FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Generated parser for the console language of the Algorithms library
Student:	Ondřej Štorc
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Study the syntax and semantics of the language used in the console interface of the algorithm library (ALT) along with its tree representation.
Study the parser definition in the ANTLR tool.
Propose a way to integrate a parser generated by the ANTLR tool into ALT in order to substitute the hand-written parser of the console interface language.
Implement the substitution of the current hand-written parser of the console interface language with the one generated by the ANTLR tool.
Test the generated parser with the existing tests and design new tests specifically to check the correctness of the generated parser.

Electronically approved by Ing. Michal Valenta, Ph.D. on 6 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Generated parser for the console language of the Algorithms library

Ondřej Štorc

Department of Software Engineering

Supervisor: Ing. Jan Trávníček, Ph.D.

May 11, 2023

Acknowledgements

I want to express my gratitude to my supervisor, Ing. Jan Trávníček, Ph.D., for the opportunity to choose this topic and for his supervision and expertise during the work on this thesis. I would also like to sincerely thank my family, especially my parents and twin brother, who have supported me, not only during my studies.

Declaration

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

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

In Prague on May 11, 2023

Ondřej Štorc

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Štorc, Ondřej. *Generated parser for the console language of the Algorithms library.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

This thesis explores the transition from a handwritten parser in the Algorithms Library Toolkit to a new parser generated with ANother Tool for Language Recognition 4 (ANTLR4). The thesis begins by familiarizing the reader with fundamental concepts of parsing and parsers and providing an overview of ANTLR4 and the Algorithms Library Toolkit. The thesis details the integration process of the ANTLR4-generated parser into the existing codebase of the Algorithms Library Toolkit. The approaches used to verify the correctness of the new parser are also described.

Keywords Algorithm Library Toolkit, Algorithm Query Language, ANTLR4, parser, AST, C++

Abstrakt

Tato práce zkoumá proces nahrazení ručně psaného parseru v Algorithms Library Toolkit novým parserem generovaným pomocí nástroje ANother Tool for Language Recognition 4 (ANTLR4). Práce začíná seznámením čtenáře se základními koncepty parsování a parserů, stejně jako s představením ANTLR4 a Algorithms Library Toolkit. Následně je v práci popsán proces integrace generovaného parseru, pomocí nástroje ANTLR4, do stávajícího kódu Algorithms Library Toolkit. Dále jsou popisovány postupy použité k ověření správnosti nového parseru.

Klíčová slova Algorithm Library Toolkit, Algorithm Query Language, ANTLR4, parser, AST, C++

Contents

List of Figures

List of Source Codes

List of Tables

Introduction

Algorithm Library Toolkit (ALT) and its interface applications, such as *aql2* (console interface for the library) and web-based interface accessible from `alt.fit.cvut.cz/webui` are used in several courses in the curriculum of the Faculty of Information Technology of Czech Technical University in Prague, mainly in *BI-AAG.21*. This library allows for a quick demonstration of presented algorithms and also allows students to verify their implementation of the algorithms against implementation in ALT.

The console interface of ALT is text-based, and so it requires parsing of user input provided via argument or by Read-Eval-Print Loop (REPL). In the current library version, the input is parsed through a manually written lexer and parser.

Manually written lexers and parsers are efficient, but the readability of that code is often not the best experience, and fixing bugs in such written parsers is not easy.

Automatically generated parsers are often not as efficient as manually written ones, but their advantage is in the readability of the source from which the parser is generated. These sources, which describe the given grammar, can be pretty similar to the examples of grammar shown in course BI-AAG.21, which makes it more accessible to other students or developers in general.

The topic of this work is to replace the current, manually written lexer and parser with those generated by a tool called ANother Tool for Language Recognition 4 (ANTLR4) [1].

Chapter 1

Goal of thesis

The goal of this bachelor thesis is to describe *ANother Tool for Language Recognition 4* (ANTLR4), *Algorithm Library Toolkit* (ALT), and then replace the current parser of *Algorithm Query Language* (AQL) in the ALT console interface with one generated by ANTLR4.

The first part of the work aims to explore the two tools, describe the functionality of the ANTLR4 library and the architecture of ALT, and then propose a way of integrating them.

In the second part, the main focus will be on the implementation itself, which will be carried out according to the results of the first part of the thesis.

In the last part of this thesis, we will explain how the new parser was tested and ensured that the implementation accepts the same input range and, if necessary, how the bugs were fixed.

The outcome of this thesis is to make the parser more readable to new users and readers and to make it easier to extend parsing in the future if necessary.

Chapter 2

Analysis

In the first part, we will introduce the basic terminology used in this thesis and also introduce the readers to the ANother Tool for Language Recognition Library and the Algorithmic Library Toolkit.

2.1 Definitions

This thesis target is not to introduce the reader to the topic of grammar and its formal definition, but this section will use quite a few of them so that we will make at least some (more or less) informal definitions of them.

We can define grammar as a quadruple (N, Σ, P, S) [2], where:

- N is a set of non-terminals, symbols that do not have any meaning in the language but are used by grammar to define rules.
- Σ is a set of terminals, which are letters of the language which the grammar should accept. Typically in programming languages, this set contains all possible tokens.
- P are rules describing how the grammar accepts the input.
- S says from which non-terminal the grammar should start accepting input. In grammar that describes programming language, there can be a rule named *compilationUnit*. [1]

[1] This is arbitrary naming, but such a rule can be found in ANTLR4 definition of C language [3]

We can classify grammar into different categories according to the Chomsky hierarchy, which consists of four types of formal grammar. The Chomsky hierarchy is comprised of the following categories [2]:

1. *Unrestricted* grammar[2]
2. *Context-sensitive* grammar[2]
3. *Context-free* grammar has every rule in the form of $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$. Most programming languages can be described with context-free grammar.
4. *Regular grammar* has all rules in the following format $A \to \alpha B|\alpha$, where $A, B \in N$ and $\alpha \in \Sigma$. Typically, the lexer would work with such grammar.

We say that x *derives* y ($x \Rightarrow y$) if there is a rule from the grammar that translates input x into y, where $x = \alpha A \beta$ and $\alpha, \beta, y \in (N \cup \Sigma)^*, A \in N$ [2, 4]. $x \overset{k}{\Rightarrow} y, k \in (0, \infty)$ is a sequence of k derivations which takes as an input x and produces y. Common notation is also $x \overset{*}{\Rightarrow} y$, which says that in any number of derivations, we will produce y, as is $x \overset{+}{\Rightarrow} y$, where we say that in one or more derivations, we will produce y. Finally, we can also define left-most derivation. In this derivation, we always derive the most-left non-terminal in the sentential form. We can write it as $x \underset{lm}{\Longrightarrow} y$. In the same style, we can also define right-most derivation [2, 4].

Sentential form of grammar with initial rule S is $a \in (N \cup \Sigma)^*$ if $S \Rightarrow a$). $S \overset{*}{\Rightarrow} a, a \in \Sigma^*$ is a sentential form and also a sentence from the grammar [2].

Left-recursive grammar is a context-free grammar that has at least one rule (A), which can be derived into a sentential form, which has itself as the left-most symbol.

$$A \overset{+}{\Rightarrow} A\alpha, \alpha \in (N \cup \Sigma)^*$$

This type of grammar can lead to infinite recursion in predictive parsers since we cannot determine when to stop with recursion and must be transformed into an equivalent grammar that is not *left-recursive* [2].

[2]Unrestricted and Context-sensitive grammar are not discussed here since they are out of the scope of this thesis.

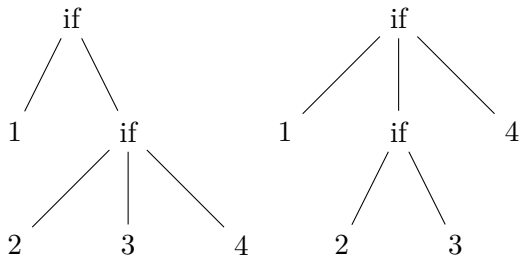

Figure 2.1: Parse tree of a word `if 1 if 2 3 else 4` in ambiguous grammar from Example 1

We can also distinguish between two types of recursion: direct and indirect. The direct recursion is the one where we need only to perform one derivation to get the required sentential form. Indirect recursion is a recursion where after one derivation, we will not get the recursive sentential form. But with two or more derivations, we will get the sentential form shown above [2].

Ambiguous grammar is grammar that can have two different left-most or right-most derivations for one input.

Example 1 (Dangling else). Consider grammar:
$G_1 = (\{\texttt{STM}, \texttt{IF}, \texttt{EXP}\}, \{\texttt{if}, \texttt{else}, \texttt{0, 1, ..., 9}\}, P, \texttt{STM})$ and rules P:

```
STM -> IF | EXP
IF  -> if EXP STM
IF  -> if EXP STM else STM
EXP -> [0-9]
```

Grammar G_1 also generates the following string: `if 1 if 2 3 else 4`. However, it can be generated in two different ways, as shown in Figure 2.1. In the first tree, the else branch statement belongs to the inner if statement, and in the second tree, it belongs to the outer if statement. Both of these results are valid, so grammar containing these rules is ambiguous.

2.2 Lexer

In the process of accepting user input, the first step is to analyze the input to see if it contains only valid sequences of characters. If we would compare it with reading English text. We would look at the text itself and try to

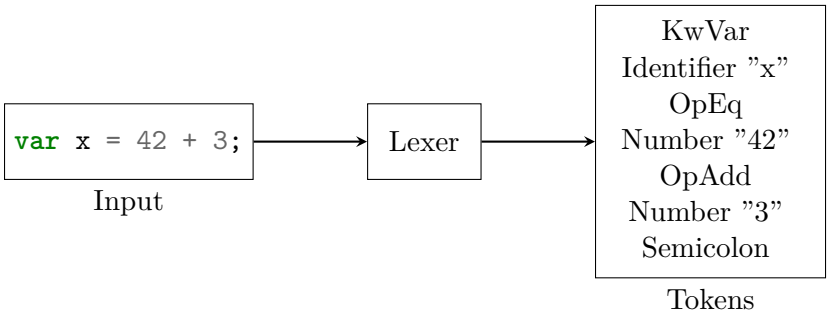

Figure 2.2: Lexer example

recognize all words and try to group them into groups like verbs, adjectives, and more. We would do this without actually understanding the meaning of the text, and this is what a lexer does. More formally lexer's job is to analyze the input string and group the characters into tokens according to the rules defined by the formal grammar; this process is known as *Lexical Analysis* [4].

Tokens are distinguished by their type and, if necessary, by their content. Each token represents a specific syntactic element of the language, such as keywords, identifiers, operators, punctuation, and literals.

In Figure 2.2, we can see that the input text is split into seven tokens:

1. Var keyword token of type ***KwVar***.
2. Identifier token with value ***x***.
3. Token with type of *OpEq*. This token does not need any value since it is already fully qualified by its type.
4. Number token, with the value of **42**
5. Token with type of *OpPlus*.
6. Number token, with the value of **3**.
7. The token representing a semicolon.

This is minimal information on what we need to know about tokens. In practical usage, we also want to know the location of a token in the input stream, which is useful, for example, when reporting errors during the later stages of the parsing and interpreting code [4].

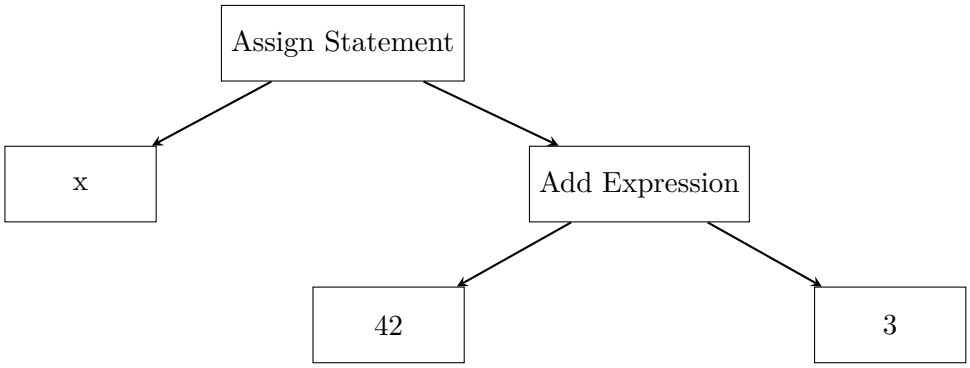

Figure 2.3: Abstract Syntax Tree example

2.3 Parser

After the lexical analysis, the parser accepts a stream of tokens and tries to match them with the grammar for which it was created. This is also called syntactic analysis. The output of such a process is often a tree-like structure called an Abstract Syntax Tree, which is a hierarchical representation of input and clearly shows how each part of the input relates to the other.

The Abstract Syntax Tree for the input shown in Figure 2.2 can be visualized, for example, as shown in Figure 2.3. In the diagram, it can be seen that we have transformed a linear stream of tokens into a tree-like graph structure [4]. This structure does not contain all the lexical details in our example, as it has no information on the semicolon token. But it can, and for example, Microsoft C# compiler *Roslyn* stores these data in a structure called `SyntaxTriva`, which is part of the Abstract Syntax Tree [5], where it can be used for formatting or linting purposes.

2.3.1 LL(1)

LL(1) is a model describing top-down parsing. The acronym LL(1) stands for Left-to-right, Leftmost derivation, with one look-ahead token. This refers to the parsing strategy used by the parser, which reads the input string from left to right and constructs a left-most derivation of the input according to the rules of the input language grammar [4].

The LL(1) parser uses a predictive parsing table to determine which rule to apply at each step of the parsing process. This table is constructed by

Table 2.1: LL(1) First-Follow table

	Rule ($A \to \alpha$)	First(α)	Follow(A)
1.	`A -> a X`	`a`	ε
2.	`X -> a X`	`a`	
3.	`X ->` ε	ε	ε
4.	`X -> x`	`x`	

Table 2.2: LL(1) parsing table

	`a`	`x`	ε
`A`	1		
`X`	2	4	3

analyzing the language's grammar and computing the First and Follow sets for each non-terminal symbol. The *First* set of a non-terminal represents the set of tokens that can begin a string derived from that non-terminal, while the *Follow* set represents the set of tokens that can follow a non-terminal in a derivation [4]. From the constructed LL(1) parsing table, we can determine what kind of rule we should apply to expand non-terminal for a given character during parsing.

Example 2 (LL(1) table). Consider grammar $G_2 = (\{\texttt{A}, \texttt{X}\}, \{\texttt{a}, \texttt{x}\}, P, \texttt{A})$ and rules P:

```
A -> a X
X -> a X
X -> ε
X -> x
```

Then LL(1) computing the table of the First-Follow symbols would produce Table 2.1. And the transformation to the parsing table will produce the table 2.2. From this table, for example, we can see that if we want to expand non-terminal `X` and our first unprocessed character is `x`, then we use rule 2.

2.3.2 Recursive descent parser

The common technique for implementing the parser is called *recursive descent parser*. This approach belongs to the top-down parser category, where the

parser starts with the initial non-terminal in grammar and tries to match the input to the grammar by recursively calling the rules it needs.

When creating a recursive descent parser, it makes sense first to construct LL(1) table (or its variant), which suggests how the code should be written.

For example, the GNU Compiler Collection parser for the C language is written with this method [6] as is the LLVM Clang front-end parser [7].

2.4 ANother Tool for Language Recognition 4

ANTLR4, or ANother Tool for Language Recognition 4, is a robust parser generator capable of reading, processing, executing, or translating structured text or binary files. ANTLR4 is a popular choice for building languages, tools, and frameworks. Taking grammar as input, ANTLR4 generates a parser that can parse a given grammar and build a parse tree[1].

ANTLR4 supports generating the parser in several languages. At the time of writing this thesis, the officially supported languages are the following: C#, C++, Dart, Go, Java, JavaScript (as well as TypeScript), Python, and Swift.

When generating the parser, the user can specify several options for the ANTLR4 that alter the generated parser:

- **Listener** and **visitor**: for a given parser, generates a visitor or Listener. Through each of these, the parsed tree can be traversed and evaluated. These options are not mutually exclusive.

- **package** specify in which package the lexer and parser will be located.[3]

- **language** tells ANTLR4 in which language to generate the lexer with the parser.

This is not an exhaustive list of options, but it covers the ones used in this thesis [8].

[3]Package is used in Java, in other languages namespace or other equivalent structure will be used.

```
stat: expr ';' // expression statement
    | ID '(' ')' ';' // function call statement ;
    expr: ID '(' ')' | INT
    ;
```

Code 2.1: Ambiguous grammar example from *The definitive ANTLR 4 reference* - Grammar definition

2.4.1 Adaptive LL(*) parsing

Adaptive LL(*) (also known as ALL (*)) parsing is used in the ANTLR4 tool. ALL(*) is a parsing algorithm that extends the capabilities of traditional LL parsers. LL parsers must stop at each decision point (to see which rule to apply). Due to this, LL(1), LL(k), and LL(*) have to perform a static analysis on the input grammar to precompute the table or similar structure, according to which the decision point will be resolved. This approach has an issue because the static analysis must consider all possible inputs. The ALL(*) does not perform the static analysis but does dynamic analysis at runtime. With that, it does not have to consider all inputs but only the finite set of inputs given to the parser. This update allows ALL(*) to be applied to all non-left-recursive context-free grammars and context-free grammars that contain direct left-recursion but not indirect ones [9].

Adaptive LL(*) can accept ambiguous grammar. This can be done because ALL(*) always selects the first possible option when deciding which rule to take. In our example of an ambiguous grammar Figure 2.1, the ALL(*) (and thus ANTLR4) would choose the first option where the *else* branch is connected to the inner *if* statement. Another example from *The definitive ANTLR 4 reference* [8] is shown in Code 2.1 and Figure 2.4. Again, in this case, the ANTLR4 will choose the first option, and that is the expression statement.

2.4.2 Grammar definition

ANTLR4 requires the user to define the input grammar in the EBNF format. EBNF stands for Extended Backus-Naur Form. The Backus-Naur form (BNF) allows specifying non-terminal symbols (rules) and how they expand to other

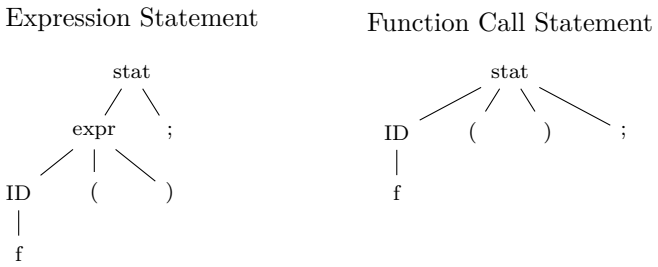

Figure 2.4: Ambiguous grammar example from *The definitive ANTLR 4 reference* - Parse Tree

```
S := '-' DIGIT | DIGIT
DIGIT := D | D DIGIT
D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Code 2.2: Integer representation in BNF

```
S := '-'? D+
D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Code 2.3: Integer representation in EBNF

non-terminal and terminal symbols [10]. In Code 2.2, we can see a grammar that matches all integers written using BNF notation.

In Extended Backus-Naur Form (EBNF), special symbols such as '?', '$*$', and '+' define the repetition and optionality of elements in the grammar rules. The '?' symbol denotes that an element is optional, meaning it can appear either once or not at all. The '$*$' symbol signifies that an element can be repeated zero or more times, allowing for any number of repetitions, including none. Lastly, the '+' symbol represents that an element can be repeated one or more times, ensuring that it appears at least once [10]. These symbols play a crucial role in EBNF, enabling the concise expression of complex grammar rules while capturing the flexibility and variation in programming languages, data formats, and communication protocols. The example in Code 2.2 could be rewritten into EBNF as shown in Code 2.3.

Lexer rules are parsed from top to down, so rules defined at the top of the file have higher priority than rules at the bottom. Thus it makes sense to specify more specific lexer rules at the top [8].

```
// ...
@parser::members{
    std::set<std::string> symbols;
}
// ...
var_decl: ID = expr {symbols.insert($ID.text);};
// ...
```

Code 2.4: ANTLR4 Grammar Action example

```
// ...
@parser::members{
    std::set<std::string> symbols;
}

var_access : '$' ID {symbols.count($ID.text)>0;}?;
```

Code 2.5: ANTLR4 Semantic Predicate example

Grammar can be split into multiple files, reused in various projects, and annotated with additional information, altering how ANTLR4 generates the lexer, parser, and other possible files.

Grammar can also be extended by adding grammar actions. Grammar actions are snippets of code in the target language injected into the lexer or parser. An example of such grammar action can be seen in Code 2.4. This example shows part of the parser grammar, which defines via `parser::members` member variable of the parser. When parsing the input, it puts the identifier name into the set of encountered symbols. This example is an arbitrary one and not suited for actual usage [8].

The grammar actions are powerful tools that can be used, for example, in evaluating the input text; however, they do not alter how the lexer or parser works; they only add side effects to the parsing or tokenizing of the input. ANTLR4 allows us to specify semantic predicates to alter how the input is recognized. These predicates are again fragments of code in the target languages; however, this time, they must be evaluated at values of **true** or **false** (or equivalent in the target language). And once again, these predicates can be specified for both the lexer and parser [8].

In Code 2.5, we can see parser semantic predicates that turn on or off the rule `var_access` based on the presence of an identifier in the primitive symbol table. The same syntax is used for the parser semantic predicates. In ANTLR4, these predicates should not be used for semantic validation; instead, other approaches such as the *Listener* or *Visitor* pattern should be used. Semantic predicates should only be used to resolve an ambiguous grammar [11].

2.5 Algorithm Library Toolkit

This section relies primarily on information from [12, 13], and also from the source code of the Algorithm Library Toolkit [14]. Algorithm Library Toolkit (ALT) is a tool developed at the Faculty of Information Technology, Czech Technical University in Prague. ALT is a software tool designed to manipulate data structures in the field of stringology, such as automata, grammars, and trees, among others. In addition, ALT provides numerous algorithms to work with these data structures. ALT supports the console line interface and the web-based interface.

2.5.1 Algorithm Query Language

This covers the basics of the Algorithm Query Language. Detailed information about it can be found in [12] and [13]. Algorithm Query Language (AQL) is a language created for usage in the Algorithm Library Toolkit. This language syntax is similar to the Bash language. It supports literals in the form of strings, integers, and doubles. AQL supports C-like comments. For single-line comments, we can use `//`, and for potentially multiline, we use `/**/`.

AQL allows the user to pass the result of one statement to the input of another statement using the pipe (`|`). This enables the user to chain statements as in the Bash language. We can see the example of this in Code 2.6. Note that `-` (dash) represents the result of the previous command.

AQL allows the user to use bindings and variables. Bindings are defined at the startup of the console line interface and are read-only values. These values can be accessed by using `#` and the name of the binding. There are always two bindings present, one for standard input (`stdin`) and one for standard

```
> print 1 | IsSame - "world"
0
> print "hello" | IsSame "hello" -
1
```

Code 2.6: Example of statement chaining

```
> print #stdin
-
> execute 42 > $a
>print $a
42
```

Code 2.7: Example of bindings and variables

```
print "a + (a b)*" | string::Parse @regexp::RegExp -
| regexp::convert::ToAutomaton - | string::Compose -
```

Code 2.8: Example of ALT algorithms usage in AQL. The command was manually formatted.

output (`stdout`). Variables are created at runtime and are a way how to pass values between two commands. When accessing a variable, we start with `$` (dollar sign) and then follow it by the name of the variable. Variables hold information about what type of value they store, but at any time, we can override the value of the variable with a new value and potentially change the type of variable. Examples of using variables can be found in Code 2.7.

AQL contains several flow-control commands. Two dominant are `if` and `while` commands. We can also use `introspect` command to get information about available algorithms, overloads, variables, bindings, casts, and other features. But the main feature of AQL is in its connection to the other modules of the Algorithms Library Toolkit. An example of such usage is shown in Code 2.8; this example was taken from [13]. In the example, we can see a print statement that contains chained statements. At first, we pass a string `"a + (a b)*"` into `string::Parse` algorithm, which parses the string into `regexp::RegExp` data-type. This regular expression is then converted into an automaton. Then we covert, the automaton into a more human-readable format, which is then printed into standard output.

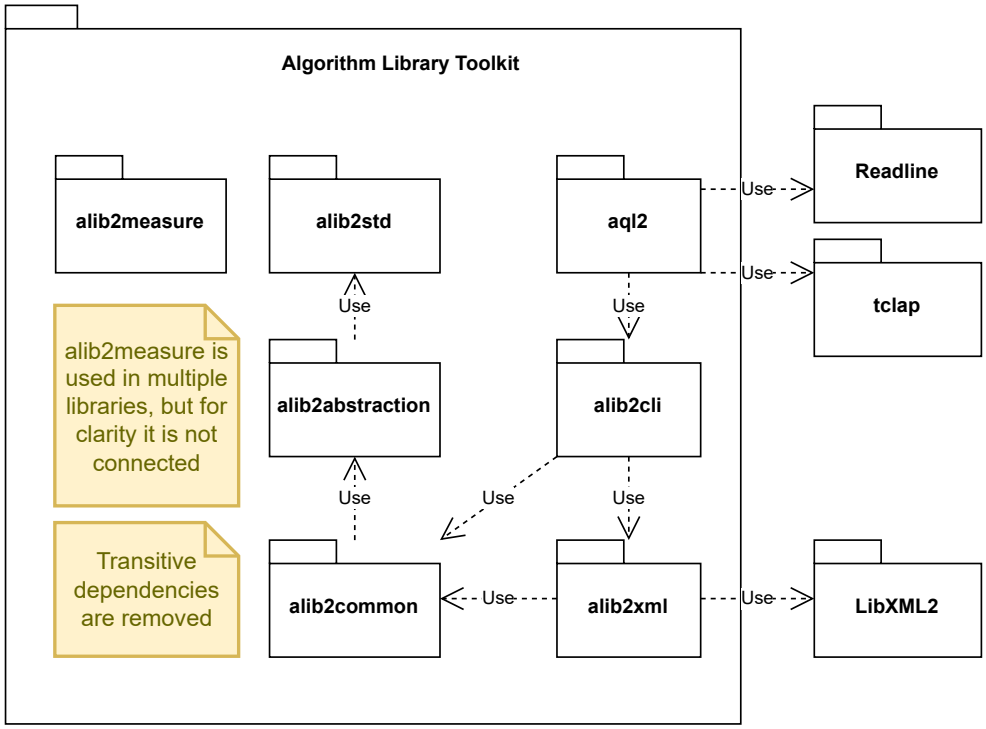

Figure 2.5: ALT modules diagram

2.5.2 Architecture

ALT architecture is highly modular. In Figure 2.5, we can see the core modules of the ALT console line interface. There are other modules, but these are not directly related to this thesis. The entry point of the CLI is in **aql2** executable.

2.5.2.1 aql2

aql2 is the entry point of the console line interface. It wraps all user input into appropriate data structures for evaluation. These inputs can come from program arguments or interactive text input (REPL). It also controls the evaluation loop.

2.5.2.2 alibcli

alibcli handles the input evaluation and parsing of it. This module contains a lexer, a parser, an abstraction over user input, and an abstract syntax tree

of the Algorithm Query Language (Section 2.5.1). This module also contains all the logic necessary to evaluate the AST produced by the parser.

2.5.2.3 alib2xml

alib2xml module handles all work related to the XML file format, allowing **alib2cli** to read the input from XML files, which can be quite complex for some data structures. Examples of such input can be found in the ALT repository.

2.5.2.4 alib2common

alib2common module implements basic data structures to represent primitive data types, standard containers, exceptions, and the common object wrapper for any datatype in the library type hierarchy. Additionally, the module includes core functionality, such as support classes for the visitor pattern, component classes, and a handler for printing stack traces in the event of segmentation faults.

2.5.2.5 alib2abstraction

This module is represented by `OperationAbstraction` and `ValueProvider` classes. These classes can be interconnected to represent algorithms, entities, and parameters.

Each algorithm, cast operation, and data type implemented in the algorithm library toolkit are registered within the internal structures of the abstraction module, enabling its subsequent execution via the command-line interface. Registration is accomplished by constructing global variables within unnamed namespaces or calling functions.

2.5.2.6 alib2std

alib2std is the module that contains the implementation of some basic data structures or improvements to the structures already present in the standard library of C++.

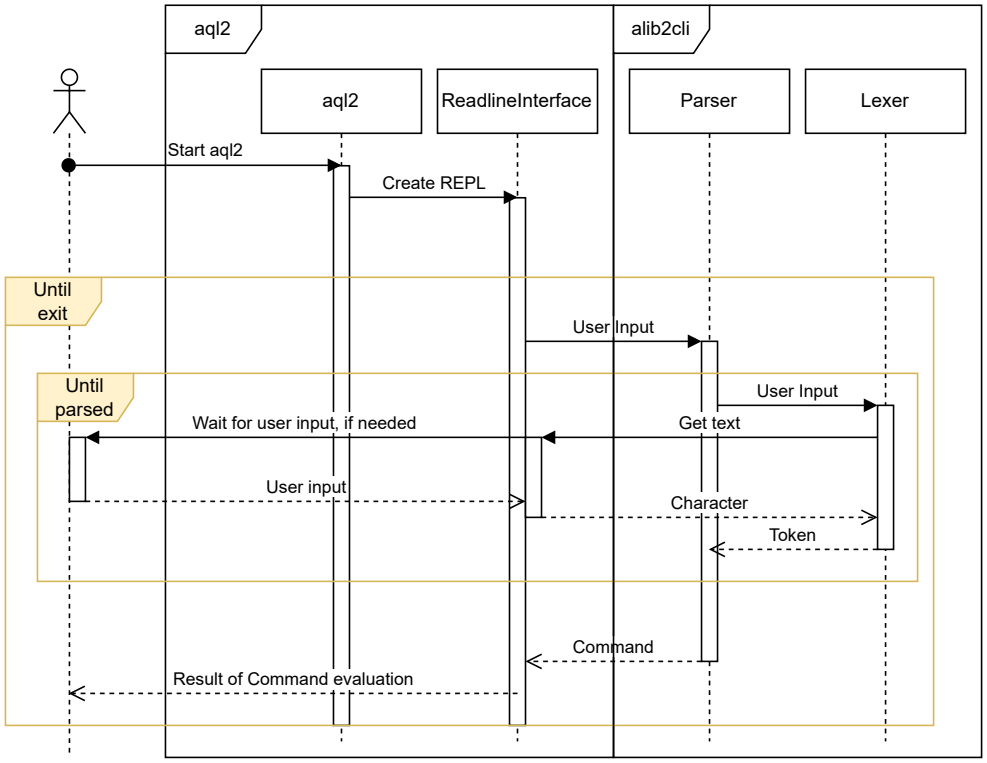

Figure 2.6: Old CLI sequence diagram

2.5.2.7 alib2measure

This module includes the data structures necessary to collect measurements. Measurements are organized into frames, each of which may contain subframes. Each frame stores the time spent within the frame, exclusive of its subframes, the change in memory usage resulting from dynamic allocations and deallocations, and general-purpose counters.

The Algorithm Library Toolkit can be extended to support more algorithms and data types. This is done through dynamically loaded libraries and by registering the algorithms in the libraries as described above.

2.5.3 Parsing and evaluating user input

In Figure 2.6, we can see a simplified process of evaluating user input.[4] The interesting thing here is that the Parser directly asks Lexer for user input,

[4]Note that some layers of abstraction are not shown in the diagram for the sake of clarity.

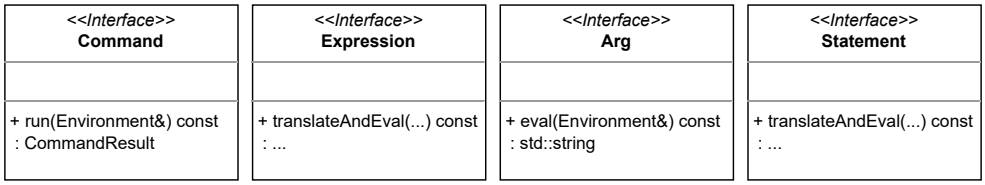

Figure 2.7: Base classes of ALT AST

and Lexer is also communicating with `ReadlineInterface`. This can be done because the parser, as is the lexer, is handwritten.

When **aql2**, the console line interface of the Algorithm Library Toolkit, starts, it parses the command line arguments. These arguments can be input files and commands passed as arguments from the command line, a redirected standard input stream, or an indicator to enter interactive mode. From these arguments, instances of `LineInterface` are constructed. When the app knows what to execute, it executes the arguments in the order passed to the app. This is done in `Prompt` class, which passes `LineInterface`s to `Enviroment` instance. In this instance is each `LineInterface` wrapped in `CharSequence`, which provides more granular control over the input.

Finally, with this wrapper, a Lexer instance is created, and then a Parser instance is created. The method `parse()` is called on the constructed parser, serving as the entry rule for the handwritten parser. The exact parsing calls depend on the input. As shown in Figure 2.6, the parser asks Lexer for tokens, and Lexer communicates through `CharSequence` with concrete `LineInterface`. If the parse fails, an exception is raised; otherwise, the parser's output will be an abstract syntax tree. Finally, this tree is evaluated.

The abstract syntax tree consists of five types of nodes. In Figure 2.7, we can see four of them, but there is missing `Option`, which does not have any common evaluation method, so showing their interface would be meaningless.

Command nodes are the only nodes that can be parsed as a top-level statement or can be found inside the *Block* command. At the time of writing this thesis, there are 35[5] commands. The most notable commands are:

[5]Before this thesis, there were only 34. `introspect ast` command was added as a side effect of this thesis.

- **block** command works functionally the same as in other languages. They create a new scope and can group several commands. The creation of scope is important since some commands can be used only in the global scope and others in the non-global scope.

- **print** and **execute** commands are the most basic commands that accept statements or expressions. *execute* only evaluates the data passed and does not show them. `print` command works similarly to the `execute` command, but also prints the result.

- **if** and **while** commands have the same meaning as in C-like languages. One limitation placed on them in the Algorithm Library Toolkit is that they cannot be used in a global scope and can only be defined in nested scopes.

- **procedure** and **function** declaration are used to declare reusable code. These declarations are available only in the global scope.

Statements can be chained or executed on their own. In the Algorithm Library Toolkit, we can divide statements into three groups; `Single` statement, `Cast` statements, and `Common` statements.

`Common` statements are typically simple values. A simple value can be as simple as literal or as complex as a result of a previous statement or redirect from a file. However, they still do not execute anything independently; they just hold some value. `Cast` statements have the same semantic meaning as in other languages, where they take any value and try to interpret it in the desired data type.

On the other hand, `Single` statement is a much more complex statement. In its most basic and frequently used variant, it accepts the name of the algorithm to be executed and the corresponding parameters. An example of a `Single` statement is `IsSame 1 "2"` where we have an algorithm with name `IsSame` and two parameters: an integer `1` and a string `"2"`. `Single` statements can also take additional arguments as template arguments, which helps determine the concrete algorithm used in the evaluation. There are also category options, which are described in grammar but are not currently used in the evaluation of the `Single` statement.

Table 2.3: Precedence and Associativity of Operators

Operator	**Precedence**	**Associativity**
Function call	Highest	Left to Right
Method call		Left to Right
Postfix Increment/Dec.		Left to Right
Cast		Right to Left
Unary Plus		Right to Left
Unary Minus		Right to Left
Logical Not (`!`)		Right to Left
Binary Negation (∼)		Right to Left
Prefix Increment/Dec.		Right to Left
Multiplication, Modulo, Division		Left to Right
Addition, Subtraction		Left to Right
Relational (`>, >=, <, <=`)		Left to Right
Equality (`==, !=`)		Left to Right
Bitwise XOR (∧)		Left to Right
Bitwise AND (`&`)		Left to Right
Bitwise OR (`>, >=, <, <=`)		Left to Right
Logical AND (`&&`)		Left to Right
Logical OR (`||`)		Left to Right
Assignment (`=`)	Lowest	Right to Left

Expressions are similar to the expression in other languages. They support chain expressions by operators with the precedence described in Table 2.3.

Args or arguments are simple values that can be either identifiers or bound values. Identifiers are any unmatched word that starts with a letter from the English alphabet or with `_` and contains only these letters or digits and colon sign (':'). Bounded arguments are environment variables passed from the command line. The name of the bound argument starts with `#` and is followed by an identifier or an integer.

Chapter 3

Implementation

This chapter will describe the implementation and integration process of the new parser generated by ANTLR4. Note that we will be using the '...' symbol in code listings, which denotes that we have removed part of the original code from the listing to make it less verbose.

3.1 Preparing grammar

The first step in generating a parser is to specify the grammar for which the parser will be generated. The starting point for this part of the work was grammar definition (from which part can be seen in Code 3.1) on the Algorithm Toolkit Library website.

```
arg : HASH_SIGN (INTEGER | IDENTIFIER) | IDENTIFIER;
batch_or_expression: KW_EXPRESSION expression | KW_BATCH? batch;
runnableParam: qualifiedType DOLAR_SIGN arg;
command
: KW_EXECUTE batch_or_expression | KW_PRINT batch_or_expression
| KW_EXIT batch_or_expression? | KW_RETURN batch_or_expression?
...
;
parse: command (SEMICOLON command) END | END;
assign_expression: or_expression (ASSIGN_OPERATOR assign_expression)?;
or_expression: and_expression (OR_OPERATOR and_expression)*;
and_expression: bitwise_or_expression (AND_OPERATOR bitwise_or_expression)*;
```

Code 3.1: Part of grammar from ALT documentation

```
// Operators or otherwise important characters
HASH_SIGN : '#';
AT_SIGN : '@';
LEFT_PAREN : '(';

// Keywords
KW_AST : 'ast';
KW_FROM : 'from';
KW_TO : 'to';
KW_ALGORITHMS : 'algorithms';

// Integers, identifiers, and string
fragment DIGIT: [0-9];
INTEGER : DIGIT+;
DOUBLE: DIGIT+'.'DIGIT* | DIGIT*'.'DIGIT+;
IDENTIFIER  :
([a-z] | [A-Z] | '_' ) ([0-9] | [a-z] | [A-Z] | '_' | ':' )*;

// Whitespace, comments
WS : [ \t]+ -> channel(HIDDEN);
COMMENT : '//' (~[\r\n] | '\\'EOL)* -> channel(HIDDEN);
MULTILINE_COMMENT : '/*' .*? '*/' -> channel(HIDDEN);
```

Code 3.2: ALT CLI Lexer grammar

3.1.1 Lexer

Seemingly the easiest step was to define the lexer for the input grammar. This was done by reviewing the original grammar and finding all keywords or other lexical elements.

We can group all the lexical elements into a few groups, as shown in Code 3.2. For each group, only a few examples are shown. Operators and keywords are straightforward and do not require any other explanation.

Integers, identifiers, and other non-constant tokens use the features of the EBNF syntax to the fullest. We can also see the usage of fragment `DIGIT`. Fragments are lexer tokens that cannot be referenced in the parser, but because of them, we can avoid duplicating some syntax in the lexer grammar.

Finally, we have arguably the most complicated part of the lexer rules, which are rules for matching whitespaces and comments. We can see that each token is redirected to the channel with the name `HIDDEN`.

```
expression
 : lhs=expression op=(STAR_SIGN | PERCENTAGE_SIGN | SLASH_SIGN) rhs=expression
 | lhs=expression op=(PLUS_SIGN | MINUS_SIGN) rhs=expression
 | lhs=expression op=(LESS_SIGN | ... | MORE_SIGN) rhs=expression
 | lhs=expression op=(EQUAL_OPERATOR | NOT_EQUAL_OPERATOR) rhs=expression
 | lhs=expression op=CARET_SIGN rhs=expression
 | lhs=expression op=AMPERSAND_SIGN rhs=expression
 | lhs=expression op=PIPE_SIGN rhs=expression
 | lhs=expression op=AND_OPERATOR rhs=expression
 | lhs=expression op=OR_OPERATOR rhs=expression
 | <assoc=right> lhs=expression op=ASSIGN_OPERATOR rhs=expression
 | prefix_expression // Subrule for all prefix expressions
 ;
```

Code 3.3: New expression grammar

This channel is a predefined channel from ANTLR4 and is used when we do not want to consider these tokens in parsing, but we still want them to stay in the token stream, so we can reconstruct the input if needed. In ANTLR4, there is also directive `-> skip` which says to skip the token completely.

We can also see the usage of `~`, which allows us to negate some expressions. We can see the usage in the `COMMENT` token, where we want to accept characters that are not a new line or carriage return. The `COMMENT` token also allows us to escape a new line inside a single-line comment, which is possible in C and C++; however, it was not possible in the old version of the parser.

3.1.2 Expressions

ANTLR4 can handle much simpler rules for expression syntax than the one shown in Code 3.1. This is due to the ability of ANTLR4 to accept ambiguous grammar and direct left recursion. The final grammar with some simplification for expression is shown in Code 3.3. This grammar follows the operator precedence described in Table 2.3.

There are a few things worth noting: All expression rules produce the same syntax tree nodes, which contain different subtrees. Each node contains `lhs`, `rhs` with expressions, and a `op` member, which defines which operation we want to use. Later in the transformation of the syntax tree to AST, we will use this. In the rule describing assign expression, we use `<assoc=right>` syntax to tell ANTLR4 that we want the assignment operator to be right-associative.

```
print < ../../input.xml
execute 1 > /tmp/out.xml
print < #file
execute 1 > "out.xml"
```

Code 3.4: Old file syntax usage

```
file
    : binding
    | string
    ;
```

Code 3.5: New file grammar

```
print < #file
execute 1 > "out.xml"
```

Code 3.6: New file syntax usage

Expressions are not limited to the binary expression shown in Code 3.3 but contain `prefix`, `suffix`, and `atom` expressions. These expressions have a similar syntax and do not use any new syntax, so they are not shown here, but they still can be found in the complete grammar.

3.1.3 Files

The old syntax supported parsing files and file paths (examples of old syntax are in Code 3.4). This was done by giving hints from the parser to the lexer. This communication was possible, as seen in Figure 2.6, where the parser sends messages to the lexer. This is not possible with an ANTLR4-based parser because we cannot give a hint from the parser back to the lexer. Thus, the support for direct file names was removed, and the only available options right now are putting the file name into a string or using binding. The new file syntax can be seen in Code 3.5, and an example of usage can be found in Code 3.6.

```
print batch
    automaton::simplify::EpsilonRemoverIncoming $automaton
    | automaton::determinize::Determinize -
    | automaton::simplify::Trim -
    | automaton::simplify::Minimize -
    | automaton::simplify::Normalize -
```

Code 3.7: Example of ambiguous grammar in ALT (newlines)

3.1.4 New Lines

New lines are not a simple topic, as it seems to be in the Algorithm Library Toolkit CLI. They have different meanings depending on the current state of the parse tree. If they are located inside blocks, their meaning is the same as in C or C++, and they do not matter. But in the top-level statements, they separate the commands instead of the semicolon, which is used inside blocks. This behavior is similar to the one in the Bash language. An example shown in Code 3.7 is not valid if it is inside the `block` command, but it is only because we miss the semicolon required at the end of each command inside the block. If it is entered as a top-level command, it is invalid because of the new lines. New lines are not allowed at all, and they are separating commands.

The approach which it is solved in this thesis is by using grammar actions and semantic predicates inside the lexer. In Code 3.8, we can see the implementation of this. Lexer holds an internal counter of how deep inside of block statement we are. If we are at the level 0, then that means that we need to consider new lines, and we disable the rule `BLOCK_NEWLINE`. Due to that, all new lines are matched by the rule `NEWLINE`; however, if the nested level is bigger than 0, we allow matching of the rule `BLOCK_NEWLINE`, and it takes precedence over the rule `NEWLINE` and consumes all newlines. This rule is also redirected to the hidden channel, so all new lines will not be considered inside the parser. This approach is not ideal, as the lexer has to know how the parser and grammar work, but due to the limitation of ANTLR4 runtime, this approach was taken. Finally, we need to use the rules inside the parser, and this can be seen in Code 3.9.

```
@lexer::members {
size_t nestedLevel = 0;
}
// ...

KW_BEGIN: 'begin' { nestedLevel++; };
KW_END: 'end' { nestedLevel--; };

ESCAPE_NEWLINE: '\\' EOL -> channel(HIDDEN);
// This new line is skipped if we are in a block
BLOCK_NEWLINE: EOL {nestedLevel>0}? -> channel(HIDDEN);
// Top level newline
NEWLINE: EOL;
```

Code 3.8: Handling new lines inside lexer

```
parse
    : NEWLINE* top_level_command (NEWLINE top_level_command?)* EOF
    | NEWLINE* EOF;
```

Code 3.9: Handling new lines inside parser

3.1.5 Top level commands

In the ALT language, there are 34 commands. Most of them can be used outside blocks. However, four of them (`if`, `while`, `break`, and `continue`) cannot be used as top-level commands, and three are not available within blocks (`undeclare` command, declaration of functions and declaration of procedures).

The old parser solves this by counting how deep it is currently nested and, after matching the beginning of the command deciding if it is allowed to use the command in the current nested level.

This approach could also be implemented in the new parser, and the first iteration of the parser used this approach when it parsed the input and then analyzed the parse tree to see if there were any violations of these constraints. However, this approach had one big drawback, and it had to parse the whole input and then analyze it once again to decide if it was the correct input or not. Instead, the grammar was rewritten to prohibit commands in unwanted scopes, as seen in Code 3.10. The rule `command` is never used directly, and only the `level_command` variants are used appropriately.

```
// All commands without scope constraints
commands: ...;

nested_level_command
    : command # NestedLevelCommand
    | command_if # If
    | command_while # While
    | (KW_BREAK | KW_CONTINUE) # CycleControl;

top_level_command
    : command # TopLevelCommand
    | KW_UNDECLARE ... # UndeclareFunction
    | KW_PROCEDURE ... nested_level_command # Procedure
    | KW_FUNCTION ... # Function;
```

Code 3.10: Commands definition inside parser

```
STRING: DQUOTE (STR_TEXT | EOL)* DQUOTE;

fragment DQUOTE: '"';
fragment EOL: '\r'? '\n';
fragment STR_TEXT: (~([\r\n\\] | DQUOTE) | ESC_SEQ)+;
fragment ESC_SEQ: '\\' ([btnrf\\] | | DQUOTE | EOF);
```

Code 3.11: String definition in lexer

3.1.6 Literals and identifiers

As literals, consider all integers, doubles, and strings. Integers and doubles are standard as in other languages. Strings support newlines inside them, which is uncommon in regular strings. The lexer rule for the string can be seen in Code 3.11.

Identifiers are more complicated. The token rules are defined in Code 3.12. Identifiers in the ALT language can match all alpha-numeric words. These words can also contain underscores, and, what is important, it also allows a colon character to be included, which is not permitted, for example, as part of identifiers in C or C++ language.[6]

This rule covers almost all possible identifiers; however, keywords are also allowed in ALT as an identifier. Thus, defining the identifier rule in the parser

[6] In fact in C++ double colon (::) is an operator [15].

```
IDENTIFIER: ID_LETTER (ID_LETTER | [0-9] | ':')*;

fragment ID_LETTER: [a-z] | [A-Z] | '_';
```

Code 3.12: Identifier definition in lexer

```
identifier
    : IDENTIFIER | KW_AST | KW_FROM | KW_TO
    | KW_NORMALIZATIONS | KW_DENORMALIZATIONS
    | KW_EXECUTE | KW_PRINT | KW_QUIT | KW_EXIT
    | ...
    ;
```

Code 3.13: Identifier definition in parser

that matches an identifier or any of the keywords was necessary. The simplified version of this rule is shown in Code 3.13.

3.2 Integrating new parser

With the defined grammar, the next step in replacing the old parser was incorporating the newly generated parser into the codebase of the Algorithms Library Toolkit and, more specifically, the `alib2cli` module.

3.2.1 CMake

The integration of ANTLR4 into the ALT and the project's build itself is done by CMake.

CMake is a cross-platform open-source build system generator that simplifies the process of building, testing, and packaging software. Developed by Kitware, CMake is designed to manage the compilation process using platform- and compiler-independent configuration files. These configuration files, typically named `CMakeLists.txt`, describe the build process of a software project and specify various parameters, such as source files, dependencies, libraries, and compilation flags.

CMake generates native build files tailored to the target platform and build environment, such as Makefiles or Ninja build files for Unix-based systems, Visual Studio project files for Windows, or Xcode project files for macOS.

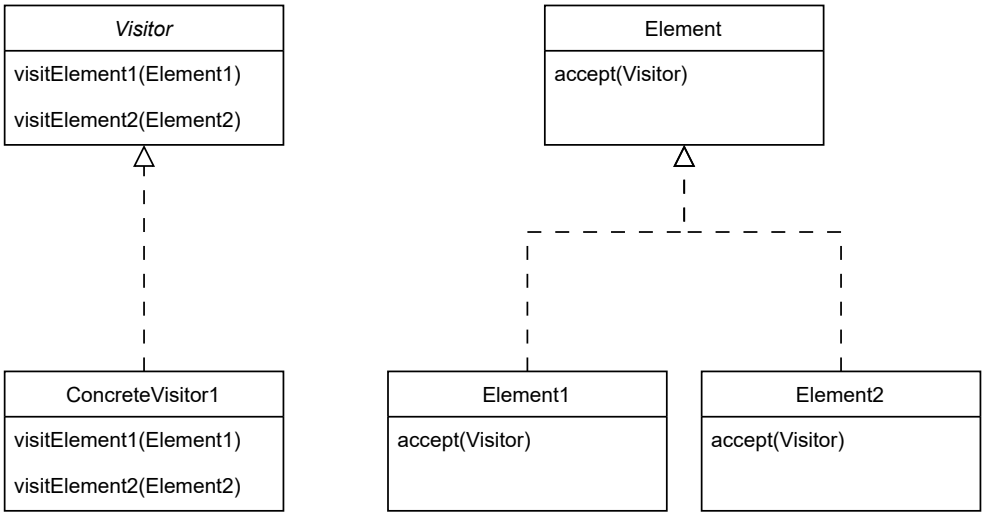

Figure 3.1: Visitor pattern

By abstracting away platform-specific details, CMake allows developers to focus on writing source code and defining build rules without worrying about the intricacies of different build systems [16].

3.2.2 Visitor pattern

ANTLR4 support two options for traversing the output of the generated parser, *Visitor* and *Listener*. The visitor pattern was chosen for its flexibility and because it can return values directly, making work with it more straightforward in this case than with the listener pattern. The listener pattern can be useful when we want to traverse all nodes in the syntax tree without the need to return specific values. One example where this could be useful is building a syntax highlighting engine. We would want to enter all nodes and then store where in the input it is located and what kind of syntax structure it represents.

The visitor pattern is a widely-used behavioral design pattern that facilitates the separation of concerns between data structures and the operations performed on them. This pattern is beneficial when dealing with object-oriented class hierarchies. By decoupling the operations from the data structures, the visitor pattern enables extensibility and maintainability [17]. As can be seen in Figure 3.1, the visitor pattern consists of the following components:

Visitor interface defines set of methods, usually called `visit()`, for each of the elements in the data structure. For example, in the case of ANTLR4, we will have a method for each node of the parse tree.

Concrete Visitor class provides the implementation for all `visit()` methods. Each concrete visitor can do different things; thus, it enables extending the functionality of the objects without modifying them.

ANTLR4 provides us with the base implementation of the `Visitor`, which has implemented all `visit()` methods. These methods do not do anything else except call the visit on all child nodes in the parse tree. With this, we do not have to provide the implementation for all methods and only override these we want to extend.

Element defines the interface for all elements that we do want to use in the visitor. In ANTLR4, this interface is described by the `ParseTree`, where it has abstract method `std::any accept(ParseTreeVisitor*)`.

Concrete Elements are the classes that represent the elements of the data structure we want to visit. In our case, these are the concrete parse tree nodes that the ANTLR4 produces.

3.2.3 Visitor for the Parser

In the Algorithms Library Toolkit, the visitor is used for the transformation of the parse tree, which is produced by the parser, to the Abstract Syntax Tree. The implementation of the visitor can be found in `AltVisitor` class in `alib2cli` module.

By default, the ANTLR4 produces `visit` method for each rule in the parser. For more complex rules, this is not detailed enough. For example, rule **semicolon_command**, as shown in Code 3.14, has 4 possible ways how it can be parsed. Due to that, we will have to have some non-trivial logic in the visitor that would decide how to construct the Abstract Syntax Tree node.[7]

[7]Note that this rule is fairly straightforward, but in the grammar are more complex rules.

```
semicolon_command
    : block
    | command_if
    | command_while
    | nested_level_command SEMICOLON
    ;
```

Code 3.14: Semicolon command definition in parser

```
semicolon_command
    : block # BlockSemicolonCommand
    | command_if # IfSemicolonCommand
    | command_while # WhileSemicolonCommand
    | nested_level_command SEMICOLON # SemicolonCommand
    ;
```

Code 3.15: Semicolon command definition with labels in parser

```
using namespace std;
// ...
any visitBlockSemicolonCommand(AltCliParser::BlockSemicolonCommandContext*);
any visitIfSemicolonCommand(AltCliParser::IfSemicolonCommandContext*);
any visitWhileSemicolonCommand(AltCliParser::WhileSemicolonCommandContext*);
any visitSemicolonCommand(AltCliParser::SemicolonCommandContext*);
```

Code 3.16: Semicolon command definition in visitor

We can avoid this situation by labeling sub-rules in the grammar itself, and this will tell ANTLR4 to generate a visit method for each sub-rule, and the logic for deciding which sub-rule is correct will be automatically generated by ANTLR4. The labeled example can be Code 3.15. Corresponding method headers for this rule can be seen in Code 3.16. These methods are usually simple, and that is by design. They should not decide if the grammar is correctly parsed or not; all this logic is left for the parser. Most methods could be implemented as one-liners, usually though they are not since they would not be as readable.

The only logic which is in the visitor is the handling of the optionality of tokens and rules or if they are allowed to repeat (see Section 2.4.2 for how to define these properties for the tokens or rules).

The visitor generated by the ANTLR4 has one property that has added

```
#include <any>

struct A { };
struct B : A { };

std::any foo() {
    return new B();
}

int main() {
    // Throws std::bad_any_cast
    A* a = any_cast<A*>(foo());
}
```

Code 3.17: `std::any` example

to the implementation more function calls, then would be ideal. All methods in the visitor pattern have to return the `std::any` type. This type is defined in the standard library [18]. `std::any` is a type-safe container for all copy constructible types. This type usually holds information about the stored type, and if supplied, the value [19]. The process of extracting the value from this container checks if the stored type information matches the one desired on the output, thus, is this container type-safe. This, however, does not work with inheritance and its usage. For example, let us have Code 3.17. This code will compile, but it will throw runtime `std::bad_any_cast` exception. If we replace the `std::any` return type in `foo` with either `A*` or `B*`, the code will work as intended.

This is, however, not applicable to the implementation of the interface of visitors created by ANTLR4. We are constrained by the fact that we must always return `std::any`. The only solution for this is then to cast the result of `foo` before we wrap it into the `std::any` container. For this reason, the implementation has the helper method `retPtr`, which constructs the desired object with supplied parameters and casts it to the base pointer type. The implementation of `retPtr` can be seen in Code 3.18 as well as how it would be used in Code 3.17. This method removes all casting from the actual implementation of visitor methods and also, with strict usage in all methods, ensures that we never perform any unwanted cast. This is done thanks to the use of constraints [20], which limits the applications of this method only to types where we have the wanted inheritance relation.

```
template <typename Base, std::derived_from<Base> T, class... Args>
Base* retPtr(Args&&... args)
{
    return new T(std::forward<Args>(args)...);
}

// ...
std::any foo() {
    // same as return (A*) new B();
    return retPtr<A, B>();
}
// ...
```

Code 3.18: `retPtr` implementation and usage

```
template <typename T>
std::unique_ptr<T> castToUnique(
                        const std::any&,
                        const std::source_location) const;

template <class Type, class OutVectorType, class InVectorType>
void fillList(
    std::vector<OutVectorType>& outVector,
    const std::vector<InVectorType>& inVector,
    const std::source_location location = std::source_location::current()
    )
{
    for (auto& ctx : inVector) {
        auto any = visit(ctx);
        try {
            // If OutVectorType is std::shared_ptr<Type>,
            // then std::unique_ptr is promoted.
            outVector.emplace_back(castToUnique<Type>(any, location));
        } catch (const std::exception& ex) {
            throw;
        }
    }
}
```

Code 3.19: `fillList` implementation

Several other methods were introduced to the visitor class, which address the issue of using `std::any`. The most complicated one of them is `fillList`, this method is shown in Code 3.19. This method takes as an input iterable object. This object is then iterated through, and all values are visited and cast into the required base type. In the implementation, we can also see the usage of method `castToUnique`, which extracts raw pointer from the `std::any` container and then wraps it into `std::unique_ptr` object. The `fillList` is

```
std::any AltVisitor::visitParse(AltCliParser::ParseContext* ctx)
{
    ext::vector<std::unique_ptr<Command>> commands;

    if (ctx->top_level_command().empty()) {
        commands.emplace_back(new EOTCommand());
    } else {
        fillList<Command>(commands, ctx->top_level_command());
    }

    return new CommandList(std::move(commands));
}
```

Code 3.20: `visitParse` implementation

```
std::any AltVisitor::visitIfCommand(AltCliParser::IfCommandContext* ctx)
{
    auto expression = castToUnique<Expression>(visit(ctx->condition));
    auto thenBranch = castToUnique<Command>(visit(ctx->then_branch));

    auto elseBranch = std::unique_ptr<Command>(nullptr);
    if (ctx->else_branch != nullptr) {
        elseBranch = castToUnique<Command>(visit(ctx->else_branch));
    }

    return retPtr<Command, IfCommand>(std::move(expression),
                std::move(thenBranch), std::move(elseBranch));
}
```

Code 3.21: `visitIfCommand` implementation

useful when working with rules or tokens that can be present multiple times in the rule which is being visited. An example can be `parse` from Code 3.9. We can see that this rule always matches one or more `top_level_command` rules, and for this reason, we will have in available `std::vector` of contexts representing `top_level_command`. We could have iterated over it in each method manually, making the code more complex, so we can use the `fillList` method to handle this. The final implementation for visiting `parse` rule can be seen in Code 3.20. We can see that in the visit method, we also handle an edge case where we do not have any commands in the input. This method does not contain any other logic and is simple as it can be, which was the target for all visitor methods, where we do not want to handle any logic related to parsing or the evaluation of the input.

```
std::map<std::string, Operators::BinaryOperators>
AltVisitor::binaryExpressionMap = {
    {"&&", Operators::BinaryOperators::LOGICAL_AND},
    {"||", Operators::BinaryOperators::LOGICAL_OR},
    ...,
    {">=", Operators::BinaryOperators::MORE_OR_EQUAL},
    {"=", Operators::BinaryOperators::ASSIGN}};


std::any
AltVisitor::visitBinaryExpression(AltCliParser::BinaryExpressionContext* ctx)
{
    auto lhs = castToUnique<Expression>(visit(ctx->lhs));
    auto rhs = castToUnique<Expression>(visit(ctx->rhs));
    auto operation = binaryExpressionMap[ctx->op->getText()];

    return retPtr<Expression, BinaryExpression>(operation,
                std::move(lhs), std::move(rhs));
}
```

Code 3.22: `visitBinaryExpression` implementation

In Section 3.1.2, we have shown that all binary expressions have the same structure. With this, the implementation of the visit method for them is straightforward, and we do not have to deal with each of the possible expressions individually. The implementation can be found in Code 3.22. Unary expressions are implemented in the same fashion.

The final example for visitor methods can be found in Code 3.21. This method is responsible for constructing `IfCommand` node in the Abstract Syntax Tree. This method represents how most methods in the `AltVisitor` are implemented. At the beginning of the method, we evaluate all mandatory rules for the rule by calling `visit` method and then wrapping the result in `std::unique_ptr` with the method `castToUnique`. After that, we check if the optional rule `elseBranch` is present; if so, we also visit it and then wrap it in `std::unique_ptr`. Finally, we construct the `IfCommand` object through `retPtr` method, which ensures that we wrap in the `std::any` pointer to the `Command` class.

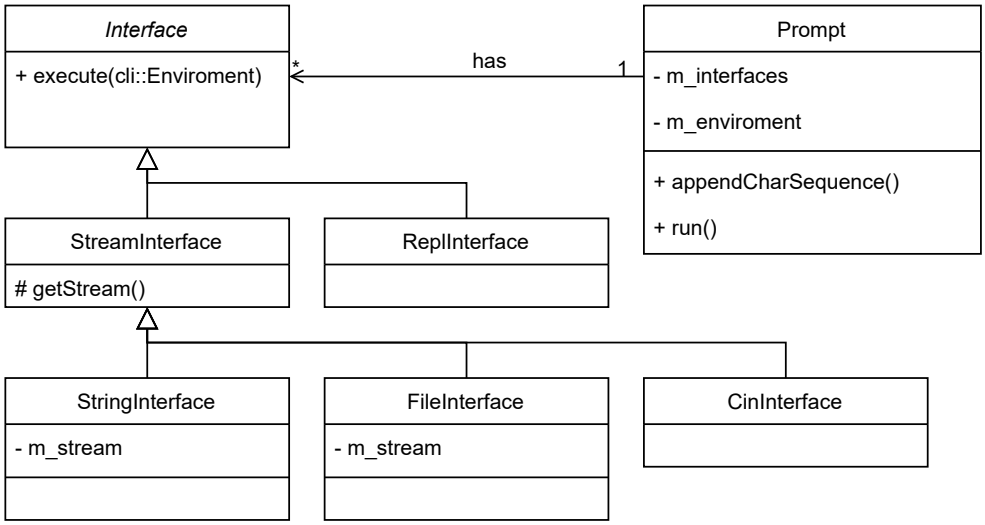

Figure 3.2: Interface class hierarchy

3.2.4 Replacing the old parser

The visitor pattern itself is not enough to replace the old parser. At first, we need to call the ANTLR4-generated parser and then pass the created syntax tree to the visitor instance, and only after that do we have an output equivalent to the original parser's output. This whole logic can be found in the file `Parser.cpp` in the `alib2cli` module. Source codes will not be shown here as it is a standard implementation and usage of ANTLR4.

One of the biggest issues when replacing the parser was how to pass the input into the parser. In Figure 2.6, we could see that it was required to construct `ReadlineInterface` before we could start the parsing process. This is, however, not required with an ANTLR4-based parser, and in fact, it is not possible since ANTLR4-based parsers accept only strings or C++ streams (which are then internally converted to strings too [21]). Replacement of the input method into the parser was in all modules straightforward except the `aql2` one, which was expected since this module has relied the most on the communication between the input interface and parser.

In `aql2` module is `Prompt` class responsible for handling all possible input sources and presenting them to the environment and parser for evaluation. This class heavily relied on the `ReadlineInterface`. This dependency was removed and was replaced with a new `Interface` structure, which is described

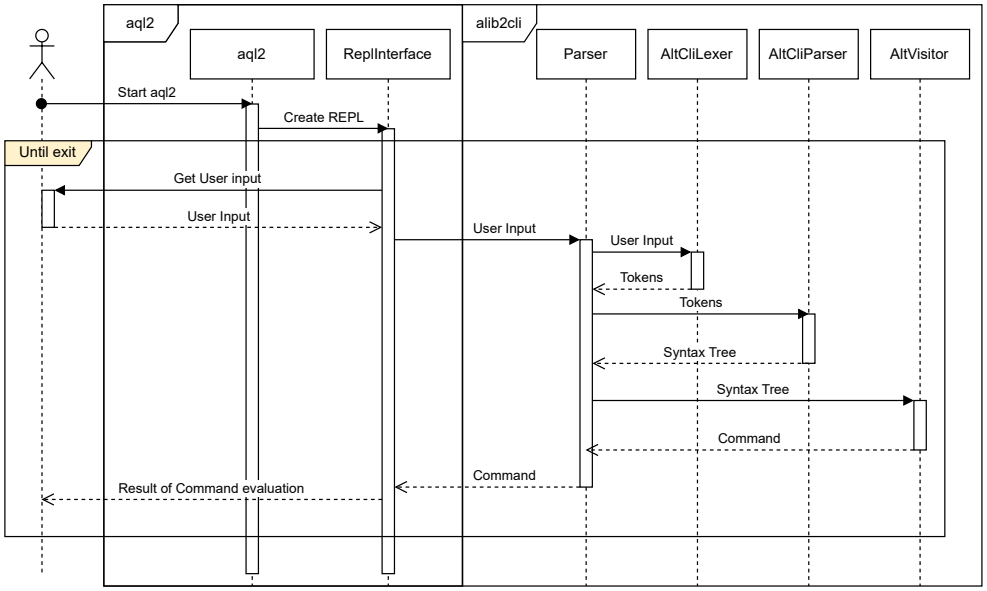

Figure 3.3: New CLI sequence diagram

in Figure 3.2. With this new class hierarchy, we still have the same flexibility and power to handle all possible input sources, which can be defined by passing arguments to the executable of the `aql2` executable. This structure makes adding a new interface without any other impact on the codebase possible.

With this, the basic integration of the parser into `aql2` was done. Thanks to removing all communication between the parser and input interface, the new sequence diagram is more streamlined, as shown in Figure 3.3.

3.3 New Features

This section will describe new features introduced into the Algorithms Library Toolkit, which were not directly related to the replacement of the parser but made the replacement easier or were possible only because of the replacement of the parser.

3.3.1 Introspect AST command

Adding a new command into the *alib2cli* module was one of the first tasks done in this thesis. There was no way to diagnose what kind of AST the

```
command
    : ...
    | KW_INTROSPECT introspect_command # Introspect
    | ...
    ;
introspect_command
    : ...
    | KW_AST top_level_command # IntrospectAst
    ;
```

Code 3.23: Introspect AST command definition

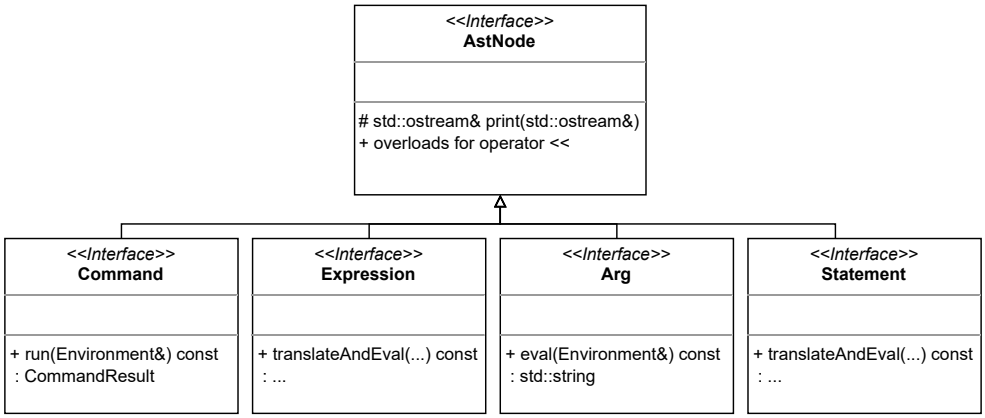

Figure 3.4: Base classes of ALT AST with print

parser produces other than manually stepping through the parser or the final AST, which is not convenient and is time-consuming.

For this reason, command `introspect ast` was created. This command was added to both the new and old parser. This command takes as input any top-level command and, on evaluation, produces a textual representation of the command into standard output. The definition of this command in the new grammar is shown in Code 3.23. Adding one common method to all possible nodes in the AST of `alib2cli` was necessary. This method prints itself on the stream passed as an argument. AST's new base class hierarchy can be seen in Figure 3.4. An example of the usage with the output can be seen in Code 3.24.

```
aql> introspect ast print expression IsSame(1, "1")
(PrintCommand
    (FunctionCallExpression
        IsSame
        (ImmediateExpression 1)
        (ImmediateExpression 1)
    )
)
```

Code 3.24: Introspect AST command usage. Output was manually formatted.

```
aql> introspect ast print expression $a.Is<tab>
IsLanguageEmpty       IsSame          IsLanguageGeneratingEpsilon
IsSymmetric           IsReflexive     IsTransitive
aql> introspect ast print expression $a.IsS<tab>
IsSame       IsSymmetric
aql> introspect ast print expression $a.IsSame(2)<enter>
(PrintCommand (MethodCallExpression (VariableExpression
(ImmediateArg a)) IsSame (ImmediateExpression 2)))
```

Code 3.25: Example usage of Replxx without color highlighting

3.3.2 Read Evaluate Print Loop ++

Read Evaluate Print Loop ++ (replxx) is a lightweight, open-source, and cross-platform readline-like library [22] for C++. It is designed to facilitate the implementation of a user-friendly and feature-rich command-line interface (CLI) in C++ applications. Replxx offers numerous features to enhance the user experience, such as syntax highlighting, autocompletion, and history management [25].

With the rework of the input processing method, which was mentioned in Section 3.2.4, it was necessary to modify how the REPL itself will work. One option was to use the already present `libreadline` library, which is shipped with GNU/Linux, FreeBSD, and MacOS [22]. Ultimately, this option was not chosen due to the old application programming interface of the library and compatibility issues with MacOS, where an old version of this library is distributed. Note that in the example, we use `<tab>` to denote that user has pressed the tab key, and then we show the output visible to the user. The same notation applies to the `<enter>` symbol. As part of this thesis, only the core

functionality of this library is used, with basic support for text highlighting, history, and autocompletion of the user input. An example of this can be seen in Code 3.25.

3.3.3 Code Completion in Console Line Interface

Previous autocomplete support in *aql2* was done through analysis of the already written input. This logic was separated from the parser logic, and if the input language of the CLI was changed, then the autocomplete had to be updated.

Thanks to the ANTLR4-based parser, we could replace this logic with code completion based on the current status of the parser. This would mean that any changes to the ANTLR4 grammar would be directly propagated to the code completion hints shown in the CLI.

As the code completion engine, we are using ANTLR C3 engine [23]. ANTLR C3, also known as ANTLR Code Completion Core, is an open-source library that can provide code completion for any ANTLR4-based parser. This library provides runtime targets only for Typescript with ports to `C#` and Java. However, the author of this library has published in the MySQL Workbench repository a port of this library into C `++`. This port can be found at GitHub [26].

The documentation of ANTLR C3 suggests refactoring the grammar for better code completion suggestions. The first suggested thing is not to skip any tokens. This will help with the easier determination of the caret position in the input text. The second refactoring which was performed was to replace all occurrences of token rule `IDENTIFIER` with rules that suggest what kind of identifier we want to suggest to the user in the current rule.

See Code 3.26 for an example of refactoring. In the example, we have rule `suffix_expression`, and it is a subrule describing the syntax of method calls. Methods in AQL are all algorithms where is the first parameter in the algorithm replaced by the `atom` on which was the method called. In the original grammar definition, we used to `IDENTIFIER` to define the method name. This is correct and works well. But the autocomplete engine does not know what identifiers we want to autocomplete. With the refactoring performed in the

```
suffix_expression
- | atom DOT IDENTIFIER bracketed_expression_list
+ : atom DOT algorithm bracketed_expression_list

+algorithm
+ : identifier
+ ;
```

Code 3.26: Example of rule refactoring for code completion

example, we specify in the grammar itself that we want to have an algorithm name here (which is an identifier). This not only provides better autocomplete suggestions but also makes the grammar more verbose since now the reader does have an understanding of what kind of values he can expect here.

Chapter 4

Testing

In the last chapter of this bachelor thesis, we will discuss how was the whole implementation tested.

4.1 Catch2

All of the unit tests in the Algorithms Library Toolkit are written with `Catch2`. Catch2 is an open-source test framework written in C++. This framework is header files only. This means that we do not have to link against any library but only include header files, but it comes at the cost of increased built time since the header files must be processed every time.

One of the key features of Catch2 is its simple and natural syntax for defining test cases and test sections. Using the BDD (Behavior Driven Development) style, Catch2 enables developers to express test scenarios in a human-readable format, making it easy to understand the intent and purpose of each test. In Code 4.1, we can see a sample test written in Catch2.

4.2 State of tests

This section will be described the state of the relevant test before this thesis. Each module has a defined `test-src` folder, in which we can find all tests for the module. Not all modules have tests, but the `alib2cli` as one of the core modules has some coverage. In Table 4.1, we can see some statistics about the coverage in the `alib2cli` module. In the table are filtered out only portions

```
#include <catch2/catch_test_macros.hpp>
unsigned int add(int a, int b ) { return a + b; }
TEST_CASE( "Add function", "[add]" ) {
    REQUIRE(add(1, 0) == 1);
    REQUIRE(add(1, 1) == 2);
    REQUIRE(add(-5, 5) == 0);
}
```

Code 4.1: Example of Catch2 test

Table 4.1: Coverage in `alib2cli` tests before

File	**Lines**		**Branches**	
Lexer.cpp	55.6%	263/473	42.9%	411/958
Lexer.h	10.1%	11/109	8.3%	5/60
Parser.h	90.6%	58/64	44.2%	72/163
Parser.cpp	51.5%	301/585	34.0%	316/930
ast folder	31.8%	223/701	19.1%	231/1208
Total	**44.8%**	866/1932	**31.1%**	1035/3319

of the module that are interesting for this thesis. Parser combined itself has around 55% lines coverage and 35.4% branch coverage. There is no systematic testing of all methods in the parser, and only a small amount of negative tests.

Negative tests are these tests where we intentionally supply invalid input to the method or part of the code we are testing and then expect some behavior from them[24].

There are other tests that test the whole application, and we could classify them as integration tests, although their primary target is to verify the correctness of implemented algorithms. These test suites are labeled as `test_cppaqltests` and `test_aqltests`. Both of them evaluate the input in the same way, and the only difference is in how the input is prepared for the tests. In the case of `test_aqltests`, it is done through prepared scripts in the CLI language. On the other hand, in the case of the `test_cppaqltests`, the input is dynamically constructed at runtime to produce valid input for the parser. Each of these tests uses algorithms that are already implemented in the Algorithms Library Toolkit, and due to this, they can take a non-trivial amount of time to finish (at the time of writing this thesis, the longest running test case took around 150 seconds on average to finish). To speed up tests

and prevent potential issues with the tests not finishing due to the timeouts, each of the test cases is run in its own fork of the application. Note that the test coverage produced by these tests is not included in the Table 4.1.

4.3 New Parser tests

Before the replacement of the old parser, there was a time period when both parsers were present in the codebase at the same time. With this, both of the parsers could be run against the same suite of tests, and the output could be compared to ensure that the same output was produced. Input for these new tests was a set of newly created scripts in the CLI language. In addition to newly created scripts, the new tests also use already written scripts from `test_aqltests` integration test. These new tests do not execute the input scripts, so algorithm names do not restrict them but only by the syntax of the language, which is what we want from these tests.

Comparing the result of two parsers is not straightforward since they produce an Abstract Syntax Tree, which in the current implementation of `alib2cli` module has no comparison operators. One approach to solve this was to implement an equality operator on the tree, which will have several limitations on usage. For the comparison, we need two Abstract Syntax Trees in memory, which we want to compare. With two parsers in the codebase, the comparison is possible and makes sense, but in the final iteration of this work, there will be only one parser; thus, this test will be impossible to execute. The second, not-so-important limitation is that it will be only usable in the tests or somewhere else in the codebase. With these facts, the indirect method of comparison is used. This is done through the already written functionality of the `alib2cli` AST, which enables the AST to print itself into a C++ stream object (for details, see Section 3.3.1). The flow of these tests is as follows:

1. Check that the script file exists
2. Ensure that the old and new parsers parse the input
3. Print result of these tests into separate `std::ostringstream`
4. Compare the content of the streams

Table 4.2: Coverage in `alib2cli` tests after

File	**Lines**		**Branches**	
AltVisitor.h	50.0%	9/18	28.8%	17/59
AltVisitor.cpp	92.3%	36/39	52.7%	29/55
AltVisitor.Arg.cpp	100.0%	24/24	91.9%	34/37
AltVisitor.Command.cpp	100.0%	229/229	94.9%	350/369
AltVisitor.Expression.cpp	100.0%	47/47	100.0%	73/73
AltVisitor.Option.cpp	100.0%	10/10	93.8%	15/16
AltVisitor.Statement.cpp	100.0%	60/60	100.0%	98/98
Parser.cpp	96.2%	25/26	85.7%	30/35
ast folder	55.8%	387/694	22.5%	215/952
Total	**72.1%**	827/1147	**50.8%**	861/1694

If all of these test points were successful, then the parsers are producing equivalent output for the given script. The flow described above is the same for all prepared input scripts, with the exception of a few tests, which do not compare the content of the streams.

Both of the parsers produce nearly equivalent AST, and if they were used in the application, they would, after some time, produce the same result. The difference here is how they handle multiple top-level commands. The old parser only takes the first top-level command and leaves the rest of the input untouched; due to this, it always returns `cli::CommandList`, which contains only one command. This can be done thanks to the fact that there is communication between the input interface and the parser. On the other hand, the new parser needs to have the input buffered so it accepts the whole input at once and produces `cli::CommandList` with potentially multiple commands in it.

This difference can not be spotted by a user who only uses REPL. REPL always tries to evaluate the input after every new line, which is the same in both the old and the new parser. The difference, however, can be spotted when executing files and only if the file contains a syntax error in the second or later top-level command. The old parser would execute all commands until the syntax error. The new parser will not execute anything.

The code in `alib2cli` module currently has 72.1% lines coverage and 50.8% branches coverage in all relevant files for this thesis. The breakdown of this coverage can be seen in Table 4.2.

After the parser was finally removed, the test methodology had to be changed since we could not compare the two parsers directly anymore. Before the removal of the old parser from the codebase, we took a snapshot of all the Abstract Syntax Tree produced by the parser in the tests and saved it next to the possible input. These files are then used as a reference for comparison in the test case against the new parser. Note that this was done in the final part of the work. The new parser was already stable, and it passed all tests. It was only kept as insurance in case a new change made the parser produce invalid or different trees.

Conclusion

The main goal of this thesis was to replace the old parser in the Algorithm Library Toolkit with a new one generated by ANother Tool for Language Recognition 4 (ANTLR4).

The first part of the thesis introduces the theory of formal languages. In that section are described terms such as the definition of grammar, context-free grammar, and ambiguous grammar. After introducing formal terms, the thesis describes what a lexer and a parser are and what part they take in the process of parsing input.

Following that, the thesis describes what ANother Tool for Language Recognition is. In this section of the analysis, the reader is introduced to the *Adaptive LL(*)* algorithm, which is used in the ANTLR4 runtime. Following that, the thesis describes how users can construct grammar for ANTLR4 and introduce the reader to the grammar actions, semantic predicates, and other features of the ANTLR4. The last section of the analysis describes the Algorithm Library Toolkit, its core modules, and the basics of the CLI language used in ALT.

At the beginning of the implementation chapter, the thesis describes how the ALT grammar for ANTLR4 was prepared and the main issues when creating the grammar. With the grammar prepared user is then introduced to the integration of the lexer and the parser generated by ANTLR4 from this grammar. The final part of the thesis describes how was the new parser tested.

In conclusion, all goals of the thesis were completed and exceeded. The initial goal was to replace the old parser with the new one generated by the ANTLR4; this was, without a doubt, done. After that, the library responsible for handling user input was replaced with a new one called *replxx*. And finally, the auto-complete for the CLI was replaced with a new one, which uses ANTLR4 grammar and current input to suggest options for an auto-complete.

This thesis could be followed by extending the syntax of the current CLI language. Among these extensions could be support for *switch* command[8], ternary expression, and more. Additional improvements to the user experience of the CLI environment could be made, starting from improvements to the auto-complete engine as far as syntax highlighting based on output from the parser.

[8]In most languages, we would classify *switch* as a statement, but in the context of ALT, it makes sense to address it as a command.

Bibliography

1. PARR, Terence. *ANother Tool for Language Recognition* [online]. ANTLR, 2013. [visited on 2023-04-02]. Available from: `https://antlr.org`.
2. HOPCROFT, John; ULLMAN, Jeffrey D. *Introduction to automata theory, languages, and computation*. CNIB, 1995.
3. HARWELL, Sam; CONTRIBUTORS, other. *ANTLR grammars V4 - c language*. GitHub, 2022. Available also from: `https://github.com/antlr/grammars-v4/blob/512d11783af25edffb4/c/C.g4`.
4. AHO, Alfred V; LAM, Monica S; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers : principles, techniques, & tools*. 2nd ed. Braille Jymico Inc, 2015.
5. WAGNER, Bill. *Get started with syntax analysis (Roslyn APIs)*. Microsoft, 2021. Available also from: `https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis`.
6. MYERS, Joseph. *New_C_Parser*. GCC Wiki, 2006. Available also from: `https://gcc.gnu.org/wiki/New_C_Parser`.
7. LLVM. *Clang - features and goals*. LLVM, 2023. Available also from: `https://clang.llvm.org/features.html`.
8. PARR, Terence. *The definitive ANTLR 4 reference*. 2nd ed. Pragmatic Bookshelf, 2013. Available also from: `https://www.`

safaribooksonline . com / library / view / the - definitive - antlr/9781941222621/.

9. PARR, Terence; HARWELL, Sam; FISHER, Kathleen. Adaptive LL(*) parsing: The power of dynamic analysis. *SIGPLAN Not.* 2014, vol. 49, no. 10, pp. 579–598. Available from DOI: 10.1145/2714064.2660202.

10. STANDARDIZATION, International Organization for. *ISO/IEC 14977:1996(en) Information Technology – Syntactic Metalanguage – Extended BNF* [online]. International Organization for Standardization, 1996. [visited on 2023-05-08]. No. 14977. Available from: https://www.iso.org/standard/26153.html.

11. HARWELL, Sam. *Force semantic error (failed predicate) in ANTLR4* [online]. StackOverflow, 2013. [visited on 2023-03-28]. Available from: https://stackoverflow.com/a/19145292.

12. TRÁVNÍČEK, Jan. *Query Language* [online]. Ed. by PECKA, Tomáš. Algorithms Library Toolkit, 2023. [visited on 2023-04-17]. Available from: https://alt.fit.cvut.cz/docs/userguide/.

13. TRÁVNÍČEK, Jan; PECKA, Tomáš. *Algorithms Library Toolkit*. Algorithms Library Toolkit, 2020. Available also from: https://alt.fit.cvut.cz/.

14. TRÁVNÍČEK, Jan; PECKA, Tomáš; PLACHÝ, Štěpán. *Algorithms Library Toolkit / Algorithms Library Toolkit Core · GitLab* [online]. GitLab, 2013. [visited on 2023-05-07]. Available from: https://gitlab.fit.cvut.cz/algorithms-library-toolkit/automata-library.

15. CPPREFERENCE.COM. *Identifiers - cppreference.com* [online]. en.cppreference.com, 2023. [visited on 2023-05-07]. Available from: https://en.cppreference.com/w/cpp/language/identifiers%5C#Qualified_identifiers.

16. MARTIN, Ken; HOFFMAN, Bill. *Mastering CMake*. .13rd ed. Kitware, Inc., 2015.

17. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

18. CUBBI. *std::any - cppreference.com* [online]. en.cppreference.com, 2016. [visited on 2023-04-28]. Available from: `https://en.cppreference.com/w/cpp/utility/any`.

19. BOCCARA, Jonathan. *How std::any Works* [online]. Fluent C++, 2021. [visited on 2023-04-28]. Available from: `https://www.fluentcpp.com/2021/02/05/how-stdany-works/`.

20. SONG, Tim. *Constraints and concepts (since C++20) - cppreference.com* [online]. en.cppreference.com, 2018. [visited on 2023-04-28]. Available from: `https://en.cppreference.com/w/cpp/language/constraints`.

21. ANTLR. *ANTLR v4 CPP runtime - ANTLRInputStream* [online]. GitHub, 2021. [visited on 2023-04-28]. Available from: `https://github.com/antlr/antlr4/blob/8dcc6526cfb154d68849/runtime/Cpp/runtime/src/ANTLRInputStream.cpp%5C#L61`.

22. RAMEY, Chet. *The GNU Readline Library*. tiswww.case.edu, 2022. Available also from: `https://tiswww.case.edu/php/chet/readline/rltop.html`.

23. LISCHKE, Mike. *ANTLR-c3 - ANTLR4 Code Completion Core* [online]. GitHub, 2023. [visited on 2023-05-01]. Available from: `https://github.com/mike-lischke/antlr4-c3`.

24. SOFTWARE, SmartBear. *Negative Testing* [online]. smartbear.com, 2020. [visited on 2023-04-28]. Available from: `https://smartbear.com/learn/automated-testing/negative-testing/`.

25. KONARSKI, Marcin. *Read Evaluate Print Loop ++* [online]. GitHub, 2023. [visited on 2023-05-08]. Available from: `https://github.com/AmokHuginnsson/replxx`.

26. LISCHKE, Mike; CORPORATION, Oracle. *mysql / mysql-workbench* [online]. GitHub, 2016. [visited on 2023-04-08]. Available from: `https://github.com/mysql/mysql-workbench/tree/8.0/library/parsers/code-completion`.

Appendix A

Acronyms

ANTLR4 ANother Tool for Language Recognition 4

ALT Algorithm library toolkit

ALL(*) Adaptive LL(*)

AQL Algorithm Query Language

AST Abstract Syntax Tree

EBNF Extended Backus-Naur Form

REPL Read Evaluate Print Loop

Appendix B

Contents of enclosed medium

This work was merged into the repository of Algorithms Library Toolkit [14] with commit `35ce5qa0e87`.

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