This is the method that specifies the valid matches. Each piece of the forest of the program CSTs are compared to each piece of the forest of the pattern CSTs. The comparison first examines the representation of the token of the node, if there are equal than it compares its children, if all the children are equal too than the match is successful, and the tree is added to the successful result matches. The process does not stop at the first successful match, just when all the trees have been processed. This way the result will contain all the possible matches. If this process come across a metavariable in the patterns CST a merged CSTs is created. The metavariable here will be substituted with its actual meaning, then the process continues. If this merged tree is a valid match it will also be added to the successful result matches. This merged tree is also holding information about the location of its metavariables. This algorithm is very similar to the unification mentioned in the publication [1], however in their representation the parse input program can only have one AST always. The other big difference is that the publications algorithm tries to match the pattern ASTs root to all of the program ASTs nodes as those were roots [1]. In my representation, the unification always starts the comparison at the roots of the CSTs. The reason behind this, is that the demand is to be able to select a code and then apply a refactoring pattern on it. This means the pattern cannot match the input programs subcomponents. However, a subcomponent of a program can be added as the input program, and the pattern that describes that subcomponent will match. The unification process not just specifies the valid matches but also sorts out the duplicates. This method can only be called after the parsing has been finished.

3.4 Design and implementation

The software implemented in my thesis can be divided into two parts. One of the parts is the reflexive parser generator and pattern matcher based on the publication [1]. The other part is the user interface and its components. The main goal on my thesis is the implementation of the reflexive parser generator for syntactic pattern matching based on the publication [1]. The user interface is created purely for demonstration purposes because the main goal is to use the model an internal library for other projects. In this context the software can be interpreted in a Model / View architecture. The Model is the implementation of the reflexive parser generator for syntactic pattern matching and the View is the User Interface. From here the user interface will be mentioned as UI. These two are totally separated and the model can be reused in its total functionality in other applications. The Lexer is part of the model however it does not play a major role in the actual functionality of the model, it rather supplies comfort mechanism in the software. (figure 12)

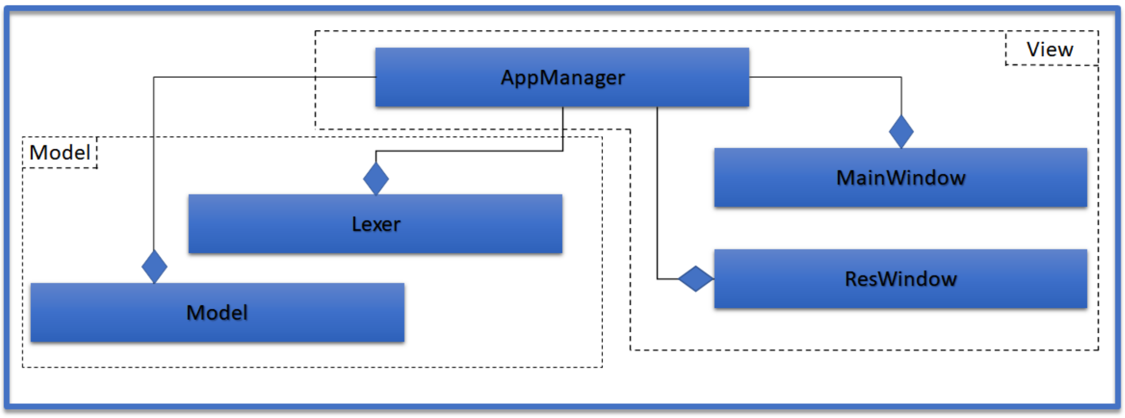


Figure 12: structure of the software[14]

3.4.1 Model

The main class of the model is the LRParser (figure 13). This class contains all the main functions of the program like the Generate() function that creates the required parser based on the input grammar, and the Parse() function that parses the input program and input pattern and produces their CSTs and even the CalValidMatches() that is responsible to determinate the valid matches. There are other functions too that are helping the main functions in their work.

[14] Source: made by own

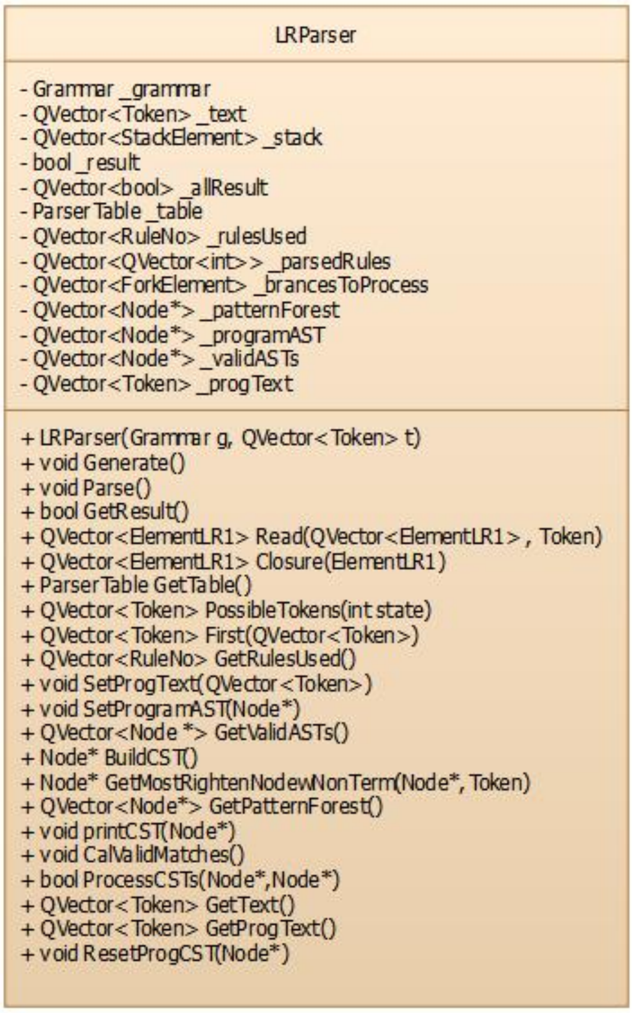

LRParser
- Grammar _grammar
- QVector<Token> _text
- QVector<StackElement> _stack
- bool _result
- QVector<bool> _allResult
- ParserTable _table
- QVector<RuleNo> _rulesUsed
- QVector<QVector<int>> _parsedRules
- QVector<ForkElement> _brancesToProcess
- QVector<Node*> _patternForest
- QVector<Node*> _programAST
- QVector<Node*> _validASTs
- QVector<Token> _progText
+ LRParser(Grammar g, QVector<Token> t)
+ void Generate()
+ void Parse()
+ bool GetResult()
+ QVector<ElementLR1> Read(QVector<ElementLR1>, Token)
+ QVector<ElementLR1> Closure(ElementLR1)
+ ParserTable GetTable()
+ QVector<Token> PossibleTokens(int state)
+ QVector<Token> First(QVector<Token>)
+ QVector<RuleNo> GetRulesUsed()
+ void SetProgText(QVector<Token>)
+ void SetProgramAST(Node*)
+ QVector<Node *> GetValidASTs()
+ Node* BuildCST()
+ Node* GetMostRightenNodewNonTerm(Node*, Token)
+ QVector<Node*> GetPatternForest()
+ void printCST(Node*)
+ void CalValidMatches()
+ bool ProcessCSTs(Node*,Node*)
+ QVector<Token> GetText()
+ QVector<Token> GetProgText()
+ void ResetProgCST(Node*)

Figure 13: LRParser class diagram[15]

The LRParser needs the representation of the input grammar. The input grammar is represented in the Grammar class that is a private data member of the LRParser. The Grammar class contains all the information about the input grammar including the rules the terminals, the nonterminals, the starter token, and even the metavariable token and the end of text token too (figure 14). To fill up the instance of the Grammar class with the input, the class has functions that provides the functionality to do that. These functions are the AddRule(Rule), SetMetaToken(Token), SetTerminals(QVector<Token>), SetNonTerminals(QVector<Token>). There is also a function to insert unique Starter and EndOfText tokens, and a function that inserts a rule with the new Starter token on the left of the rule and the original start symbol given by the user on the right. To properly create the Grammar class, we need two more classes

[15] Source: made by own

that are indispensable for representing a grammar. These two are the Rule and The Token classes.

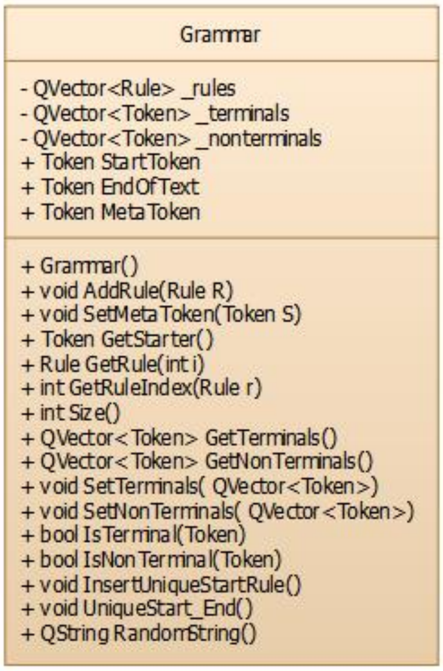

Grammar
- QVector<Rule> _rules
- QVector<Token> _terminals
- QVector<Token> _nonterminals
+ Token StartToken
+ Token EndOfText
+ Token MetaToken
+ Grammar()
+ void AddRule(Rule R)
+ void SetMetaToken(Token S)
+ Token GetStarter()
+ Rule GetRule(int i)
+ int GetRuleIndex(Rule r)
+ int Size()
+ QVector<Token> GetTerminals()
+ QVector<Token> GetNonTerminals()
+ void SetTerminals(QVector<Token>)
+ void SetNonTerminals(QVector<Token>)
+ bool IsTerminal(Token)
+ bool IsNonTerminal(Token)
+ void InsertUniqueStartRule()
+ void UniqueStart_End()
+ QString RandomString()

Figure 14: Grammar class diagram[16]

The Rule class is the representation of the rules of the grammar (figure 15). Therefore, it contains a left side and a right side. It also must implement the equal operator for proper comparisons. The left side of a Rule is a Token, and the right side is a vector of Tokens.

The Token class is one of the most low-level classes (figure 16). The tokens are the representations of the terminals and nonterminals of the grammar. These can consist more than one character especially in the grammars if programming languages where the terminals are the keywords and variables etc. So, the Token class consist of a sequence of a string that is basically a sequence of characters, and it must implement the comparison between two tokens.

[16] Source: made by own

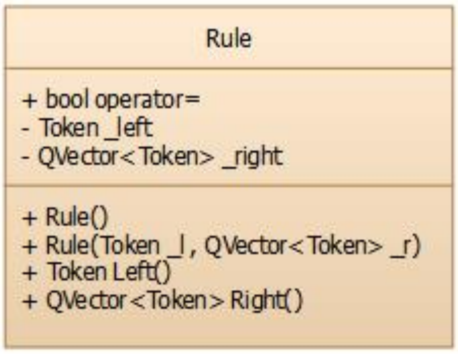


Figure 15: Rule class diagram[17]

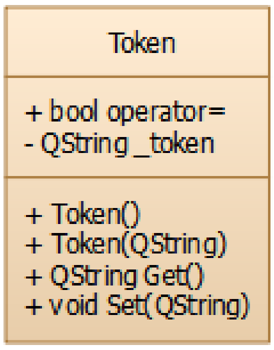
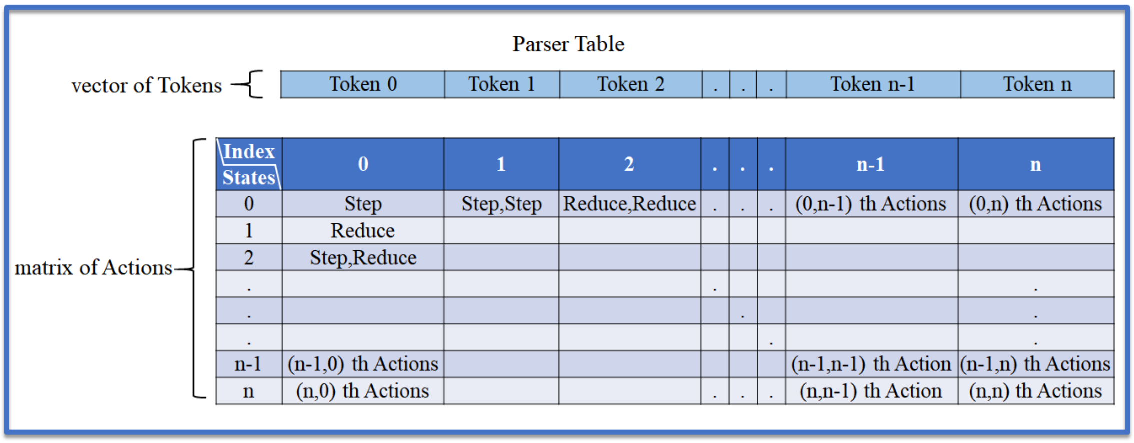


Figure 16: Token class diagram[18]

The next building component of the program is the parser table. The LRParsers Generate() function generates the content of the parser table and it is used by the Parse() function to parse the input program and pattern. Therefore, the program does not generate the program code of a parser that than must be compiled to use, it fills in the contents of the parser table dinamically. This decision was made because this way we get a simpler library. The parser table is represented by the ParserTable class (figure 17). The LRParser class contains an instance of the ParserTable as a private data member. The parser table is represented as a vector of tokens and a matrix of Actions. The vector of Tokens must contain every terminal and nonterminal and even the end of text token. The order of the tokens is important and should not be reorganised. Only the LRParsers Generate() function should deal with this vector and it should be in sync with the matrix of Actions which is also the Generate() functions responsibility. It is important because the positions of the tokens in the vector determinates the corresponding column in the matrix of Actions that what are the executable Actions in the appropriate states through the parsing. In this representation the states are the indexes of the matrix rows. The following figure shows the representation of the Parser Table (figure 18).

[17] Source: made by own
[18] Source: made by own

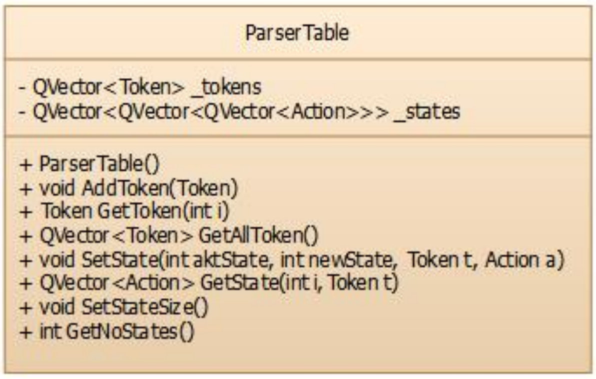

Parser Table
vector of Tokens
matrix of Actions

Token 0	Token 1	Token 2	.	.	.	Token n-1	Token n

Index / States	0	1	2	.	.	.	n-1	n
0	Step	Step,Step	Reduce,Reduce	.	.	.	(0,n-1) th Actions	(0,n) th Actions
1	Reduce							
2	Step,Reduce							
.				.				.
.					.			.
.						.		.
n-1	(n-1,0) th Actions						(n-1,n-1) th Action	(n-1,n) th Actions
n	(n,0) th Actions			.	.	.	(n,n-1) th Action	(n,n) th Actions

Figure 18: Representation of the ParserTable[19]

The actions in the matrix are Action structs. An Action struct contains an indicator enmun that can be: Ok, Step, Reduce, Error. Based on the indicator the Action contains more information about the corresponding action. If the indicator is Step, it contains the state where the parser must step, if the indicator is Reduce it contains an ElementLR1 that will be explained later, for now it is the rule that the reduction is based on. If the indicator is Ok or Error, there are no additional information in the struct because these actions do not require information other than the actual indicator.



```
ParserTable

- QVector<Token> _tokens
- QVector<QVector<QVector<Action>>> _states

+ ParserTable()
+ void AddToken(Token)
+ Token GetToken(int i)
+ QVector<Token> GetAllToken()
+ void SetState(int aktState, int newState, Token t, Action a)
+ QVector<Action> GetState(int i, Token t)
+ void SetStateSize()
+ int GetNoStates()
```

Figure 17: ParserTable class diagram[20]

The ElementLR1 mentioned above is a class (figure 19). This class implements the LR1 element that has an outstanding role in the parser generation process. This class contains a rule, a lookahead Token and an index called dot. The index is for keeping track of the already used tokens of the right side of a rule. This class also implements comparison between ElementLR1s. LR1 elements are used to create the canonical sets

[19] Source: made by own
[20] Source: made by own

of a grammar, and from the canonical sets the parser table can be created. The reason for using LR1 elements through the generation of the parser table is that the GLR parser that is implemented in the software is based on the LR1 parsing. This can be traced back to the fact that the GLR parser is a special version of the LR parser.

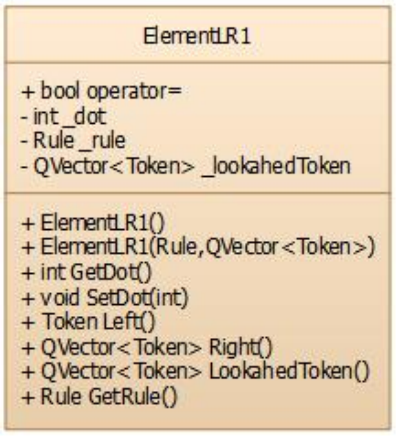


Figure 19: ElementLR1 class diagram[21]

After the generation is complete and the parser table is filled, we can move on to the actual parsing phrase, where the software parses the input program and the input pattern. For the easier handling I created a basic lexical parser that converts the raw text-based input program and pattern into sequences of tokens. With this feature the Users does not need to provide their code and pattern in a tokenised form.

The Parser() function found in the LRParser is responsible to parse the input program and the input pattern. This is a GLR parser as stated in the publication [1], however the publication supplements an already existing GLR parser, on the other hand I created my own GLR parser. This should handle the parsing of the input program without any issue. On the other hand, the pattern is different, because it can contain metavariables at any point of the code. So, the parser must be prepared for such situations. Based on the solution of the publication [1], in these cases the parser must fork the parsing into every possible substitution of the metavariable. This is almost the same mechanism that the GLR is using when it comes across a conflict in the parser table. These conflicts occur when the parser table contains more than one Action in the actual Token State pair. These conflicts are resolved with a forking mechanism that forks the parser into all the Action at the current conflict.

[21] Source: made by own

The main component of the Parser() is the Stack that is a main element of a parser, it can contain StackElements. StackElement is a struct that hold informations about a Token, State pair that describes an actual point of the parsing. Other elements of the Parser() are the Text that is the actual sequence of Tokens that are parsed, the index that shows the actual token read by the parser from the Text. There is another stack called IsMetaNonTerminal that contains BoolTokens and it is used to determine if the tokens in the Stack are metavariables or not The BoolToken is a struct that contains a bool flag that tells if a token is a metavariable, and if so the BoolToken also contains the name of the metavariable. The order of the rules that were used in reduction through the parsing are also stored in a vector that contains RuleNos. RuleNo is a struct, that contains the index of a rule, a bool that tells if the rules right side were containing metavariables, and if so, this also saves the locations of the metavariables position in the rules right side. The parser has an element that tells if the last action was a reduction and if so, the reductions result token will be stored in the RedToken variable.

The next decision was about the branches created by the forking mechanism. The decision was to deal with the branches in synchronous manner. To achieve this, I created a struct called ForkElement. The Forkelement contains all the necessary information to continue a parse from any given consistent point. This struct contains the next Token , the text index used to keep track of the actually parsed input, the actual state, the indexes of the used rules in this parse, the index of the Action that should be used in this branch if the fork was caused by a conflict, another stack that is used to determine if a given token is substitution of a metavariable, a bool that tells if the last Action was a reduction, and a reduced token that contains the token that the last reduction resulted.

These forking mechanisms are the cause that at the end there are more than one possible result. While the forking mechanisms can fork into dead ends these branches are sorted out when the parsing reaches the point where it cannot continue the parsing. This does not mean that there cannot be more than one possible result. This can happen because of the different constitutions of the metavariables and the ambiguous grammars.

The result of the parsing is the corresponding concrete syntax tree of the parsed input. Because of the forking mechanisms there can me more than one CST for one input. That means the result of the parsing is a forest of CSTs.

There is also another mechanism in the Parse() function that causes a similar effect. This mechanism is the solution of the starting point problem. This mechanism crates branches by the good starting state candidates before the actual parse starts. This means that the branches will be different in only the staring state

The final part of the Parse is the CST creation. When a branch has been parsed, and it was a successfully parsed, the corresponding CST is created by the BuildCST() function. The CSTs of the input program and the input patterns are stored separately. The CSTs are based on the Node class. Therefor the CSTs representations are trees of Nodes. This function creates the CST based on the sequence of the rules used through the parsing of the branch. This sequence is processed from the last rule and continues backward. This way the left Token of the last rule becomes the root Node of the CST, and the Tokens of the right side becomes the children. Then the left Token of the next rule is searched in the tree, the rightmost occurrence is needed that is not a metavariable to build the correct CST. When this mechanism comes across a rule that has a metavariable flagged Token in its right side, the child is marked as metavariable. The metavariable node also holds the name of the metavariable. A metavariable node cannot have any children, so the tree will contain a hole where the metavariable node is located. These holes will be filled with the actual substitution of the metavariables when a program CST is matched on a pattern CST. The data that fills the hole becomes the substitution of the metavariable in the result of the software.

The Node class stores information about the Token that it stands for, a flag that tells if this Token is a metavariable or not, and the pointers to its child nodes (figure 20). This class implements the comparison between Nodes and also contains getters and setters for its data.

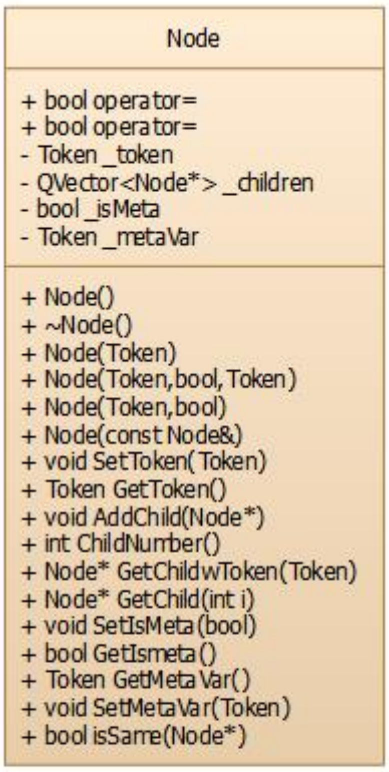


Figure 20: Node class diagram[22]

By the end of the Parse() function the parser created two forests of CSTs, one for the input pattern and one for the input program. If we compare the pieces of these two forests, we can get the actual result of the software, the matched CSTs. This comparison is discussed in the publication as Unification [1]. However, my implementation differs at some point from the method discussed there. First my program works with CST and in my implementation the program also has a forest of CST not just the pattern, but the main difference is that the algorithm in my implementation is not compares the pattern CSTs root to all the nodes of the programs CST. The reason behind this, is that the demand is to be able to select a code and then apply a refactoring pattern on it. The consequence of this is that a program will only match to a pattern if their beginning is the same. This means that their beginning is either starts from the same state or the pattern starts with a metavariable.

This mechanism is implemented in the CalValidMatches() function that can be found in the LRParser class. This function should be called after the Parse() function to work properly. The CalValidMatches() function compares all the CSTs of the program with all the CSTs of the pattern. The actual comparison between two CSTs is done by

[22] Source: made by own

the ProcessCSTs() function that is a helper function for the CalValidMatches() function. The comparison starts at the root node, if the tokens are the same in the pattern node and in the program node, the function compares the children. This continues in a recursive manner. When this comparison finds a metavariable Node in the pattern, this node will get the copy of the programs corresponding nodes children as its own children. This is the actual substitution of the metavariable, that is a subtree whose root is the metavariable Node and it were copied from the program CSTs corresponding subtree. When the comparison is complete, and it was a success the matched CST is added to the set of the valid matches. If two different matches result the same CST than only one will be present in the result set.

The structure of the whole model can be seen in the following figure (figure 21).

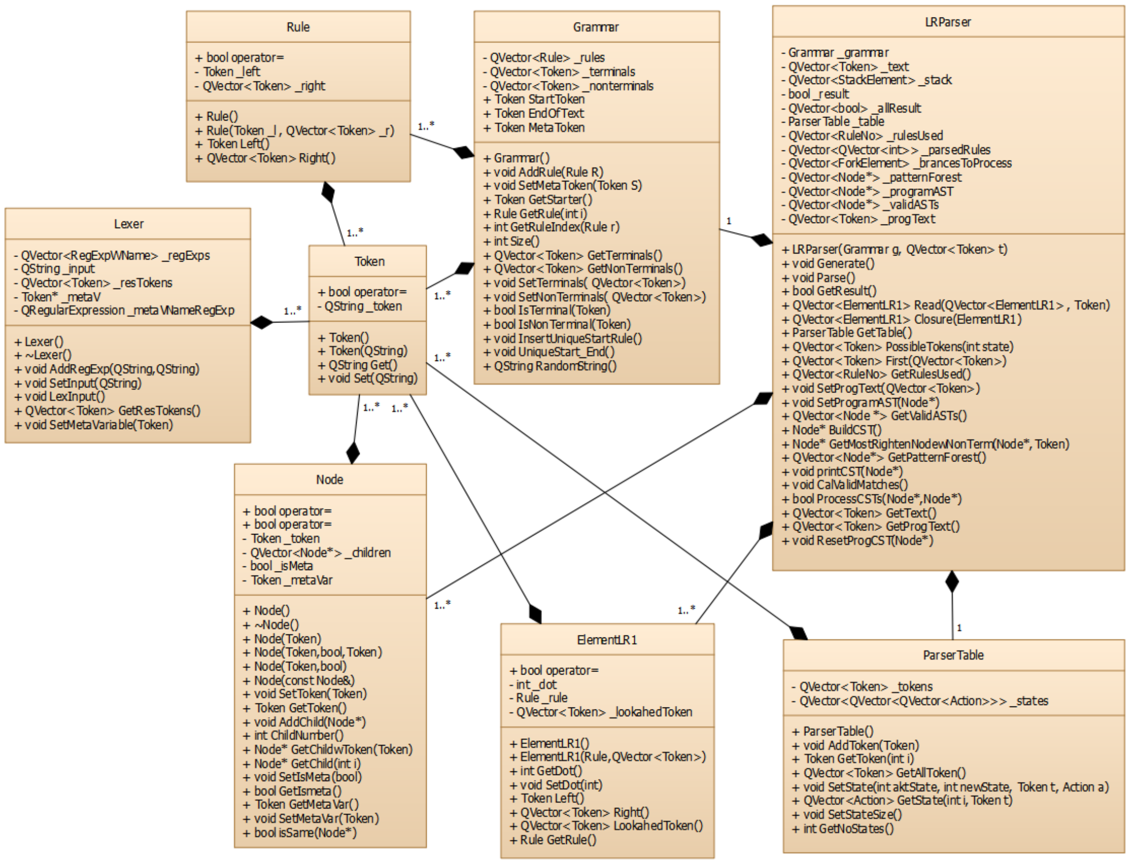

Rule
+ bool operator=
- Token _left
- QVector<Token> _right
+ Rule()
+ Rule(Token _l, QVector<Token> _r)
+ Token Left()
+ QVector<Token> Right()
Grammar
- QVector<Rule> _rules
- QVector<Token> _terminals
- QVector<Token> _nonterminals
+ Token StartToken
+ Token EndOfText
+ Token MetaToken
+ Grammar()
+ void AddRule(Rule R)
+ void SetMetaToken(Token S)
+ Token GetStarter()
+ Rule GetRule(int i)
+ int GetRuleIndex(Rule r)
+ int Size()
+ QVector<Token> GetTerminals()
+ QVector<Token> GetNonTerminals()
+ void SetTerminals(QVector<Token>)
+ void SetNonTerminals(QVector<Token>)
+ bool IsTerminal(Token)
+ bool IsNonTerminal(Token)
+ void InsertUniqueStartRule()
+ void UniqueStart_End()
+ QString RandomString()
LRParser
- Grammar _grammar
- QVector<Token> _text
- QVector<StackElement> _stack
- bool _result
- QVector<bool> _allResult
- ParserTable _table
- QVector<RuleNo> _rulesUsed
- QVector<QVector<int>> _parsedRules
- QVector<ForkElement> _brancesToProcess
- QVector<Node*> _patternForest
- QVector<Node*> _programAST
- QVector<Node*> _validASTs
- QVector<Token> _progText
+ LRParser(Grammar g, QVector<Token> t)
+ void Generate()
+ void Parse()
+ bool GetResult()
+ QVector<ElementLR1> Read(QVector<ElementLR1>, Token)
+ QVector<ElementLR1> Closure(ElementLR1)
+ ParserTable GetTable()
+ QVector<Token> PossibleTokens(int state)
+ QVector<Token> First(QVector<Token>)
+ QVector<RuleNo> GetRulesUsed()
+ void SetProgText(QVector<Token>)
+ void SetProgramAST(Node*)
+ QVector<Node *> GetValidASTs()
+ Node* BuildCST()
+ Node* GetMostRightenNodewNonTerm(Node*, Token)
+ QVector<Node*> GetPatternForest()
+ void printCST(Node*)
+ void CalValidMatches()
+ bool ProcessCSTs(Node*,Node*)
+ QVector<Token> GetText()
+ QVector<Token> GetProgText()
+ void ResetProgCST(Node*)
Lexer
- QVector<RegExpWName> _regExps
- QString _input
- QVector<Token> _resTokens
- Token* _metaV
- QRegularExpression _metaVNameRegExp
+ Lexer()
+ ~Lexer()
+ void AddRegExp(QString,QString)
+ void SetInput(QString)
+ void LexInput()
+ QVector<Token> GetResTokens()
+ void SetMetaVariable(Token)
Token
+ bool operator=
- QString _token
+ Token()
+ Token(QString)
+ QString Get()
+ void Set(QString)
Node
+ bool operator=
+ bool operator=
- Token _token
- QVector<Node*> _children
- bool _isMeta
- Token _metaVar
+ Node()
+ ~Node()
+ Node(Token)
+ Node(Token,bool,Token)
+ Node(Token,bool)
+ Node(const Node&)
+ void SetToken(Token)
+ Token GetToken()
+ void AddChild(Node*)
+ int ChildNumber()
+ Node* GetChildwToken(Token)
+ Node* GetChild(int i)
+ void SetIsMeta(bool)
+ bool GetIsmeta()
+ Token GetMetaVar()
+ void SetMetaVar(Token)
+ bool isSame(Node*)
ElementLR1
+ bool operator=
- int _dot
- Rule _rule
- QVector<Token> _lookahedToken
+ ElementLR1()
+ ElementLR1(Rule,QVector<Token>)
+ int GetDot()
+ void SetDot(int)
+ Token Left()
+ QVector<Token> Right()
+ QVector<Token> LookahedToken()
+ Rule GetRule()
ParserTable
- QVector<Token> _tokens
- QVector<QVector<QVector<Action>>> _states
+ ParserTable()
+ void AddToken(Token)
+ Token GetToken(int i)
+ QVector<Token> GetAllToken()
+ void SetState(int aktState, int newState, Token t, Action a)
+ QVector<Action> GetState(int i, Token t)
+ void SetStateSize()
+ int GetNoStates()
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1
1

Figure 21: Structure of the model[23]

[23] Source: made by own

3.4.2 View

The main goal of my thesis was to create an implementation of the reflexive and automated pattern matching based on the publication. This does not include any front-end user interface. The user interface implemented in this project is created for purely demonstration purposes. The UI was created in the Qt framework. It has three main components. The Main window, the Result window, and the AppManager. When the program is started the main window appears. Here can be found seven fields. These fields provide the possibility to add the inputs of the program. Here can be added the grammar, the terminals and nonterminals, the regular expressions, the input program and pattern, and also the metavariable token. At the bottom right corner of the window, a button can be found called "Match". This button starts the matching process, but only if all the fields are filled correctly. If any of the fields are left empty or does not match to the specified format described in the user documentation the Matching process will not start, and an error message will appear about the location of the misplaced input.

After the correct inputs were given, and the match is finished the result window will appear. The result window contains the result of the match. If no matches were found the indicator at the top will become red and will show the text: Failed. Also, the number of matches will become 0. However, if there were one or more matches, the in indicator at the top will become green and will show the text: Successful. Also, the number of matches will be shown. The result CSTs can be viewed, one at a time. A CSTs is shown in the middle of the window. The CST is represented in a Tree model, that is implemented as a QTreeWidget. The root of the CST will appear in the tree model widgets top left corner, and its children next to it and one line lower as well. For the easier transparency, the nodes with children can be collapsed or expanded. If the match contains a metavariable, its name and its node is highlighted. The subtree that has a metavariable in its root is the concrete substitution of the metavariable in this representation.

The result window gets the result CSTs and converts their nodes into QTreeWidgetItems that can be added to the QTreeWidget for display. Under the tree model widget there are three mentionable parts of the result window. In the middle there is the index of the current CST displayed in the tree model widget. To its left and right there are two buttons. To the left the Previous button takes its place. It is used to switch the displayed result CST to the previous one. To the right the Next button can be found.

Similarly, it is used to switch the displayed result CST to the next. These buttons can only be used if there is next or previous CSTs compared to the actually displayed CST. The result window can be closed with the “X” symbol in the top right corner. After the result window closed a new match can be started, and a new result window will appear. The main window can be closed similarly with the “X” symbol in the top right corner.

The view is entirely separated from the model. These two communicate with each other through the AppManager class. The view provides the inputs, the app manager converts it into a processable formant than passes on to the model. The model processes the received inputs and manufactures the results that are sent to the result window where the results are displayed. The result window convers the received result CSTs nodes into displayable tree model items if the result contains matches. The result window is generated in a code first manner. However, the main window was generated with the Qt designer.

The main window is implanted in the MainWindow class, and the result window is implemented in the ResWindow Class. The ResWindow class is inherited from the QDialog class (figure 22). It has two main functions, the SetResTrees(QVector<Node*>) and the BuildTree(Node*). The SetResTrees(QVector<Node*>) function is responsible for setting the result trees value, and based on that is sets the result windows appropriate fields too. If the result trees vector that the function gets is empty it is interpreted as the match was failed and the window will show just that. The indicator becomes Fail and the background red. However, if the result trees is not empty it contains the valid matched trees, so the functions sets the result to Success, sets the number of valid matches to the size of the vector, and displays the first tree from the vector. To display the tree, the BuildTree(Node*) function is called. The Next and Previous buttons are enabled or disabled based on the number of the valid matches. The BuildTree(Node*) function is responsible to create the displayable tree, because the TreeWiget that displays the tree needs a special type of tree made from QTreeWidgetItems. It creates QTreeWidgetItems from the nodes and their children recursively (figure 23).If a nodes children in an epsilon that the child and its parent are hidden form the result tree for the clearer transparency.

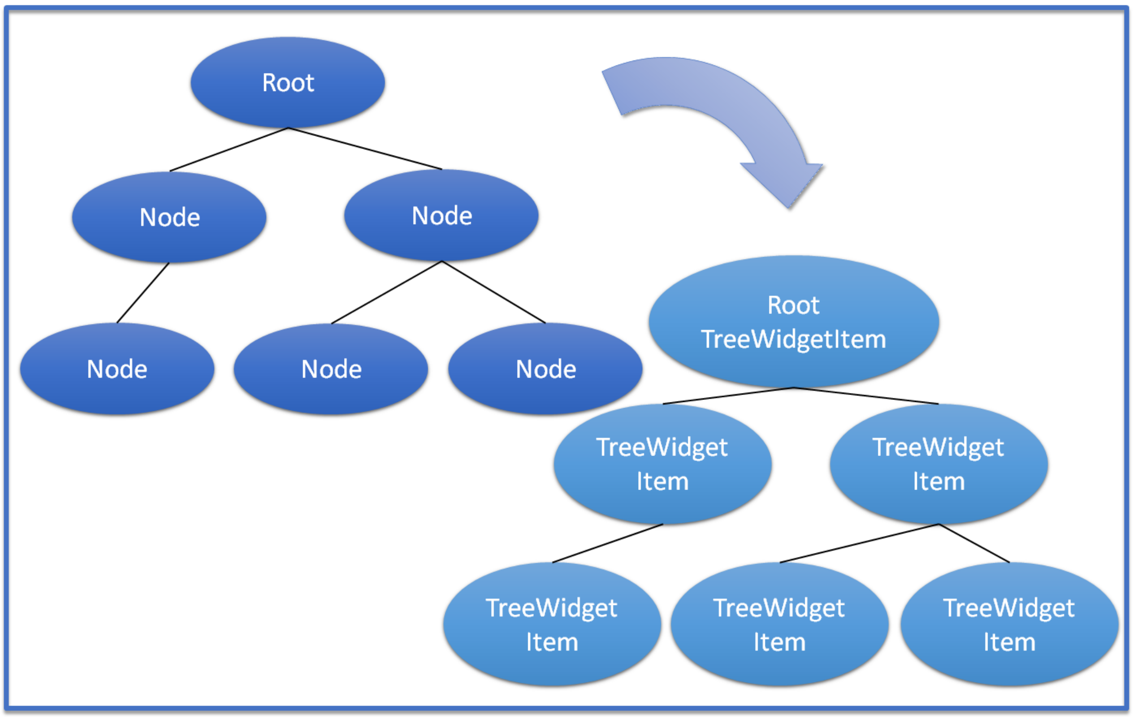
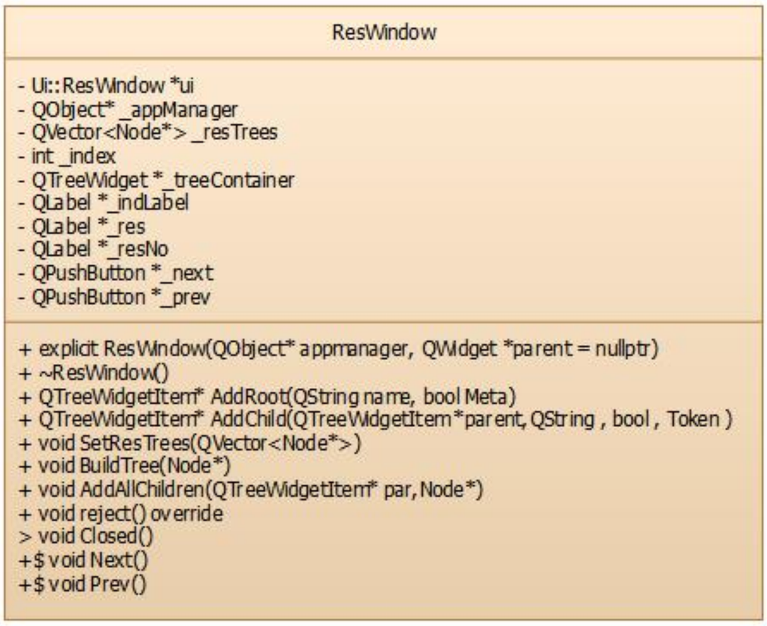


Figure 23: Result tree conversion to displayable TreeWidgetItem tree[24]

For this mechanism the class implements two helper functions, the AddRoot(QString name, bool Meta) and the AddChild(QTreeWidgetItem *parent,QString , bool , Token). Both of these functions return a pointer to a QTreeWidgetItem. The AddRoot function is responsible to create the root node of the displayable tree. While the AddChild function is responsible for creating and adding a child node to a parent node. For smoother working the AddAllChildren(QTreeWidgetItem* par,Node*) helper function adds recursively all its children to a parent using the AddChild functions mentioned before. The ResWindow class also implements two public slots, the Next() and the Prev(). These two slots are connected to their corresponding buttons, the Next buttons and the Previous buttons clicked() signal. The Next slot is responsible to display the next result CST, and if the last result CST is reaches, it disables the Next button, also if the displayed CST is not the first it enables the Previous button too. The Prev() slot is responsible to display the previous CST, it disables the Previous button if the first CST is reached and it also enables the Next button if the displayed CST is not the last one. At the bottom of the result window the number of the actually displayed CST is found. This also changes on the effect of these two slots.

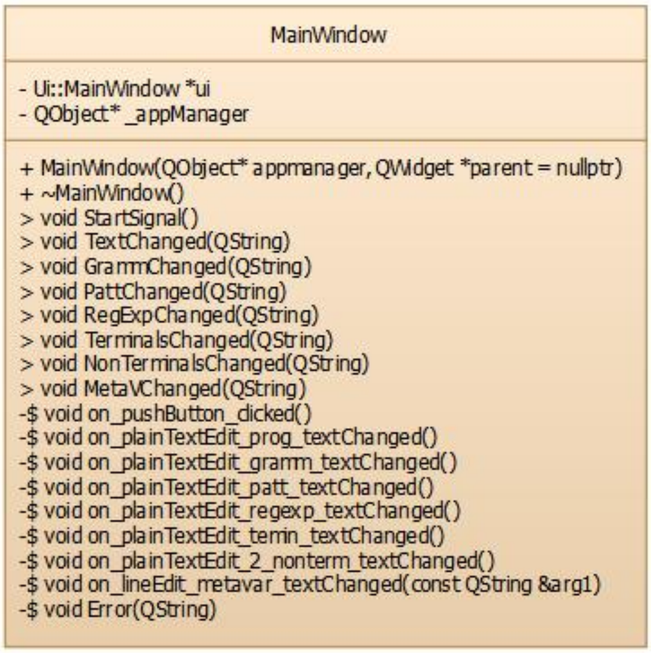
[24] Source: made by own

The Reswindow class also implements a signal that is called: Closed(). This signal is fired when the result window is closed. To achieve this the reject() function has been overridden, to emit this signal. This signal is connected to the ResClosed() slot of the AppManaggers.


```
ResWindow

- Ui::ResWindow *ui
- QObject* _appManager
- QVector<Node*> _resTrees
- int _index
- QTreeWidget *_treeContainer
- QLabel *_indLabel
- QLabel *_res
- QLabel *_resNo
- QPushButton *_next
- QPushButton *_prev

+ explicit ResWindow(QObject* appmanager, QWidget *parent = nullptr)
+ ~ResWindow()
+ QTreeWidgetItem* AddRoot(QString name, bool Meta)
+ QTreeWidgetItem* AddChild(QTreeWidgetItem *parent, QString , bool , Token )
+ void SetResTrees(QVector<Node*>)
+ void BuildTree(Node*)
+ void AddAllChildren(QTreeWidgetItem* par, Node*)
+ void reject() override
> void Closed()
+$ void Next()
+$ void Prev()
```
Figure 22: ResWindow class diagram[25]

The MainWindow class is inherited from the QMainWindow class (figure 24). The slots of this class are handling the inputs from the plain text editors that are the fields that is filled by the user with the inputs. There is also an Error(QString) slot that handles the signals that are coming from the AppManager while processing the raw inputs. The same can be said about the signals of the class, although these are connected to the AppManagers appropriate slots. These signals are emitted from the slots mentioned above, and their purpose is to notify the AppManager about the change of the inputs. There is one exception signal, the StartSignal(), it is emited in the on_pushButton_clicked slot that I sconnected to the Match buttons clicked() signal. The StartSignal() is connectet to the AppManagers Start() slot. However, the on_pushButton_clicked slot contains checks. These check are watching if any of the inputs are empty and is so the StartSignal() is not fired, but a QMessageBox is created which contains the error information.

[25] Source: made by own



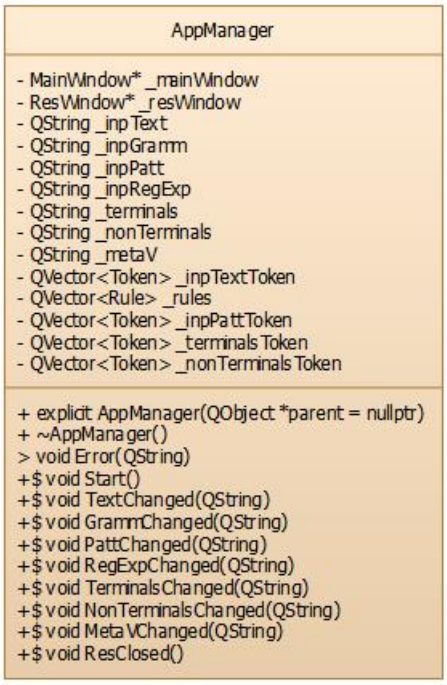
```
MainWindow

- Ui::MainWindow *ui
- QObject* _appManager

+ MainWindow(QObject* appmanager, QWidget *parent = nullptr)
+ ~MainWindow()
> void StartSignal()
> void TextChanged(QString)
> void GrammChanged(QString)
> void PattChanged(QString)
> void RegExpChanged(QString)
> void TerminalsChanged(QString)
> void NonTerminalsChanged(QString)
> void MetaVChanged(QString)
-$ void on_pushButton_clicked()
-$ void on_plainTextEdit_prog_textChanged()
-$ void on_plainTextEdit_gramm_textChanged()
-$ void on_plainTextEdit_patt_textChanged()
-$ void on_plainTextEdit_regexp_textChanged()
-$ void on_plainTextEdit_termin_textChanged()
-$ void on_plainTextEdit_2_nonterm_textChanged()
-$ void on_lineEdit_metavar_textChanged(const QString &arg1)
-$ void Error(QString)
```

Figure 24: MainWindow class diagram[26]

The third element of the View is the AppManager class (figure 25). This class is responsible for the communication between the components of the software. This class stores the raw inputs and converts them into a processable format for the model. This class is inherited from the QObject class. Most of its slots are connected to the MainWindows signals that are notifying the AppManager about the change of the raw input. However, there are two exceptions, the ResClosed() slot the is connected to the result windows Closed() signal, and the Start() slot that is connected to the main windows StartSignal() signal. The Start() slot is responsible for staring the model. First it converts the inputs. It tokenises the raw Terminal and nonterminal inputs and creates Rules from the raw input grammar. From these data it created the Grammar, it also tokenises the metavariable specifier and add it to the Grammar.It fills up the Lexer with the regular expressions. Then it uses the Lexer to create the sequences of tokens from the raw input program and input pattern. If any of these procedures encounter a problem, an Error() signal is fired to the main window and an error message appears. If there were no problem, an instance of the LRPraser is created. The AppManagger fills up the LRParser with the processed inputs, and the matching starts. When the matching finished, an instance of the ResWindow is created and the results are given to it. When the result window appears, the main window is disabled. The main window gets enabled

[26] Source: made by own

again when the result window is closed. The ResClosed() slot is responsible for enabling the main window after the result window is closed.



```
AppManager

- MainWindow* _mainWindow
- ResWindow* _resWindow
- QString _inpText
- QString _inpGramm
- QString _inpPatt
- QString _inpRegExp
- QString _terminals
- QString _nonTerminals
- QString _metaV
- QVector<Token> _inpTextToken
- QVector<Rule> _rules
- QVector<Token> _inpPattToken
- QVector<Token> _terminalsToken
- QVector<Token> _nonTerminalsToken

+ explicit AppManager(QObject *parent = nullptr)
+ ~AppManager()
> void Error(QString)
+$ void Start()
+$ void TextChanged(QString)
+$ void GrammChanged(QString)
+$ void PattChanged(QString)
+$ void RegExpChanged(QString)
+$ void TerminalsChanged(QString)
+$ void NonTerminalsChanged(QString)
+$ void MetaVChanged(QString)
+$ void ResClosed()
```

Figure 25: AppManager class diagram[27]

3.4.3 Lexer

The easiest way to the user to provide the input program and the input pattern is in a text format. Normally most of the programs are written in a text-based formant. However, the parser can only process them as a sequence of tokens. This conversion is solved by the lexical parser. To convert the input program and input pattern into a processable sequence of tokens I implemented my own basic lexical parser. The lexical parser needs specific rules that describes the form of the possible tokens. These rules are called regular expressions. This is the reason that the user must define the regular expressions of the grammar. The lexical parser processes the input program and the input pattern by characters and converts the longest matching character sequence to a regular expression. This way the software produces the token sequences which meanings are equal to the input program and pattern[4].

The lexical parser was created with the help of the Qt frameworks build in QRegularExpression class. This class provides us a reliable matching tool that can

[27] Source: made by own

match a regular expression onto a string. The QRegularExpression class implements Perl-compatible regular expressions. For more information about the QRegularExpression class visit the Qt official documentation[28][6].

The lexical parser is implemented in the Lexer class (figure 26). After the creation of the Lexer, it needs regular expressions to operate, these regular expressions can be added by the Lexers AddRegExp method that requires two QString arguments, the first is the actual regular expression, the second is the keyword or the terminal symbol that the expression explains. To use the Lexer the text that we want to parse lexically must be set. This can be achieved by the Lexers SetInput method that requires one QString argument that is the actual input that we want to parse. To begin the lexical parsing, to create the token sequence the LexInput method can be called. The result is produced into the _resTokens private variable, that can be reached through the GetResTokens methold.

The Lexer implements a special feature that is the ability to parse lexically metavariable containing inputs. This comes handy while lexically parsing the input pattern. When the Lexer matches a metavariable token it switches into a metavariable mode. This mode will match the following characters of the input on a general regular expression that is usually used to describe variables in programming languages. This regular expression is: "[_a-zA-Z][_a-zA-Z0-9]*", and it is hardcoded in the Lexer class as a private constant called _metaVNameRegExp, for easy accessibility and modification purposes. This regular expression results a unique token that holds the name of the metavariable that will be processed by the GLR parser. When the next set of characters matched on this regular expression the lexical parser switches back to normal mode and continues the parsing. The regular expression that describes the metavariable token is automatically added to the Lexer. The process always watches longest fitting regular expression. The error handling is implemented as the illegal characters are considered as white spaces. This way if the input is "apple?tree" and the "?" is an illegal character, the result will be two tokens: "apple", "tree"[4].

[28]https://doc.qt.io/qt-5/qregularexpression.html#details

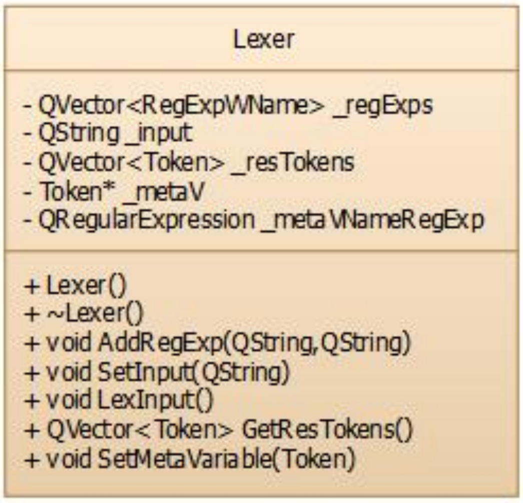
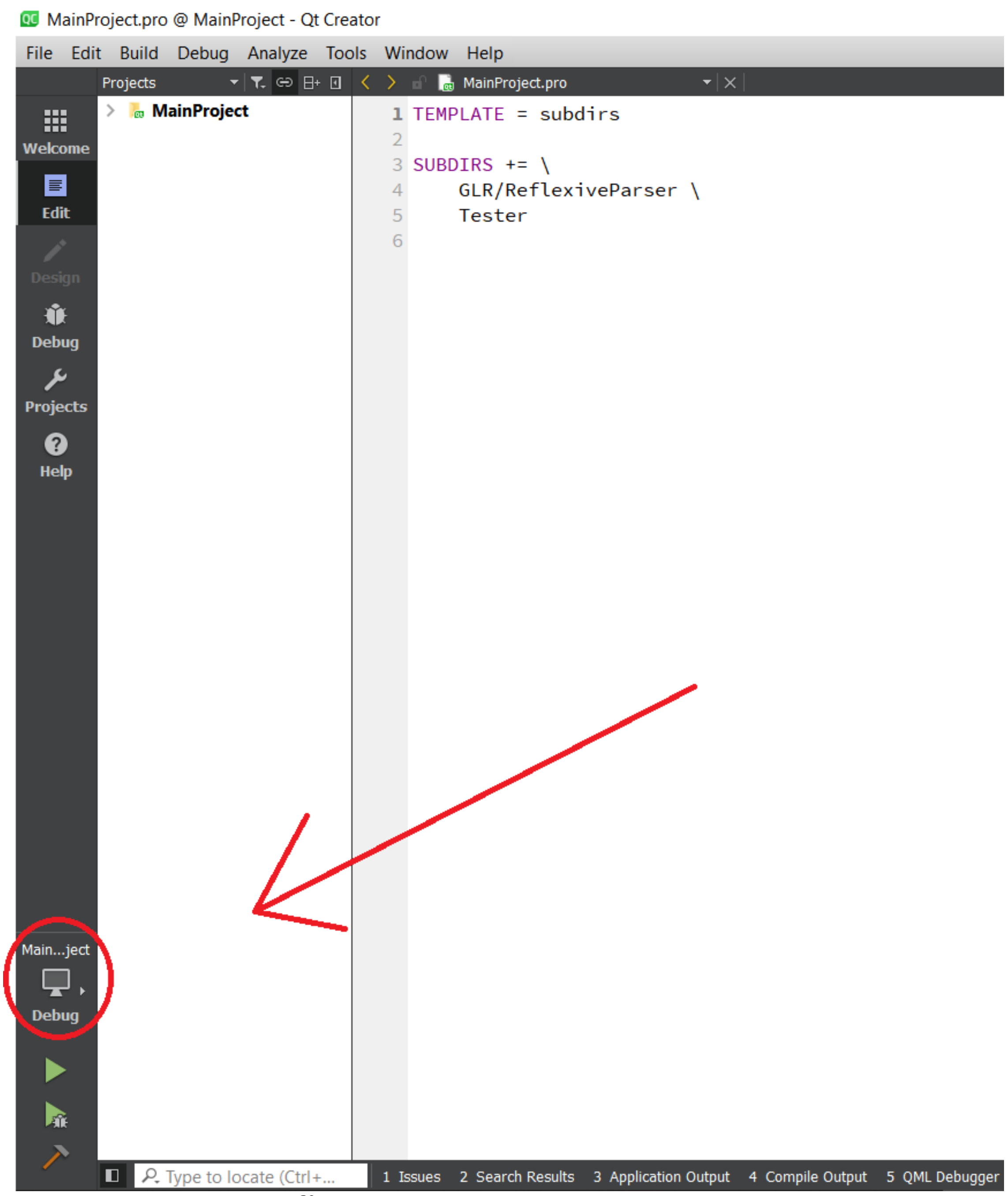

Lexer
- QVector<RegExpWName> _regExps
- QString _input
- QVector<Token> _resTokens
- Token* _metaV
- QRegularExpression _metaVNameRegExp
+ Lexer()
+ ~Lexer()
+ void AddRegExp(QString,QString)
+ void SetInput(QString)
+ void LexInput()
+ QVector<Token> GetResTokens()
+ void SetMetaVariable(Token)

Figure 26: Lexer class diagram[29]

3.4.4 Tests

To test the correct functioning of the model I created a testing module. The different tests for the different functions are implemented in the Tester class. The tester class is in a subproject of the software. This subproject was created especially for the testing mechanism. The project consists the tst_tester.cpp, this file implements the tester. It has its own main function that is created from a built in Qt macro. To run the tests, open the MainProject in Qt creator. Then find the symbol of a monitor that is originally found on the bottom left of the Qt Creators window, just over the green triangle, that is the build and run button and click that(figure 27).

[29] Source: made by own

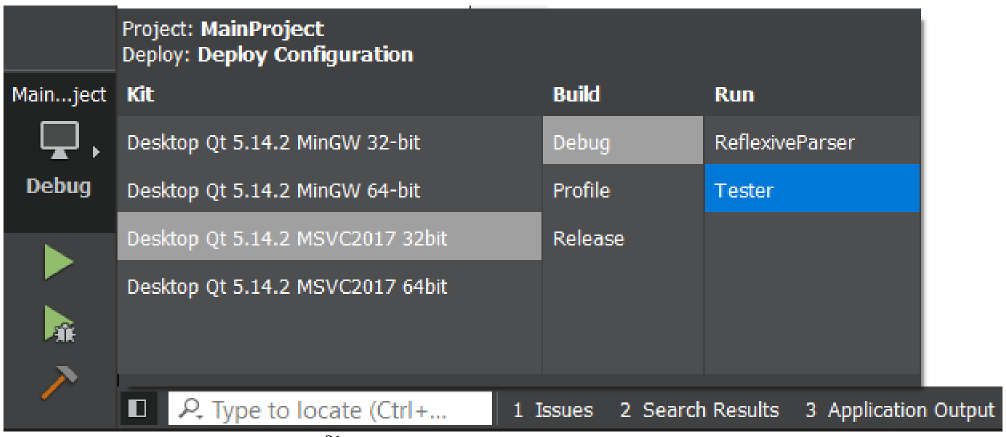


```
TEMPLATE = subdirs

SUBDIRS += \
    GLR/ReflexiveParser \
    Tester
```

Figure 27: Setting the tester 1.[30]

This will open a small Window where you can configure some options of the project. From the rightmost column called Run, select Tester (figure 28).

[30] Source: made by own


Project: **MainProject**
Deploy: **Deploy Configuration**

Kit	Build	Run
Desktop Qt 5.14.2 MinGW 32-bit	Debug	ReflexiveParser
Desktop Qt 5.14.2 MinGW 64-bit	Profile	Tester
Desktop Qt 5.14.2 MSVC2017 32bit	Release	
Desktop Qt 5.14.2 MSVC2017 64bit		

Figure 28: Setting the tester 2.[31]

Now the program will run the tester if you start the project with the Run button that is the green triangle at the bottom left. The result of the tester can be seen in the Application Output. To set back to the default program you must repeat the previous steps but at the end select ReflexiveParser.

The Tester implements various of tests. The first one is the Grammarfunctions() that tests the functionality of the Grammar class. Here the UniqueStart_End() and the InsertUniqueStartRule() functions are tested by specifying a grammar that uses the default Startoken and EndofTexttoken as terminals or nonterminals, this must cause the Grammar to create new unique tokens for these. It also tests if the number of rules are appropriate. And that the newly inserted rule has the new starter token on its left side.

The next test in the FirstSet() that is testing the First() function that helps to create the parser table. Here the program uses a basic grammar and its rules to specify the different rules first set.

The Closure_Read() test is examines the Closure() and the Read() functions that are also helping in the process of the parser table generation. This test uses a Grammar that needs at least an LR1 parser for parse its sentences. The tester uses the functions to create the appropriate sets that these functions create, then those sets are compared with the expected ones.

The TableGen() is tests the Generate() function. The input grammar has the following rules in this test: S->U, S->E, U->a, E->V = V, V->a. After the Generate() function is called, its data members are checked if they are containing the correct tokens. The correctness of the size of the parser table is also checked. Than all of the actions are checked that they match the excepted ones.

[31] Source: made by own

The Parser_LR1Grammar() test is checking if the parser can parse an LR1 grammar. The grammar is the following: Q->S, S->U, S->E, U->a, E->V = V, V->a. The terminals are: a, =. And the nonterminals are: Q, S, U, E, V. The parser parses the “a = a” input first. Then the “a” input is parsed also. Both inputs must be parsed without problem.

The following test is the Parser_fai(). This tests the Parse() function with an input that cannot be parsed by the parser. The input grammar is the following: Q->S, S->U, S->E, U->a, E->V = V, V->a. The terminals are: a, =. And the nonterminals are: Q, S, U, E, V. The parsed input is “a m”. The parsing of this input must fail because the generated parser table does not contain an “m” token because this symbol is part of the grammar.

The Parser_GrammarWithEpsilon() test is check if the Parser() function can parse a Grammar that has Epsilon rule. The grammar used here has the following rules: Q->S, S->Epsilon, S->(S)S. The “(“, “)” symbols are the terminals and the Q, S symbols are the nonterminals. The parsed text is the following: “((()))”. This test proves that the Generate() function can create a parser that can parse Epsilon containing grammars.

The next test is called Parser_GrammarGLR(). In this test such a grammar and input text is used that triggers the parsers forking mechanism more. This means that the parser will have more branches during the process of the parsing. The input text is the following: “1+1+1”, The rules of the grammar: Q->S, S-> S+S, S->1. This grammar has only two terminal symbols: “1” and “+”. Similarly, the two nonterminals are the “Q” and the “S”. At the end, the result of the parsing is checked.

The next test checks if the Parser() function can deal with metavariables. This test is called Parser_Input_With_Metavariable(). The grammar used in this test is the same as the one used in the Parser_GrammarGLR() test. However, now the input contains a metavariable. The input that is parsed here is the following: “1+@a”. In this input the “@” symbol is the metavariable specifier and the “a” is the name of the metavariable. The input must be parsed without problem, the result of the parse is checked at the end of the test.

The RulesUsed test is the following. This test checks if the sequence of the used rules through the parsing is matching the expected one. This is checked on a simple grammar and a simple input text.

CSTEquiv() it the test responsible to check the comparison between CSTs. Here four CSTs are created, that last three is compared to the first one. Also, the first and the second CSTs are the same (figure 29).

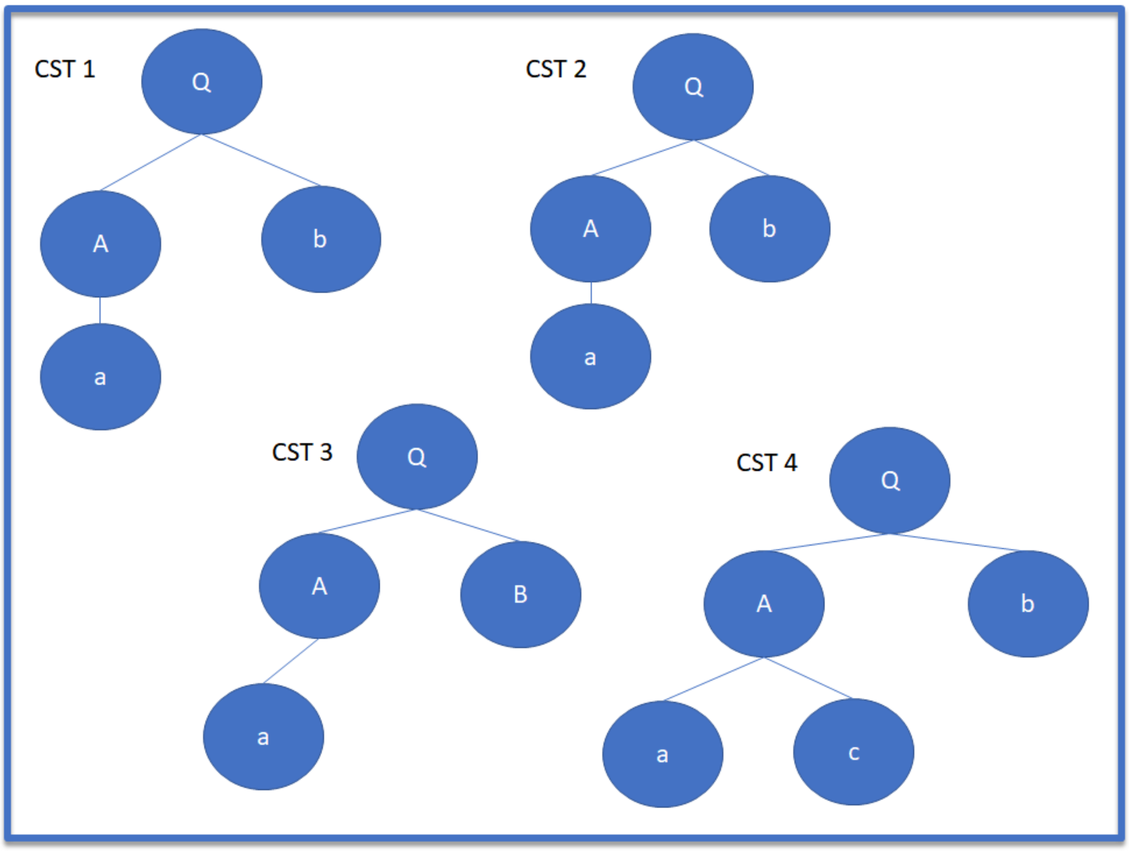


Figure 29: The CSTs used in the CSTEquiv() test[32]

The first check is if CST 1 and CST 2 are equal, then if CST 1 is not equal with CST3, then if CST1 is not equal to CST4.

The ValidMatches() test checks if the results of the matching is correct. This means the main feature tested here is the CalValidMatches() function because all the other function in this test were already tested on their own. This test checks if there are as many results created as expected. This tests uses the same grammar that were used in the Parser_Input_With_Metavariable() test however, now there are two input texts, on stands for the input program and one stands for the input pattern. First the two inputs are the same and the pattern does not contain metavariables. Then a second check is made, where the pattern contains a metavariable. This way we can see if the functions works correctly with and without metavariables.

[32] Source: made by own

In the StarterProblem() test the Parser() functions special part is checked. This special part is the solution for the starter point problem. The grammar used here is resembles a bit more for a “program” languages grammar. The rules of the grammar are the following: Q->S, S->Pkg Clss, Clss-> Class Name Classes, Classes-> Class Name Classes, Classes->Epsilon. The terminals are the following: Class, Pkg. And the nonterminals are these: Q, S, Clss, Classes. The interesting part here is the input text. It does not start with the Pkg key word that is always the first token of the sentences described by this grammar. However, with the solution of the starter point problem, subsentences are also parse able. This provides the opportunity to parse subprograms and subpatterns. The input that the parser must pars in this test is the following: “Class Name”. The result of the parse is checked at the end of the test.

The LexerTest() tests the functionality of the Lexer class. In this test several different inputs and regular expressions are added to the Lexer and after it parses them lexically the results are checked if they are the same as the expected.

After all the components of the model were tested an overall test comes. The TestRun() is the last test that uses all the components of the model in one test. This test checks if the components can work together. At the end of the test it is checked if there were valid matching CSTs. The input is the following:

- Grammar:

```
S->pkg name coma A
A->id equ EXP coma A
A->
EXP->num
EXP->EXP add EXP
EXP->id
```

- Terminals:

  pkg, name, coma, id, equ, num, add

- Nonterminals:

  S, A, EXP

- Regular expressions:

```
package pkg
add add
, coma
= equ
[A-Za-z]+ name
_[A-Za-z]+ id
[0-9]+ num
```

- Metavariable specifier: @

- Pattern:

```
package alma ,
_myres = 1 add @ metav ,
```

- Program:

```
package alma ,
_myres = 1 add 2 add 3 ,
```

The expected results are all the CSTs that contains all the different substitution of the metavariable. In this case the substitution of the @metav metavariable is the node that represents the EXP nonterminal, and its child nodes that are representing the num ,add ,num terminals.

3.5 Future work

The main development opportunities of the software are in the world of code transformations. As refactoring and code transformations are needed more and more. The demand for code transformation and code refactoring tools is growing. However, to use the existing refactoring tools the user must invest a huge amount of time into learning the internal structure of that given tool. The software implemented in my thesis can be a good building brick of a new automated refactoring tool. A refactoring tool that minimises the invested time and maximises the productivity.

The implemented model can also be reused in bigger refactoring projects that makes pattern writing easier. This way the overall refactoring process of a large code can be simplified. This implementation provides a good pattern matching engine that can even be reused as a library in other code transformation softwares.

Some improvements in the model can be made in the future. The asynchronous implementation of the forking mechanism in the Parse() function. Improve the Grammar so precedence and associativity can be added as addition information. Implementing the ability to handle different typed metavariables. The Lexer can also be improved as saving the important tokens actual name like the name of the variables and the actual numbers.

4. Conclusions

Overall, the development of the project went well. However, it had its ups and downs. I found especially hard to implement the forking mechanism upon metavariables, because there were plenty of odds and ends that must be kept in mind. On the other hand, I enjoyed the challenge. The implementation of the other functions like the parser generation went smoother than I expected. However, if I had a few more weeks I would definitely improve some functions, like the ability to define metavariable types or to improve the Lexer so the exact names of the variables can be shown in the result screen.

All in all, I think I reached the main goal of my work and the completed software will be useful and can be used in the HARP project as a parser generator, pattern matcher engine.

Bibliography

[1] Jason Lecerf, John Brant, Thierry Goubier, Stéphane Ducasse. A Reflexive and Automated Approach to Syntactic Pattern Matching in Code Transformations. ICSME 2018 - 34th IEEE International Conference on Software Maintenance and Evolution, Sep 2018, Madrid, Spain. ff10.1109/ICSME.2018.00052ff. ffhal-01851857f

[2] Horpácsi, Dániel, Kőszegi, Judit, and Horváth, Zoltán. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In Lisitsa, Alexei, Nemytykh, Andrei P., and Proietti, Maurizio, editors, Proceedings of the Fifth International Workshop on Verification and Program Transformation, Uppsala, Sweden, 29th April 2017, volume 253 of Electronic Proceedings in Theoretical Computer Science, pages 92–108. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.253.8.

[3] Horpácsi, Dániel, Kőszegi, Judit, and Thompson, Simon. Towards Trustworthy Refactoring in Erlang. In Hamilton, Geoff, Lisitsa, Alexei, and Nemytykh, Andrei P., editors, Proceedings of the Fourth International Workshop on Verification and Program Transformation, Eindhoven, The Netherlands, 2nd April 2016, volume 216 of Electronic Proceedings in Theoretical Computer Science, pages 83–103. Open Publishing Association, 2016. DOI: 10.4204/EPTCS.216.5.

[4] Aho, Alfred V. and Lam, Monica S. and Sethi, Ravi and Ullman, Jeffrey D. : Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., 2006, [1009], IBSN – 0321486811.

[5] https://doc.qt.io/qt-5/qregularexpression.html#details [Date accessed: 2020.05.12.]

[6] https://www.qt.io/download? [Date accessed: 2020.05.12.]