

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Informatik

Bringing Row Pattern Matching to DuckDB: Parsing and Frontend

Marcel Knüdeler

11.03.2025

Prüfer

Prof. Dr. Torsten Grust

Betreuerin

Louisa Lambrecht

Marcel Knödeler:

Bringing Row Pattern Matching to DuckDB: Parsing and Frontend

Bachelorthesis Informatik

Eberhard Karls Universität

01.09.2024 - 11.03.2025

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Des Weiteren erkläre ich, dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Tübingen, 17.3.2025
Ort, Datum


Marcel Knüdeler

Abstract

Row pattern matching in the form of `MATCH_RECOGNIZE`, a technique that allows finding sequences of rows matching a regular expression-defined pattern, is currently not supported by a number of database management systems (DBMS), such as PostgreSQL and DuckDB. This thesis is part of a research project focused on developing a transpiler that converts SQL containing `MATCH_RECOGNIZE` into standard SQL. In this context, we address two main parts: The parsing of `MATCH_RECOGNIZE` clauses and the creation of a web front end for demonstrating the transpiler.

For the parsing, we implement a workflow that uses the ANTLR parser generator, finding it to be a suitable solution for our purpose and greatly reducing the implementation effort. After analyzing the syntax of `MATCH_RECOGNIZE`, we cover the creation of a grammar and the adaptation of a visitor, generated by ANTLR, for the construction of an abstract syntax tree following a specific class structure. Through a test setup involving syntax-exhaustive example queries, we validate that our workflow is functioning as intended and effectively parses `MATCH_RECOGNIZE` clauses, providing a basis for further transpilation.

As a frontend solution, we modify the existing web application *Compiler Explorer*. We cover the integration of the `MATCH_RECOGNIZE` transpiler into the backend and setting up a mechanism for passing pre-executed SQL statements. We add functionality for displaying intermediate representations as an image. To be able to directly execute transpiled statements, we integrate a WebAssembly version of the DuckDB DBMS into the frontend, including a pane containing the web shell of DuckDB. Special attention is hereby given to handling technical challenges concerning query submission and asynchronous behavior. Finally, the application is containerized using Docker. We find that the chosen workflow is a stable and comfortable solution for demonstrating the `MATCH_RECOGNIZE` transpiler, but requires implementation effort due to *Compiler Explorer*'s extensive code base and complex technology stack.

Contents

Selbständigkeitserklärung	iv
Abstract	v
Acronyms	ix
1. Introduction and Background	1
1.1. Pattern Matching and Why it is needed	1
1.2. Syntax and Semantics: A Usage Example	1
1.3. Thesis Objectives	3
1.3.1. The Bigger Picture: Creation of a Transpiler	3
1.3.2. This Thesis: Parser and Frontend	4
1.4. Code	4
2. Parsing	5
2.1. What is Parsing?	5
2.2. Parsing SQL: Existing Solutions	5
2.3. Grammars	6
2.4. Our Parsing Needs	7
2.4.1. The MATCH-RECOGNIZE AST Interface	8
2.5. Parser Generators	10
2.6. Alternative Methods	12
2.7. ANTLR	12
3. Implementation of the Parser	14
3.1. Designing the Grammar	14
3.1.1. Handling Expressions as Strings	16
3.1.2. Parse Tree vs. Abstract Syntax Tree	18
3.2. Visitor Implementation	19
3.3. Testing and Evaluation	24
3.4. Discussion of the Parser	26
3.4.1. Limits	27
4. Frontend	28
4.1. Requirements	28
4.2. Compiler Explorer	29
4.2.1. Architecture and Design	30
4.2.2. Technology Stack	32

5. Implementation of the Frontend	33
5.1. Command Line Interface	33
5.2. Web Application	33
5.2.1. Integrating the Transpiler	34
5.2.2. Integrating an Image Viewer	35
5.2.3. Fitting a DBMS into Compiler Explorer's Frontend	35
5.2.3.1. WebAssembly	35
5.2.3.2. Integrating DuckDB Wasm	36
5.2.3.3. DuckDB Shell	37
5.3. Deployment	39
5.3.1. Docker Container Setup	39
5.4. Further Challenges	41
5.5. Discussion of the Frontend	42
6. Conclusion	44
6.1. Future Work	45
Bibliography	46
List of Figures	48
Technologies and Tools used	48
A Appendix	49
A.1 Grammar for Parsing MATCH_RECOGNIZE	49
A.2 Minimal Web Frontend Test	53

Acronyms

DBMS Database Management System
AST Abstract Syntax Tree
UML Unified Modeling Language
NFA Nondeterministic Finite Automaton
CTE Common Table Expression
CLI Command Line Interface

Introduction and Background

Nearly a decade ago, the SQL:2016 standard introduced `MATCH_RECOGNIZE`, a construct which enabled row pattern matching in SQL. Row pattern matching is a technique which allows finding sequences of rows based on a pattern defined by regular expressions. Up to now, `MATCH_RECOGNIZE` is not supported by many database management systems, including PostgreSQL and DuckDB.

This thesis contributes to the overarching research topic of making pattern matching available to DBMSs that do not natively support it, so that a wider range of users can benefit from its power and expressiveness.

1.1. Pattern Matching and Why it is needed

Pattern matching has many important use cases. In contrast to conventional queries matching rows individually against one or more predicates, pattern matching takes preceding and subsequent rows into consideration and returns groups of rows when they follow a specified pattern. This way it is possible to retrieve specific shapes, such as V-, W- or U-shapes, within the data. Hence, pattern matching is predestined for all kinds of processes driven by sequences of events, such as detecting unusual behavior in security applications, analyzing stock price development, sensor data analysis, fraud detection and generally complex event processing [1].

1.2. Syntax and Semantics: A Usage Example

For initial understanding, a brief introduction to the `MATCH_RECOGNIZE` statement shall be provided, based on a usage example.

Let's imagine the following scenario: We consider a dammed lake with an inflow of water. We are particularly interested in all intervals in which water is withdrawn and restored. We may want information on how quickly the water was drained or how long it took to refill. Each water withdrawal is a continuous, self-contained process. The data available to us consists of a table of water levels and time stamps. We are therefore looking for a V-shaped fall and rise in the water level.

Broken down, a query addressing that matter could look as follows:

```

1 SELECT *
2 FROM lake
3 MATCH_RECOGNIZE (
4     ORDER BY tstamp
5     MEASURES
6         MATCH_NUMBER() AS withdrawal_no,
7         STRT.tstamp AS start_tstamp,
8         LAST(DOWN.tstamp) AS bottom_tstamp,
9         LAST(UP.tstamp) AS end_tstamp,
10        STRT.waterlevel - LAST(DOWN.waterlevel) AS amount
11     ONE ROW PER MATCH
12     AFTER MATCH SKIP TO LAST UP
13     PATTERN (STRT DOWN+ UP+)
14     DEFINE
15         DOWN AS DOWN.waterlevel < PREV(DOWN.waterlevel),
16         UP AS UP.waterlevel > PREV(UP.waterlevel)
17 ) AS MR
18 ORDER BY MR.withdrawal_no, MR.start_tstamp;

```

Listing 1: Example `MATCH_RECOGNIZE` query.

- ❶ The `MATCH_RECOGNIZE` clause is an extension of a `Table Expression` that resides in a `FROM` clause. This means that it is wrapped into a surrounding query, which in our case is a single `SELECT` statement.
- ❷ Specifying the order in which the rows are observed, in this case by timestamp, is crucial for pattern matching to be meaningful.
- ❸ We define two row pattern variables `UP` and `DOWN` by giving a condition that decides whether the currently examined row fits the pattern variable. We use built in functions such as `PREV()` to navigate within the examined rows. `PREV()` refers to the row before the current one.
- ❹ From these pattern variables we now construct a pattern to look for, in a *regular expression*-like syntax. We use the quantifier `+` to specify that `DOWN` and `UP` must occur at least once for the pattern to be recognized. Other quantifiers like `*`, `?` or range quantifiers are also possible. `STRT` is not defined with a condition, so it matches any row, serving as a starting point for the pattern.
- ❺ We define five measures we want to be present in our resulting table:
 - `MATCH_NUMBER()` provides us with the number of the group of matched rows.
 - Three timestamps representing the beginning, the bottom and the end of the V-shape. In this section we can again make use of navigation operations such as `LAST()` to navigate within the matched rows. `LAST(UP.tstamp)` returns the `tstamp` value of the last row which matched the row pattern variable `UP`.
 - The difference between the start and the bottom of the withdrawal is calculated and returned as `amount`.

⑥ Only one line should be returned for each pattern found.

⑦ After a match, the search should start again at the last row that matched the condition of the UP variable.

Finally, we define an alias for the result set, generated by `MATCH_RECOGNIZE` and continue with the surrounding `SELECT` statement ①, specifying an order in which to return all columns and all rows from it.

1.3. Thesis Objectives

1.3.1. The Bigger Picture: Creation of a Transpiler

This thesis is one of three that are part of a current research topic at the Database Systems Group at the University of Tübingen: The creation of a so-called transpiler, an application written in Python that translates `MATCH_RECOGNIZE` statements into plain SQL code that is understood by a larger number of DBMSs, including DuckDB and PostgreSQL.

It therefore uses recursive CTEs and window functions. Recursive CTEs are part of the SQL:1999 standard and are expressed using the `WITH_RECURSIVE` construct, which has made a big difference since SQL became effectively a Turing-complete language with its introduction [2].

The basic working principle of the transpiler is shown in Figure 1. A SQL statement containing `MATCH_RECOGNIZE` is parsed and an *Abstract Syntax Tree* (AST) is constructed from it. This tree representation of the `MATCH_RECOGNIZE` statement is then converted into an equivalent statement that makes use of `WITH_RECURSIVE` instead. The outcome is a tree representation as well and is, among other techniques, accomplished by using a *Nondeterministic Finite Automaton* (NFA) for processing the regular expression pattern. The resulting AST is then retransformed into SQL code.

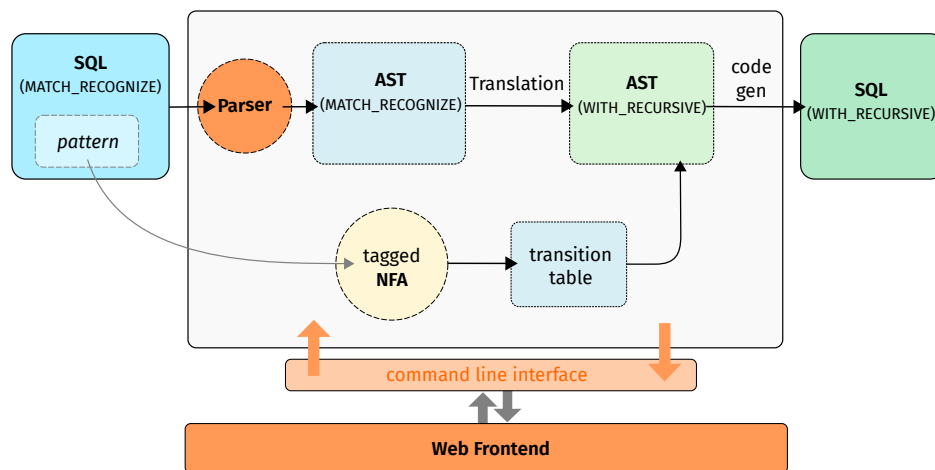


Figure 1: Structure of the transpiler, topics covered in this thesis.

1.3.2. This Thesis: Parser and Frontend

While the translation of the AST and the NFA construction are covered in other theses, this particular piece of work focuses on two aspects (emphasized in orange in [Figure 1](#))

1. The parsing of a `MATCH_RECOGNIZE` SQL statement.
2. The creation of a frontend. This includes a simple *command line interface* (CLI) and a *web application* which allows the transpiler to be used and demonstrated.

The first half of this thesis is dedicated to parsing and the implementation of the parser. After that, we make a thematic leap towards web development and move on to the creation of the frontend.

1.4. Code

The program code of the parser and CLI can be found in the **Compiler repository**¹, mainly residing in `src/parser`.

The program code of the web application is located in the **Compiler Explorer Repository**².

¹Compiler Repository: <https://db.cs.uni-tuebingen.de/> [REDACTED]

²Compiler Explorer repository: <https://db.cs.uni-tuebingen.de/> [REDACTED]

Parsing

Before we look into the actual implementation of the parser, let us now turn our attention to the term parsing, the associated methods and concepts and our parsing requirements.

2.1. What is Parsing?

In the field of computer languages, parsing can be understood as the analysis of textual information given according to the rules of a formal grammar and its transformation into a machine-processable representation [3].

This basically means, a text written by a human is being looked at. The rules according to which the text was constructed are known and it is inferred, which rules led to its syntactic construction. The result of this process is usually a tree data structure which reflects the derived construction of that text.

In our case, parsing means performing two steps: *Lexical analysis* and *syntactic analysis*.

During the lexical analysis a stream of predefined meaningful pieces, so-called tokens, is constructed from the individual characters of the input string. Keywords, identifiers, strings and numbers could be examples of tokens.

The syntactic analysis that follows takes this stream of tokens as input, derives its construction from the provided grammar, and translates it into a tree representation.

An associated task of the parser is also to check the correctness of the syntax. If no derivation is possible, errors are reported, indicating the point at which the text was not constructed according to the rules of the grammar.

2.2. Parsing SQL: Existing Solutions

There is a publicly available tool for parsing conventional SQL - the **PostgreSQL parser** [4]. It has been extracted from the PostgreSQL source code by the **libpg_query** project [5] and released as a standalone C library. Based on **libpg_query** there is another library, in this case for the **Python** language, called **pglast** [6]. It makes the abstract syntax tree constructed by the PostgreSQL parser available in Python as a tree of classes, where these classes are wrappers for the original C structures of the nodes. **pglast** also contains functionality to again generate SQL code from ASTs. Although **pglast** is not able to parse **MATCH_RECOGNIZE**, we will have use cases for it in the parsing process.

[Listing 2](#) shows a minimal example of parsing a SQL statement with **pglast**, displaying the AST as console output and converting it back to SQL code.

```

1 tree_root = parse_sql("SELECT 42;")
2 print(tree_root)
3 print(RawStream()(tree_root))

```



```

(<RawStmt stmt=<SelectStmt targetList=(<ResTarget val=<A_Const isnull=False
1 val=<Integer ival=42>>>,) groupDistinct=False
  limitOption=<LimitOption.LIMIT_OPTION_DEFAULT: 0> op=<SetOperation.SETOP_NONE: 0>
  all=False> stmt_location=0 stmt_len=9>,)
2 SELECT 42

```

Listing 2: Parsing a SQL statement in *pglast* and converting it back to SQL code.

2.3. Grammars

Before we look at an AST representation, let us clarify what is meant by grammar and how it is structured.

A grammar formally describes the syntax of a language [7]. Elementary parts of a grammar are:

- *terminal symbols*, which represent tokens, elementary symbols of the language. In this thesis the terms *terminal* and *token* are used interchangeably.
- *nonterminals*, representing sets of strings of terminals, and
- *productions rules*.

Each production rule consists of a nonterminal on the left hand side, followed by an arrow. The right hand side can contain terminals and nonterminals. A production rule describes the manner in which a particular construct of the language is written [8].

A grammar is now specified by a list of production rules. In the following example (Figure 2) which shows two rules, S denotes a nonterminal and a the terminal letter a .

$$\begin{aligned}
 S &\rightarrow a \\
 S &\rightarrow aS
 \end{aligned}$$

Figure 2: Example grammar with two production rules.

This grammar describes a string which consists of one or more occurrences of the letter a . It becomes apparent that a grammar uses recursive definitions to define sets of strings.

Because in a grammar of this form, which is also referred to as *Backus-Naur-Form* (BNF), the left-hand side of each production rule consists of only one non-terminal symbol, it is called a *context-free* grammar, which describes a context-free language [9].

2.4. Our Parsing Needs

We will now have a closer look at what exactly we have to parse and what the output shall look like. We will start by examining the syntax of a query featuring `MATCH_RECOGNIZE`, as shown in Listing 3, and spotting its important parts.

```
1  SELECT <expression> [, ...] FROM <table>
2      MATCH_RECOGNIZE (
3          [ PARTITION BY <column> [, ...] ]
4          [ ORDER BY <column> [, ...] ]
5          [ MEASURES <expression> [AS] <alias> [, ... ] ]
6          [ ONE ROW PER MATCH | ALL ROWS PER MATCH
7            [ { SHOW EMPTY MATCHES | OMIT EMPTY MATCHES | WITH UNMATCHED ROWS } ]
8          ]
9          [ AFTER MATCH SKIP
10             { PAST LAST ROW | TO NEXT ROW | TO [ { FIRST | LAST} ] <variable> }
11          ]
12          PATTERN ( <row_pattern> )
13          [ SUBSET <subset_definition> [, ...] ]
14          DEFINE <variable> AS <condition> [, ... ]
15      ) AS <alias>
16  <rest>;
```

Listing 3: Syntax of a `MATCH_RECOGNIZE` query.

First, we note that `MATCH_RECOGNIZE` is an optional subclause of the `FROM` clause, and we need an enclosing statement for the query to be complete. This *context* may be arbitrarily large. It could be a single `SELECT` statement, but `MATCH_RECOGNIZE` could also reside inside a CTE, a `VIEW` or an inline subquery, or have CTEs before it. For reasons of simplicity, it was decided to limit the permitted context in the transpiler to a single `SELECT statement`.

Nevertheless, in its trailing part, denoted as `<rest>`, it can still contain a number of constructs SQL constructs like `WHERE`, `ORDER BY`, especially combinations with other queries *e.g.* through a `UNION`. The transpiler should be able to handle complete queries, but support for attached queries introduces unwanted complexity. We therefore add another restriction, by not allowing keywords like `UNION`, `INTERSECT`, `EXCEPT` and so on.

Before we continue examining the different parts of the `MATCH_RECOGNIZE` subclause, we will introduce the class structure of the AST we want to translate our statement into, as this might affect the parsing process. It is referred to as the *interface*.

2.4.1. The MATCH-RECOGNIZE AST Interface

The interface was provided by the Database Systems Research Group and reflects the grammar given in the Oracle documentation [1].

Figure 3 shows a UML-like diagram of the interface. The individual clauses follow the same color code as in Listing 3

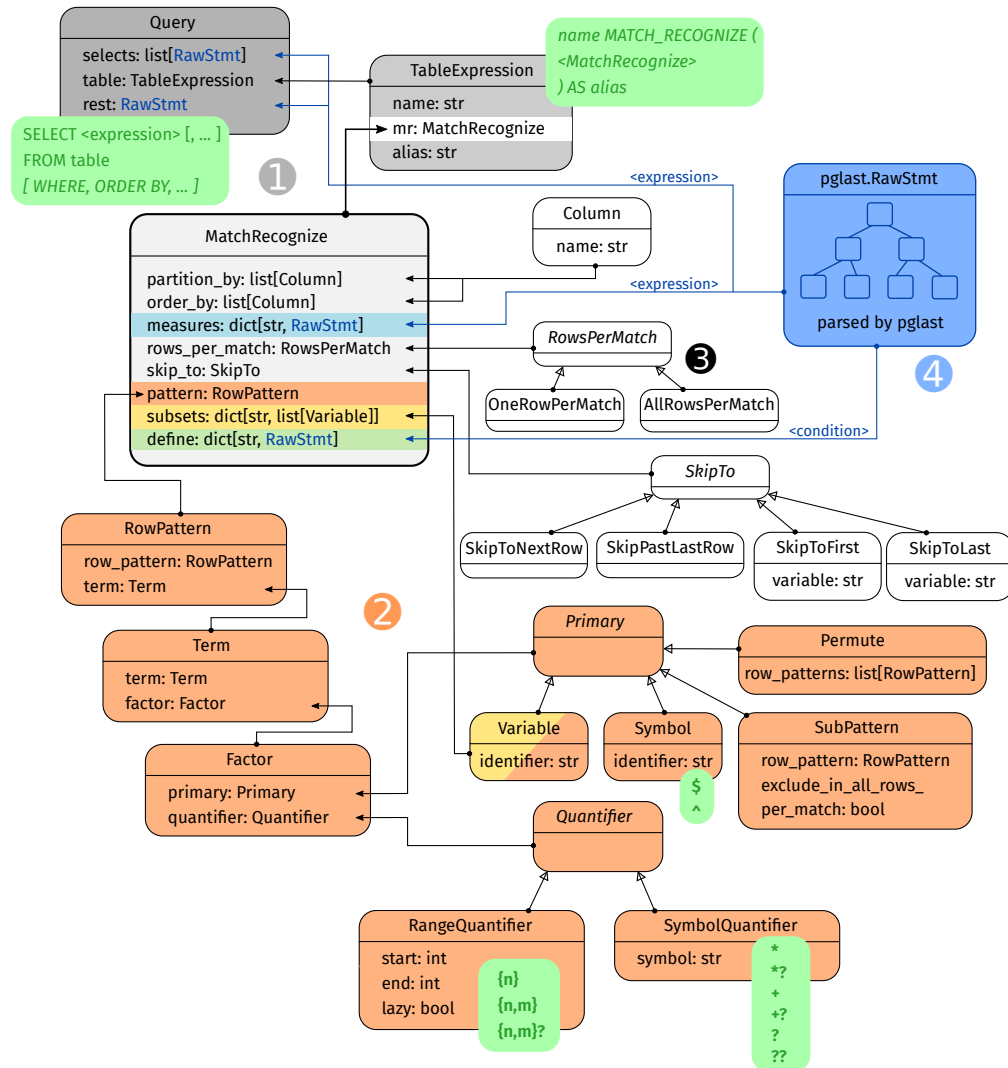


Figure 3: Class structure of the MR-AST interface

When inspecting the interface, we find four main groups of nodes: Nodes representing the basic structure ①, nodes forming the pattern ② and nodes for simple options of `MATCH_RECOGNIZE` ③. Finally, we observe a kind of object denoted by the datatype `RawStmt` ④. Behind this is the

fact that we will be using the `pglast` library (Section 2.2) to parse `<expression>` and `<condition>` parts of the statement. Therefore, these expressions need to be recognized as raw strings by our parser and then forwarded to be processed by `pglast` functions. Thus, among the nodes of self defined classes, our AST will contain subtrees consisting of `pglast` nodes.

We are particularly interested in what kinds of keywords, symbols, names can appear in the individual clauses and which of them need to be extracted as tokens because they are to be stored as data in objects or objects themselves. We want to pay special attention to the clauses `MEASURES`, `PATTERN` and `DEFINE`, color coded in Listing 3, as they are the most complex ones and consider the unmarked clauses last.

In `MEASURES` we expect expressions which are named with an alias. These expressions can contain several built in functions like the navigation operations `PREV`, `NEXT`, `FIRST`, `LAST` as well as `CLASSIFIER` and `MATCH_NUMBER`. Furthermore, they can contain standard SQL aggregates like `SUM` or `COUNT`. Those functions are called on identifiers (Row Pattern Column References), using a parentheses notation. Arithmetic with these functions and also numbers is possible. `FIRST`, `LAST` and aggregates can be combined with a preceding `RUNNING` or `FINAL` keyword.

The AST definition specifies that these expressions should be recognized as strings without further distinction and forwarded to be parsed by `pglast`. The returned subtrees are to be stored in a dictionary with their alias as key.

In `DEFINE` contains user-defined identifiers of pattern variables together with a condition expression. This condition can consist of identifiers (Row Pattern Column References), navigation functions, aggregates, parentheses, numbers as well as arithmetic and comparison operators. Again, the requirement is that they are taken “as they are”, forwarded to `pglast` and then stored in a dictionary under their alias.

With the expressions in the enclosing query, specifically each term in `SELECT` and the complete `<rest>` clause, we proceed similarly - we leave it to `pglast`.

`PATTERN` whose associated classes are highlighted in orange in Figure 3, requires us to detect row pattern variable identifiers that are declared in the `DEFINE` clause combined in a regular expression-like syntax. That means we also expect *anchor symbols* (^ and \$), *symbol quantifiers* (*, *? and +) and *range quantifiers* (in the form of {n}, {n,m}, {,m}, {n,} and {,} with an optional ?). Each possesses a corresponding AST class. Furthermore, patterns in parentheses are to be matched as `SubPattern` where we have to distinguish the `exclude in all rows per match` case, parenthesized by {- -} instead of ().

Note that the `MATCH_RECOGNIZE` statement’s `pattern` field contains the only part of the tree where nodes can be nested arbitrarily deep due to the recursive definition of some of their possible children.

In the `SUBSET` clause one may define unions of row pattern variables in the form of `XY = (X, Y)` or `SUBSET2=(A,B, C)` which will be represented as a dictionary containing lists of variables. Here, no nesting is allowed. Generally, we can’t rely on a consistent placement of spaces, so the irregularities in the second example are intended. This applies to the entire `MATCH_RECOGNIZE` statement.

`partition_by` and `order_by` are comma-separated lists of column identifiers, to be stored in string form.

`rows_per_match` and `skip_to` consist of different options, each demanding the creation of a different node.

In conclusion, our task narrows down to the question: how can we translate a textual statement as seen in [Listing 3](#) into the tree representation in [Figure 3](#) in an efficient way in terms of workload and time effort?

2.5. Parser Generators

Since we are aiming for productivity, the question arises whether there are ways to simplify the implementation of a parser. In fact, what we need, the creation of a lexer and a parser, is a process that can be automated. The kind of software addressing this issue is called a *parser generator* [7].

It takes a grammar as input and constructs the source code of a parser that understands the language's syntax defined by that particular grammar. The language the source code turns out to be in is called the *target language*, which in our case must be `Python`, since that is the language the transpiler is written in. There are a considerable number of parser generators available. Candidates that support `Python` as a target language are, among others:

- `PLY`, a `Python` implementation of the 1970 `Lex/Yacc` tools [10].
- `LARK`, described as “a modern approach, with a focus on ergonomics, performance and modularity” [11].
- `ANTLR`, a parser generator advertised as “widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees” [12].

In order to make a reasonable choice which parser generator to use in the project, some criteria considered relevant have been extracted from the documentation of `PLY` [10], `LARK` [11] and `ANTLR` [13], [14] for a comparison in [Table 1](#).

	PLY (Lex/Yacc)	LARK	ANTLR
Algorithm	LALR(1)	LALR, Earley	Adaptive LL(*)
Grammar	BNF	EBNF	EBNF
Handling of Ambiguity	Operator precedence has to be specified	"rules can be assigned a priority. When using Earley, rule priorities are used to resolve ambiguity."	"uses the order of the productions in a rule to resolve ambiguities" + semantic predicates
Left Recursion Support	yes	yes (in LALR mode)	yes
Language	Python native	Python native	Java (needs Java runtime to generate Python parser)
AST generation	no	yes (parse tree)	yes (parse tree)
Tree Visualizer	no	Dot	Java GUI (with antlr4-tools ³)
Traversing functionality	no	transformer is provided, but methods have to be written manually	generated automatically

Table 1: Comparison of Parser Generators

It is assumed, that due to the manageable size and complexity of our statement to be parsed, any of the three options would produce a satisfactory result. Nevertheless, there are two reasons that made me opt for ANTLR. The first is ANTLR's ability to provide a visual tree view using a Java GUI, which should help with debugging. The second, more crucial, is the automatic generation of a traversing tool for the parse tree. It is expected that the tree output from the parser generator will need to be transformed, so there is a use for such a construct. There also is an empirical study which shows a significantly higher performance, when using ANTLR, compared to Lex/Yacc, implying also higher simplicity, intuitiveness, and maintainability [15].

Hence, in this thesis shall be examined, if the chosen workflow, using ANTLR in combination with the constructed `MATCH_RECOGNIZE` AST interface, is suitable for parsing the `MATCH_RECOGNIZE` statement.

³antlr4-tools: <https://github.com/antlr/antlr4-tools>

2.6. Alternative Methods

An alternative to using a parser generator is to write a lexer and a parser by hand. Writing a simple lexer is generally a manageable task. For the parser, a method called *LL(1) recursive descent parsing* would be suitable for simplicity reasons. *LL(1)* stands for a reading direction from left to right, a left derivation order, and a look ahead of one token. Using the *recursive descent* method, each production rule is translated directly into a function [7], which makes it a relatively straightforward way to write a parser.

However, there are drawbacks: Since this type of parser cannot handle left recursion and only looks one token ahead, the grammar rules must be written in a less convenient way [7] as it would be the case with any of the considered parser generators. Furthermore, a change in the grammar requires a change in the code of the parser. Since the grammar of our project is evolving and is expected to change frequently, this would add to the already higher time effort, so the approach of manually writing a parser is not pursued.

2.7. ANTLR

ANTLR (another tool for language recognition) is a parser generator under active development since 1989. It is written in Java, but supports a large number of target languages. In version 4 it uses the specially developed adaptive *LL(*)* parsing algorithm, which is stated to be effective for a wide variety of applications [14].

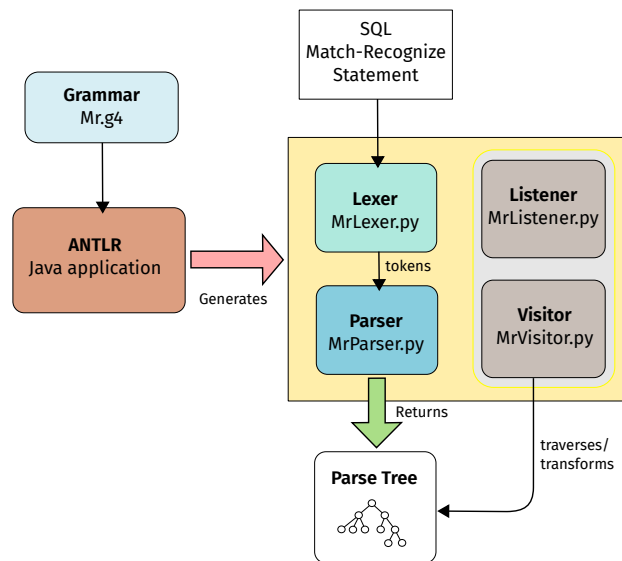


Figure 4: ANTLR scheme in the context of the thesis' use case.

In [Figure 4](#) we see a functional diagram of ANTLR, assuming the use of a grammar called *Mr.g4* and the specification of Python as target language.

ANTLR 4 accepts as input any context-free grammar that does not contain indirect left-recursion (A rule calling itself through another rule) [14].

The grammar is provided in *Extended Backus-Naur Form*, which is similar to BNF, but complemented with operators from regular expressions like `|`, `*` and `+`.

From this grammar, ANTLR generates both a lexer and a parser. Hence, the grammar can contain lexical rules as well as syntactical rules.

[Listing 4](#) shows an example ANTLR grammar, that defines the syntax of arithmetics with strings of letters and function definition.

```
1 grammar Example;
2
3 stmt: expr '=' expr ';' // parser rule 1 for nonterminal 'statement'
4     | expr ';'         // parser rule 2, also for nonterminal 'statement'
5     ;
6 expr: expr '*' expr
7     | expr '+' expr
8     | expr '(' expr ')'
9     | ID
10    ;
11
12 ID: [A-Za-z]+ ; // lexer rule 1: match ID with
13     // lower and uppercase letters
14 WS: [ \t\r\n]+ -> skip ; // lexer rule 2: ignore whitespace
```

Listing 4: A basic example of an ANTLR grammar, based on [14]

An valid string constructed according to this grammar could be: `f(x) = CoEff * x + Cons;`

By convention, parser rules come first and are lowercase, lexer rules follow and are all uppercase. Besides specifying a lexer rule for them, terminals can also be defined directly by enclosing token literals in quotes.

The working principle of the parser is as follows: From the stream of tokens, produced by the lexer, it takes in one token after the other. It then checks which rule matches by going through the rules from top (the start rule) to bottom. The first rule that matches will be used. The parser can make this decision by looking ahead a certain number of tokens. Also, by taking the first matching rule, ANTLR deals with ambiguity in the grammar and operator precedence.

This way, it is figured out how every token can be derived from the start rule (if it cannot be derived, a syntax error is reported) [8].

The data structure in which ANTLR stores this information as a result is called a *parse tree*.

ANTLR auto generates tools to traverse this tree, namely *listener* and *visitor*.

Let us now turn to the implementation of the parser, where we construct a grammar and take a closer look at how well the parse tree representation can be transformed and which of the supplied instruments is better suited for this purpose.

Implementation of the Parser

3.1. Designing the Grammar

One of the remaining challenges when using a parser generator is the creation of a grammar. Here, the syntax description of the `MATCH_RECOGNIZE` statement from Oracle [1] served as a starting point. Many of the parser rules could be adopted directly from it, as shown in [Listing 5](#), with the help of a small syntax adjustment.

```

1 row_pattern_measures ::= oracle
2   MEASURES row_pattern_measure_column[, row_pattern_measure_column]...

1 row_pattern_measures : ANTLR
2   'MEASURES' row_pattern_measure_column(',' row_pattern_measure_column)* ;

```

Listing 5: Transformation of the Oracle syntax description into ANTLR grammar.

Furthermore, some basic settings were made ([Listing 6](#)). The SQL dialect we are aiming at is case insensitive. We can specify a rule to tell ANTLR to ignore the case of letters. Then, a lexer rule is created that recognizes white space characters as tokens, but ignores them. This way, our tokens are separated at white spaces, and we avoid having to include them explicitly in every rule where they could occur.

```

1 options { caseInsensitive = true; } ANTLR
2
3 WS: [ \t\r\n]+ -> skip;

```

Listing 6: Case insensitivity and whitespace handling.

Enclosing Query

For the enclosing query, the Oracle syntax description shown in [Listing 7](#) contained clauses that were not planned to be supported for the time being. In particular, `query_table_expression`, which is not further specified but can be any construct that produces a result set such as a CTE, subquery, etc., had to be restricted.

```

1 table_reference ::= oracle
2   {only (query_table_expression) | query_table_expression }[flashback_query_clause]
3   [pivot_clause|unpivot_clause|row_pattern_recognition_clause] [t_alias]

```

Listing 7: Syntax description from Oracle for the enclosing query.

To better match the interface, two new rules ([Listing 8](#)) were introduced to replace `table_reference`:

- `query` now does not support the unwanted constructs `ONLY`, `PIVOT/UNPIVOT` and `FLASHBACK`, but allows providing additional clauses like `WHERE` and `ORDER BY` under a `rest` rule.
- `table_expression` makes a `MATCH_RECOGNIZE` clause mandatory, while not allowing CTEs or subqueries in the `table_name` rule.

```
1 query : ANTLR
2     'SELECT' expression( ',' expression)* 'FROM'
3     table_expression
4     rest?
5     ';' ? ;
6
7 table_expression :
8     table_name
9     row_pattern_recognition_clause
10    t_alias? ;
```

Listing 8: New parser rules for the enclosing query.

Tokens

To define tokens, we proceeded in two different ways. All keywords like `'MEASURES'` or `'AS'`, but also most symbols such as `'('`, `'*'`, `'<='` could be directly defined in the parser rules, by putting them into quotes. By not using aliases, grammar clarity benefited.

Tokens following more complex rules were defined as lexer rules, using regular expressions, as illustrated in [Listing 9](#).

```
1 TEXT : ([a-z_#@.]) ([a-z0-9_#@.])* ; // identifiers ANTLR
2 NUMBER : [0-9]+ ; // unsigned integer
3 SQL_STRING : '\'' ( ~'\'' | '\\\'' )* '\'' ; // strings enclosed in quotes
4 COMMENT : '--' ~[\n]* -> skip ; // ignore comments
```

Listing 9: Using lexer rules to define tokens in the `MATCH_RECOGNIZE` grammar.

3.1.1. Handling Expressions as Strings

The expressions in `MEASURES` and conditions in `DEFINE` turned out to be surprisingly challenging. As mentioned earlier, the intention was to get ANTLR to recognize them as a single raw string token, *i.e.* “as they are”, so that they could be passed on⁴ to `pglast`. Figure 5 illustrates this. Among identifiers and numbers, these terms can contain a variety of symbols, and spaces.

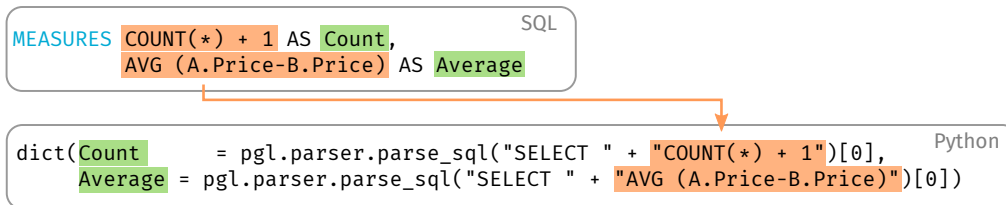


Figure 5: Storage of expressions in `MEASURES` clause.

It is not possible to just define a token as in Listing 10 which matches any character (including spaces) and make it part of the parser rule that represents a term in the `MEASURES` clause (Listing 11).

```
1  EXPR : ~[\n,;]* ; ANTLR
```

Listing 10: Lexer rule that matches anything, until new line, comma or semicolon.

```
1  row_pattern_measure_column : ANTLR
2    expression 'AS' c_alias ;
3
4  expression : EXPR
```

Listing 11: Parser rule of measures term.

This approach fails for several reasons: First, hypothetically, the resulting token would simply include `'AS'` and the following `c_alias`, since the rule allows spaces and the lexer categorizes text parts into tokens according to the rule that allows the token to have a maximum length. Second, such a rule would override all others, because of the way the lexer works. Since it is a process that precedes the parser, it is not aware of the parser rules. Hence, it does not know, that an `EXPR` token is only to be expected inside of a term derived from a `row_pattern_measure_column` parser rule. So, the first line in the statement will directly match the `EXPR` rule as a whole instead of the first word matching the `SELECT` token. This also applies to most of the other lines. Again, the reason for this is that the `EXPR` rule would allow spaces, making it the longest possible match.

A first idea to deal with this issue was to use a so-called `lexer mode`. This concept allows the lexer to use a different set of rules while being in that mode. However, a mode is entered or left when the lexer encounters a certain symbol or string. For the `MEASURES` clause, a mode for the purpose of recognizing just one consecutive raw string could have been entered on encountering

⁴Since `pglast` does not parse incomplete statements, it is necessary to prefix expressions/conditions with a `SELECT` keyword to be able to generate `pglast` nodes. The transpiler’s translation stage takes this into account.

of 'MEASURES' and turned off with ' AS ', but after passing the first line of the subclause, no suitable symbol could be found to safely switch it back on.

The problem was solved by explicitly defining every possible fragment of the MEASURE expressions, so that it could be parsed as a separate token. Bigger fragments could be consolidated by a lexer rule TEXT, but this had to be as restrictive as possible to not catch symbols that could appear in other clauses: Symbols like the *Kleene-Star* (*) for example, that could also represent a quantifier in the pattern clause had to be tokens of their own and could not be part of the regular expression defining TEXT. Otherwise, a combination like A* in the pattern clause would have been wrongly classified as a TEXT token.

Listing 12 shows the part of the grammar belonging to expression.

```
1 row_pattern_measure_column : ANTLR
2     expression 'AS' c_alias ;
3
4 expression : expression_part+ ;
5 expression_part : TEXT | NUMBER | SQL_STRING | '=' | '*' | '/' | '+' | '-' | '(' |
6     ')' ;
7 TEXT : ([a-z_#@.]) ([a-z0-9_#@.])* ; // identifiers
8 NUMBER : [0-9]+ ; // an unsigned int
9 SQL_STRING : '\\' ( ~'\\' | '\\\\' )* '\\'; // a string enclosed in quotes
```

Listing 12: Handling of the nonterminal <expression>.

We see that expression is structured with sufficient granularity, *i.e.* tokens that have a meaning in other clauses are defined separately.

The condition nonterminal in the DEFINE clause could be handled similarly, but it had to support comparison operators like <= and <> in addition.

We then received a chain of individual tokens from which the original expression had to be reproduced. Simply joining them without a delimiter caused problems, because this led to a missing separation of logical operators AND and OR that could occur in conditions:

A.price > B.price AND A.price < 25 would have resulted in A.price>B.priceAND A.price<25 leading to a missing recognition of AND when forwarded to pglast. Joining with a one space delimiter might have been an option, but it was not trusted to work in every situation. It was decided to fully rely on the SQL parsing abilities of pglast by retrieving the exact text from the original input, including ignored whitespace characters, before passing it on. From the ANTLR Runtime, it was possible, to retrieve the original character stream⁵ for the lexer. Since each token is indexed⁶, the interval containing the complete expression could be deduced from the first token's first index and the last token's last index. Therefore, a Python method was written (Listing 13), that fetches the raw text of all terminal nodes under a provided rule context node from the input stream. This method was then used during the traversal of the parse tree.

⁵GetInputStream: [https://www.antlr.org/api/java/org/antlr/v4/runtime/CommonToken.html#getInputStream\(\)](https://www.antlr.org/api/java/org/antlr/v4/runtime/CommonToken.html#getInputStream())

⁶Index: <https://www.antlr.org/api/java/org/antlr/v4/runtime/CommonToken.html#index>

```

1 def get_raw_text_with_spaces(ctx):
2     # Get the CharStream (source of characters for ANTLR lexer)
3     input_stream = ctx.start.getInputStream()
4     # Extract the exact substring of the original input using
5     # start and stop tokens of the context
6     start_index = ctx.start.start
7     stop_index = ctx.stop.stop
8     return input_stream.getText(start_index, stop_index)

```

Listing 13: A method for retrieving the exact input from tokens under a rule context node.

As a side note, SQL strings enclosed in '' could be matched as a whole because of their unambiguous separator. The corresponding lexer rule can be found in [Listing 12-9](#).

The Rest clause

For the rest clause, we just specified supported keywords, along with one or more expression ([Listing 14](#)). Note, that this rule would allow wrong SQL code, like multiple **WHERE** clauses or a wrong order. Since we pass it on to pglast, which then does a separate check for syntactic correctness, we can limit ourselves here to the specification of allowed keywords.

```

1 rest:
2     rest_part+ ;
3 rest_part:
4     ( 'WHERE' | 'GROUP BY' | 'HAVING' | 'ORDER BY' | 'FETCH FIRST' | 'LIMIT' )
5     expression( ',' expression)* ;

```

Listing 14: Handling of the rest clause.

The full grammar can be found in the Appendix ([Section A.1](#)).

3.1.2. Parse Tree vs. Abstract Syntax Tree

In this section we look at the shape of the parse tree produced by ANTLR and how it differs from our desired AST representation. We then consider what steps need to be taken to achieve the latter.

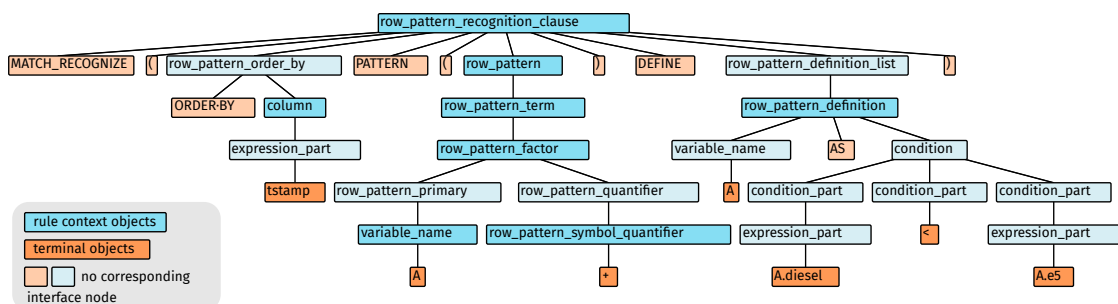


Figure 6: ANTLR parse tree (section) representing a minimal `MATCH_RECOGNIZE` subclause.

In a parse tree, like the one coming from ANTLR (Figure 6), every recognized token is represented as a leaf node. Every production rule that has been utilized in the derivation process of these tokens becomes an interior node whereby the root node is the starting rule.

Technically, the ANTLR parse tree consists of `rule context objects` for production rules and `terminal objects` for tokens.

It represents the derivation, *i.e.* the order in which productions were used to replace nonterminals, in a graphical way [8]. Hence, for each token, you can deduce which rule it was placed under, *i.e.* which syntactic construct it is part of, by moving up the tree.

In contrast to that, an *Abstract Syntax Tree* does not represent every syntactic detail of the language. Instead, it focuses on information about structure and content [16].

Accordingly, by studying our AST interface in Figure 3, we find that it does not foresee any tree nodes for keywords and parentheses. These are expressed within the structure of the tree. Furthermore, a number of rule nodes has no direct representation in the AST and information that exists as terminal objects in the parse tree is to be stored as class data, *e.g.* in lists and dictionaries.

To create an AST, we can now traverse the parse tree, and decide upon visiting a parse tree node, if a corresponding AST node needs to be created. We find the information needed to populate the fields and construct the children of that particular interface node further down the current branch.

We have to use the rule structure as a guidance, but often don't convert rule objects into interface objects. Instead, we often skip over them to gather values from the terminal objects to place them in fields of interface objects to complete their creation. Nevertheless, there are rules that have a direct equivalent in our AST nodes. For example, the pattern structure consisting of `row_pattern`, `row_pattern_term` and `row_pattern_factor` is to be reproduced relatively accurately.

ANTLR provides us with two tools for traversing the parse tree: *Listener* and *Visitor*. During first attempts, the *Listener* did not prove suitable for our needs. It traverses the tree in a depth-first-search manner, calling a rule's event method on entering and exiting a node of the type of this particular rule. Because it can't return anything from its methods, it was assumed that its usage would have required a significant number of helper objects for tracking, collecting, and decision purposes, so this approach was not pursued and the *Visitor* became the method of choice.

3.2. Visitor Implementation

In this chapter, we cover how the *Parse Tree* as an intermediate result of the parsing process is transformed into an *Abstract Syntax Tree* using a *Visitor* template provided by ANTLR.

Generally, a visitor is a design pattern in object oriented programming. Instead of having to extend multiple classes with a method for a specific functionality, a visitor object is utilized. This visitor object is then passed to every object we want this functionality to be applied to, taking the object itself as an input. The object only has to have a method for accepting the visitor. The visitor object then executes a specific method which corresponds to the type of the object it received. A visitor works together with a traverser [17].

In ANTLR the visitor pattern is expressed in such a way, that ANTLR auto generates a visitor class which provides a method for every rule defined in the grammar, *i.e.* for any rule context node of the parse tree that could possibly be visited. The visitor has a `visit()` method, with which it can pass itself to a tree node, taking it as a context (`ctx`) argument. An entry point, usually the root of the tree, has to be provided. Then the visitor can continue itself navigating the tree by using its default `visitChildren()` methods, or the order of traversal can be controlled by the user.

```
1 class MrVisitor(ParseTreeVisitor):
2
3     # Visit a parse tree produced by MrParser#query.
4     def visitQuery(self, ctx:MrParser.QueryContext):
5         return self.visitChildren(ctx)
6     ...
```

Listing 15: Visitor generated by ANTLR: Method of the first rule.

Listing 15 shows the basic skeleton of the visitor generated by ANTLR. A similar method is added for each rule, receiving a context object of the type of this particular rule (the currently visited node).

The visitor had to be subclassed because it was auto-generated and therefore was subject to change. This allowed us to override rule methods only when an action needed to be taken, and keep the visitor class uncluttered. In Listing 16 we see our own subclassed visitor and the overridden implementation of the rule method for the entry rule `query`.

```
1 class BuildAstVisitor(MrVisitor):
2
3     # Visit a parse tree produced by MrParser#query.
4     def visitQuery(self, ctx: MrParser.QueryContext):
5         # Get text from all recognized expressions and forward it to _pglast_
6         selects = [pgl.parser.parse_sql(
7             "SELECT " + get_raw_text_with_spaces(e))[0] for e in ctx.expression()]
8
9         table = self.visitTable_expression(ctx.table_expression())
10        rest = self.visitRest(ctx.rest())
11        return ast.Query(selects, table, rest)
12    ...
```

Listing 16: Subclassing of the visitor and overriding the first rule method.

In each rule method, we created and returned the corresponding interface object, using results that were obtained from the current context (Listing 16-6), or by manually continuing the visiting process on the current context object's children (Listing 16-9). Rules that should not create an interface object could be skipped by leaving them untouched. The default `visitChildren()` method took care of continuing the traversal of the subtree. This method realized a bottom-up creation of the AST, starting with the leafs.

Accessing the current rule context

Several ways were used to obtain information from the current visited node, the so-called rule context. We will describe them exemplarily using the `row_pattern_primary` node, whose corresponding grammar rule is shown in [Listing 17](#). When encountered, it required a complex distinction to decide what type of AST node to create and how to traverse the parse tree further.

```

1 row_pattern_primary : ANTLR
2     variable_name      // create ast.Variable
3     | SYMBOL           // create ast.Symbol
4     | '(' row_pattern? ')' // create ast.SubPattern
5     | '{-' row_pattern '-}' // create ast.SubPattern with exclusion flag
6     | row_pattern_permute ; // continue visitor with row_pattern_permute method

```

Listing 17: The grammar rule for a row pattern primary.

[Listing 18](#) shows the implementation of the distinction in the visitor.

```

1 def visitRow_pattern_primary(self, Python
2                               ctx:MrParser.Row_pattern_primaryContext):
3     if ctx.variable_name():
4         return ast.Variable(ctx.variable_name().getText())
5     elif ctx.getText() == "$" or ctx.getText() == "^":
6         return ast.Symbol(ctx.getText())
7     elif ctx.children[0].getText() == "(":
8         return ast.SubPattern(
9             self.visitRow_pattern(ctx.row_pattern()) if ctx.row_pattern() else
10            None
11        )
12     elif ctx.children[0].getText() == "{-":
13         return ast.SubPattern(self.visitRow_pattern(ctx.row_pattern()), True)
14     return self.visitRow_pattern_permute(ctx.row_pattern_permute())

```

Listing 18: Different ways of accessing the context in the `Row_pattern_primary` visitor method.

If a non-terminal (e.g. `variable_name` in [Listing 17-2](#)) gets picked in the derivation process, it becomes a child node of the current context and is reachable through a method of the context with the name of the picked non-terminal. Its presence could then be queried ([Listing 18-3](#)), its context accessed ([Listing 18-4](#)) or it could be passed on to another rule method, continuing the traversal ([Listing 18-12](#)). The concatenated text of all token nodes in the whole subtree below a rule context node could be accessed with the `ctx.getText()` method. There was also the possibility to access all children of the current node via a list, or a specific one by its list index ([Listing 18-7](#)).

Grammar Granularity

In some scenarios, the question came up, when to refine the grammar to do additional ANTLR parsing in contrast to handling a series of not further differentiated tokens as a string⁷ afterwards in Python. For example, when implementing the pattern's quantifiers, the AST interface foresaw a distinction between the symbol quantifier and the range quantifier, regarding the created node. Since the Oracle grammar had only one rule for both, it would have required a string comparison in Python. To avoid this, I decided to let ANTLR make the distinction, and refined the grammar by splitting the rule into two more rules (see Listing 19). Because these rules were now present as nodes in the parse tree, a decision could be made in the visitor based on the visited rule node.

```
1 row_pattern_quantifier ::= Oracle
2     *{?}
3     |+{?}
4     |?{?}
5     |"{ "[unsigned_integer ],[unsigned_integer]" }{?}
6     |"{ unsigned_integer "}"

↓

1 row_pattern_quantifier : ANTLR
2     row_pattern_symbol_quantifier
3     | row_pattern_range_quantifier ;
4
5 row_pattern_symbol_quantifier :
6     '*' '?' | '+' '?' | '?' '?' ;
7
8 row_pattern_range_quantifier :
9     ( '{' NUMBER? , 'NUMBER?' )
10    | '{' NUMBER '}' ) '?' ;
```

Listing 19: Refining the rules for quantifiers.

In other situations, it was more practical to analyze the string from the concatenated tokens in Python, like extracting start, end and lazyness values from the range quantifier string `{n,m}?`. This kept the grammar more compact.

Listing 20 illustrates the use of Python string operations while keeping the grammar coarser (and more tidy), Listing 21 demonstrates how an approach would have looked that used additional grammar rules to distinguish between the multiple range quantifier options. I followed the rule that if a different node needed to be created, the grammar would be refined; if only fields in interface nodes were involved, Python could handle it.

⁷Note: This problem is to be distinguished from the one described in Section 3.1.1. Here, it is dealt with already recognized tokens.

```

1 row_pattern_range_quantifier : ANTLR
2   ( '{' NUMBER? ',' NUMBER? }'
3   | '{' NUMBER '}' ) '?'? ;

```

```

1   # Do string operations to distinguish between {n}, {n,m}, {n,}, Python
2   # {,m}, {,}, each with optional '?' (reluctant quantifier)
3   def visitRow_pattern_range_quantifier(self,
4       ctx: MrParser.Row_pattern_range_quantifierContext):
5       text = ctx.getText()
6       lazy = True if text[-1] == "?" else False
7       nm = text.strip("{}?").split(",")
8       if len(nm) == 1: # case {n}
9           start, end = int(nm[0]), int(nm[0])
10          else: # case {n,m}
11              start = int(nm[0]) if nm[0] else 0
12              end = int(nm[1]) if nm[1] else float('inf')
13          return ast.RangeQuantifier(start, end, lazy)

```

Listing 20: Variant A: Handling range quantifier with Python string operations.

```

1 row_pattern_range_quantifier : ANTLR
2   quantifier_n | quantifier_nm ;
3
4 quantifier_n : '{' number_n '}' lazy? ;
5 quantifier_nm : '{' number_n? ',' number_m? '}' lazy? ;
6 number_n : NUMBER ;
7 number_m : NUMBER ;
8 lazy : '?' ;

```

```

1   # On this rule, just continue traversing. Python
2   # (Overriding not necessary, only shown for explanation.)
3   def visitRow_pattern_range_quantifier(self,
4       ctx: MrParser.Row_pattern_range_quantifierContext):
5       return self.visitChildren(ctx)
6
7   # Build the {n}-Quantifier when its rule context node is encountered
8   def visitQuantifier_n(self, ctx: MrParser.Quantifier_nContext):
9       end = ctx.number_n().getText()
10          return ast.RangeQuantifier(0, end, True if ctx.lazy() else False)
11
12  # Build the {n,m}-Quantifier when its rule context node is encountered
13  def visitQuantifier_nm(self, ctx: MrParser.Quantifier_nmContext):
14      start = ctx.number_n().getText() if ctx.number_n() else 0
15      end = ctx.number_m().getText() if ctx.number_m() else float('inf')
16      return ast.RangeQuantifier(start, end, True if ctx.lazy() else False)

```

Listing 21: Variant B: Parsing range quantifier by grammar refinement.

RUNNING / FINAL

One particular problem was the missing ability of pglast to parse clauses containing **RUNNING** / **FINAL** syntax. Providing a string with two identifiers in a row ([Listing 22](#)) caused a syntax error.

```
1 "FINAL LAST(UP.tstamp)"
2 "RUNNING AVG (A.Price)"
```

Listing 22: Examples of strings not parseable by pglast.

To overcome this problem, each string was preprocessed in Python, before being passing to pglast. Hereby, regular expressions were used to remove the **RUNNING** keyword as it is the default, and combine **FINAL** with its successor keyword using a connecting *unusual string*, as shown in [Listing 23](#). This allowed the string to be parsed by pglast and **FINAL** could be identified in the translation stage of the transpiler by filtering out the *unusual string*.

```
1 code_str = "_1F1PR0C1_"
2 result = re.sub(r"^( ) (FINAL\s+)(COUNT|AVG|SUM|MIN|MAX|FIRST|LAST)",
r"\1FINAL{} \3".format(code_str), input_string, flags=re.IGNORECASE)
```

Python

Listing 23: Using Python's regular expression module to merge **FINAL** with its possible successor.

The fact that the use of aggregates in **MATCH_RECOGNIZE** is restricted to only a few, they could be entirely covered in the regular expression. This made the solution quite robust, avoiding accidentally selecting other uses of the words **running** and **final** within the statement.

3.3. Testing and Evaluation

To test the parser, two types of tests were written, using the Python test framework **unittest**.

The first type was a tree equality test between the tree produced by the parser and a manually constructed tree. Here, a structural and a value comparison was performed. Due to the effort of manually creating ASTs, especially for the pattern, a second type of test was devised. This used the printing functionality of our AST interface, realized by the Python `__str__` methods, and did a string comparison between the actual printed output of a tree and the correct one.

For the first type, three test statements were designed to cover a wide range of syntactic possibilities. [Listing 24](#) shows such a statement. Since it is only for testing syntax, it does not claim to be logical. Irregularities like missing whitespaces between operators were also tested. These statements were then constructed in nodes, whereby especially the pattern tree became quite large. [Listing 25](#) shows an AST of a simple pattern.

The second type of test allowed quick addition of test statements, so there were numerous. In particular, the Oracle documentation [1] was a helpful source of test statements. However, these string comparison tests theoretically allowed for certain differences in the class structure (*e.g.* a different number of nested **Term** objects in a pattern), not admitting them to be a completely safe indicator for correctness.

The integration of these tests into a continuous integration setup enabled their mandatory execution on every pull request.

```

1 SELECT num, SUM(size) FROM t
2     MATCH_RECOGNIZE (
3         PARTITION BY num
4         ORDER BY id
5         MEASURES MATCH_NUMBER() AS feature,
6                 PREV(a.num)+5 AS start1,
7                 COUNT(*) AS totalcount
8         ONE ROW PER MATCH
9         AFTER MATCH SKIP TO LAST size
10        PATTERN (PERMUTE (a, b){7} b* ( c {- b+ -})? (a+ b{,}) a{1,2}?
11               c{,4} (a)a b{2,})
12        SUBSET STDN= (a, b)
13        DEFINE
14            a AS t.a+ 2 > t.b,
15            b AS t.a <= t.b,
16            c AS a/45.7 <> b
17        ) AS M
18    WHERE sky = 'blue'
19    HAVING SUM(size) = 4
20    ORDER BY num;

```

Listing 24: Test statement, designed to be syntactically exhaustive.

```

1 pattern = ast.RowPattern(
2     ast.Term(
3         term=ast.Term(
4             term=ast.Term(
5                 factor=ast.Factor(ast.Variable('STRT')),
6                 factor=ast.Factor(ast.Variable('DOWN'),
7                                     ast.SymbolQuantifier('+')),
8                 factor=ast.Factor(ast.Variable('UP'), ast.SymbolQuantifier('+'))))
9         )
10    )

```

Listing 25: A manually created AST of a simple pattern (STRT DOWN+ UP+).

To extend the test environment, the web frontend was used to display the parser output, simplifying manual insertion and dynamic editing of test statements.

3.4. Discussion of the Parser

With the chosen workflow, the parsing of the `MATCH_RECOGNIZE` subclause and the enclosing query, could be adequately managed within the defined limitations. It must be stated though that SQL is an extensive language with rich set of syntactic possibilities. During the creation process, non-parsable constructs were encountered on a regular basis. Although the majority was implemented on spotting, there are still known problematic constructs left and new ones are expected to be found. As an example, `substrings` in the `SELECT` clause, that were discovered in the late stage of writing had to be declared as unsupported. The parser aims to be a practical approximation, it does not claim to support full SQL.

Based on project experience, the decision to use a parser generator contributed significantly to speeding up the process because the creation of the lexer and parser was reduced to the construction of a grammar.

The grammar creation process proved to be intuitive and allowed for experimentation and refinement, since the parser could simply be re-generated after a change in the grammar. If the parser had been written by hand, a change in the grammar would have required a change in the program code of the parser leading to increased effort.

ANTLR in particular turned out to be a suitable choice, as it provided auto-generated tools. So, a traverser for the tree and a visitor did not have to be written manually. Also, on a grammar change, the visitor was updated automatically as well, requiring only a change of the methods in the subclassed part. This made the creation of the AST very efficient.

However, ANTLR requires a certain familiarization period. This might raise the question of whether, given the manageable complexity of the statement to be parsed, a manual development of a parser would have been justified, as it might have allowed for a quicker start. Based on prior experience with manually writing a simple parser⁸, I tend to find, that the advantages of using a parser generator outweigh this initial learning curve. A parser generator like ANTLR has built-in error reporting, which would otherwise have to be written first, giving a clear feedback. It has a visual representation of the parsed tree outcome, helping with debugging. Furthermore, it is expected, that the `MATCH_RECOGNIZE` interface will evolve in the future, particularly concerning the handling of parts parsed by *pglast* (Section 3.1.1). They might have to be refined further in the first parsing stage. The use of a parser generator represents a clear advantage in this case, as significantly fewer parts of the program have to be manually modified when the grammar is to be extended.

It should be noted that ANTLR is a sophisticated tool and includes more concepts, such as *lexer modes* and *semantic predicates*, whose possibilities to contribute to the result were not exhaustively investigated in this scenario but could be in future work.

⁸The Construction of an SASL-Compiler: <https://db.cs.uni-tuebingen.de/teaching/ss24/teamprojekt-sasl/>.

3.4.1. Limits

In some parts, this work had to concentrate on the correct parsing of error-free statements, not on error detection. While the syntactic correctness of the entered statement is ensured by the parsing with ANTLR and pglast, a statement can contain semantic incorrectnesses, especially inside the clauses parsed by pglast. As an example, the statement shown in Listing 26 will be parsed, but contains a number of references to undefined variables and aliases. Future work could address this issue.

```
1 SELECT price FROM t -- price is not defined in the MEASURES clause
2 MATCH_RECOGNIZE (
3   ORDER BY id
4   MEASURES
5     SUM(C.price1) AS p -- C is not defined in the DEFINE clause
6   ONE ROW PER MATCH
7   PATTERN (B) -- B is not defined in the DEFINE clause
8   DEFINE
9     A AS u.price1 > u.price2 -- u is not defined in the FROM clause
10  );
```

Listing 26: A statement containing semantic errors.

Frontend

The other major topic covered in this thesis is the creation of a frontend.

In this context, we understand frontend to mean the following two things:

- A **command line interface** (CLI), that allows the user to use the transpiler from the command line.
- A **web application**, which provides a graphical user interface (GUI) for the transpiler.

The web application will use the CLI to control the transpiler, but will also provide additional functionality to the user.

4.1. Requirements

We will first establish what kind of functionality is expected from each part of the frontend.

The CLI must be able to run the transpiler with a statement specified as a file. It must also provide the optional ability to specify a file path to write the resulting statement to. It needs to be able to take a location for a graphical representation of the NFA, to advise the program to store the SVG image created during the compilation process in that location.

The web application should provide a text box, where the statement to transpile can be inserted and edited. It must be able to execute Python to run the transpiler with the inserted statement as input, outputting the result in a second text area. On request, it should also display an image of the NFA constructed for that particular statement in the web browser.

As a special feature, the goal is to integrate a WebAssembly-compiled version of DuckDB in the form of a web shell⁹ that can be used to directly test the transpiled statement. It should also be possible to pass initiating SQL commands to this DuckDB instance to insert data to work with. The idea was, to be able to insert these commands in the same text area, just before the actual statement to transpile, separated by a delimiter.

We decided, not to build such a web application from scratch, but to adapt an existing one to our needs, with the intention of benefiting from a pre-existing infrastructure and features, thus reducing the implementation effort.

⁹DuckDB Shell: <https://shell.duckdb.org/>

4.2. Compiler Explorer

In fact, there is a web application that is well suited for our purpose due to its similar nature. It is called **Compiler Explorer**¹⁰, and is available under an open source license.¹¹ Its original purpose is to display the assembly output of various compilers for a large number of languages such as C++, Rust, Go, etc., making it possible to analyze and compare different compilers, versions and settings.

It already has an infrastructure with a server backend that can run executable files and a frontend that allows inserting and editing code, provides syntax highlighting and displays the result in a customizable window layout. It is also worth noting that Compiler Explorer supports caching of compilation results, which benefits speed, traffic and server load.

However, Compiler Explorer provides a lot of functionality, much of which is not needed for our use case, e.g. library importing, complex hovering functionality, many different views like a Control Flow Graph, LLVM intermediate representation, etc. Furthermore, supporting a multitude of languages and compilers requires it to have a complex infrastructure to handle many compilation cases. Compiler Explorer also uses itself a variety of different web technologies. These circumstances affect extent, clarity and comprehensibility of the codebase.

We need to remove all other languages, compilers and unnecessary functionality, or hide it from the user. This includes clearing the web interface of unused elements and buttons.

Compiler Explorer has been used by the Database Systems Research Group in the past for a similar case, **Apfel-DB**.¹² Compiler Explorer has undergone a major code change since then, having been ported from JavaScript to TypeScript. Although we want to use the current version, some modifications done in Apfel-DB, especially the integration of a custom compiler and an image viewer, could serve as a reference. So, a focus of our efforts will be on the integration of DuckDB.

Figure 7 shows a screenshot of Compiler Explorer, modified to run the `MATCH_RECOGNIZE` transpiler, with an integrated DuckDB web shell and an NFA image viewer. The subsequent course of this thesis explains how these changes were achieved.

¹⁰Compiler Explorer: <https://www.godbolt.org>

¹¹Public Repository: <https://github.com/compiler-explorer/compiler-explorer>

¹²Apfel-DB: <https://apfel-db.cs.uni-tuebingen.de>

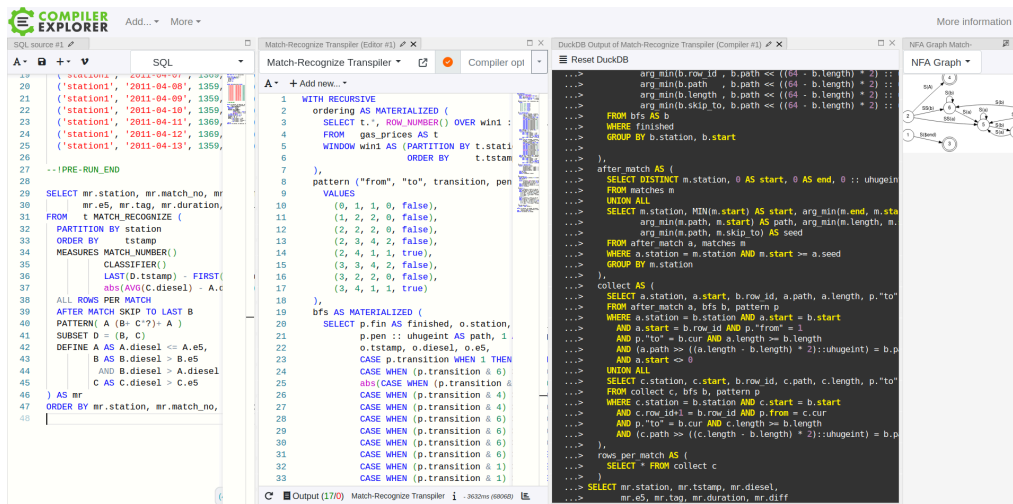


Figure 7: Modified Compiler Explorer. Panes from left to right: Source editor, compiler, DuckDB shell, NFA graph viewer.

4.2.1. Architecture and Design

It is important to understand how Compiler Explorer works under the hood to know where modifications and extensions are needed to implement what is described in [Section 4.1](#).

[Figure 8](#) gives an overview of the architecture and working principle of Compiler Explorer, broken down to the essential functionality.

Compiler Explorer

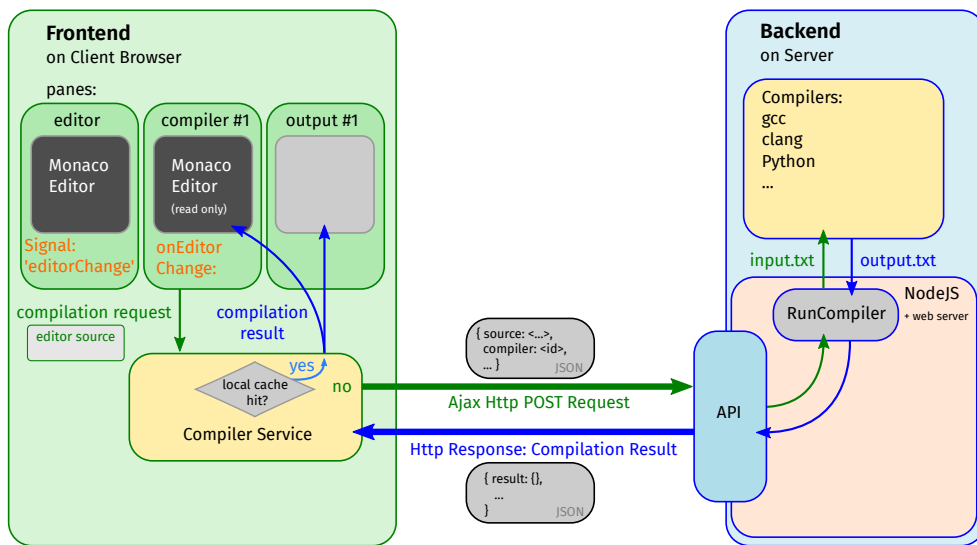


Figure 8: Simplified architecture of Compiler Explorer.

It consists of frontend code, that runs on the client's web browser and backend code, that is executed by a Node.js environment on a server machine. This server also hosts all compiler executables. The frontend features a configurable window layout containing the following window panes: One or more **source editors** of which each is connected to one or more **compiler panes**, showing the assembly output of a selected compiler. Each **compiler pane** has an optional associated **output pane** which displays the compiler's **stdout** and **stderr** messages.

Entering or modifying code in the source editor triggers an event upon which each compiler pane sends a **compilation request**.

That **compilation request**, which includes the source text and the desired compiler, among a lot of other information, is handled by the frontend's **Compiler Service** object. When the **compilation result** belonging to that particular request is not present in the local cache, the request is converted into a JSON representation and sent to the backend's API in an asynchronous HTTP POST request.

In the backend, the execution of the compiler is handled. Data exchange between the Node.js process and the compiler executables is managed through text files which are stored on the hard disk during this process. After compilation, a **compilation result** object containing various information about the compilation process is generated and sent back with an HTTP Response.

On reception, an event is emitted by the **compiler service** which contains the **compilation result**. The window panes associated with the result's compiler act upon the event and display their part of the result.

4.2.2. Technology Stack

Now that we have a basic understanding of the architecture, this chapter will give a brief description of the technologies used by Compiler Explorer to implement it, the entirety of which is usually referred to as the *tech stack*. Due to the large number of technologies Compiler Explorer depends on, we limit ourselves to the most important.

The backend of Compiler Explorer is formed by the JavaScript runtime environment **Node.js**¹³ which also includes an HTTP server. The majority of dependencies to other software packages are managed through **npm**¹⁴, a package manager for Node.js. The routing of API end points is done by a framework called **Express**¹⁵. The backend code is not written in JavaScript but in **TypeScript**¹⁶, a strongly typed JavaScript-based language. This allows type checking the code and helps catching errors earlier, ideally already in the editor. TypeScript needs to be transpiled to JavaScript to be understood by web browsers and node.js. This is one task among others for which **webpack**¹⁷ is responsible, a so-called *bundler*. Its main purpose is to bundle JavaScript files into one file for usage in the browser as well as to bundle assets like images, HTML- and CSS-files which are converted into modules for that purpose.

The frontend utilizes **Golden Layout**¹⁸, which is used to divide Compiler Explorer's page layout in creatable, dockable Windows. The HTML content of these windows is generated by the template engine **pug**¹⁹. It allows writing site templates in a simplified syntax using PUG-files and render them on the server side. The code editor used in the `compiler` and `source` windows are implemented using **Monaco Editor**²⁰, which is a web release of the text editor of Visual Studio Code. As with the backend, all the logic of the frontend is written in TypeScript. It still relies on **jQuery**²¹ for DOM manipulation, event handling and Ajax-Http-Requests.

There is a variety of other tools which help with the development process, like **Husky**²² for automatic pre-commit linting, **Cypress**²³ for testing purposes, and numerous small helper libraries. The build process of the project is managed with **make**²⁴. A convenient way for development is offered by running the command `make dev`, providing automatic cached recompilation after changing the code.

¹³node.js: <https://nodejs.org>

¹⁴npm: <https://www.npmjs.com/>

¹⁵Express: <https://expressjs.com/>

¹⁶TypeScript: <https://www.typescriptlang.org>

¹⁷webpack: <https://webpack.js.org/>

¹⁸Golden Layout: <https://golden-layout.com/>

¹⁹pug: <https://pugjs.org>

²⁰Monaco Editor: <https://microsoft.github.io/monaco-editor/>

²¹jQuery: <https://jquery.com/>

²²Husky: <https://typicode.github.io/husky/>

²³Cypress: <https://www.cypress.io/>

²⁴make: <https://www.gnu.org/software/make/>

Implementation of the Frontend

5.1. Command Line Interface

The implementation of the CLI is accomplished in a standard way using the Python `argparse` library. Three command line flags were configured, allowing to run the compiler as shown in Listing 27. The transpiler is run with a single command only, there is no runtime interaction.

```
1 python -m src [-h] [-o OUTPUT] [-g NFAGRAPH] FILENAME Shell
```

Listing 27: Running the transpiler in the command line.

5.2. Web Application

This chapter provides an in-depth description of how the necessary additions and changes to Compiler Explorer were implemented. For an overview, in Figure 9 these have been inserted and highlighted in red.

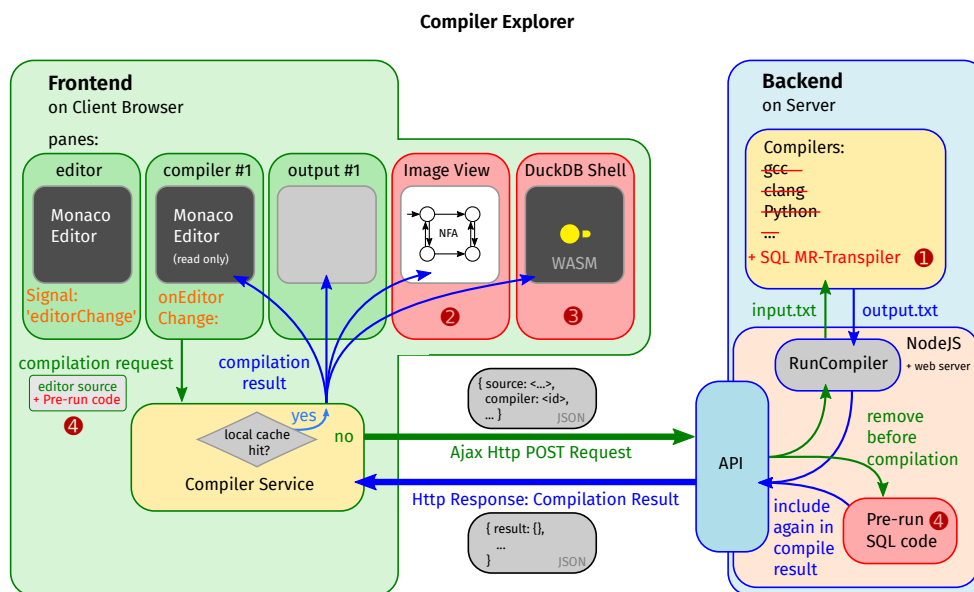


Figure 9: Additions and changes that need to be made to Compiler Explorer.

The `MATCH_RECOGNIZE` transpiler 1 will be added to Compiler Explorer's backend as a new compiler. All of the existing compilers will need to be removed.

The NFA image viewer ② and DuckDB Shell ③ will become new types of window panes, residing in the frontend. There is also a need to find a way to process pre-run SQL statements ④ inserted into the editor.

5.2.1. Integrating the Transpiler

We will first cover the addition of the `MATCH_RECOGNIZE` transpiler as a compiler object. This was essentially accomplished by creating three new files extending the backend, which are visualized in Figure 10.

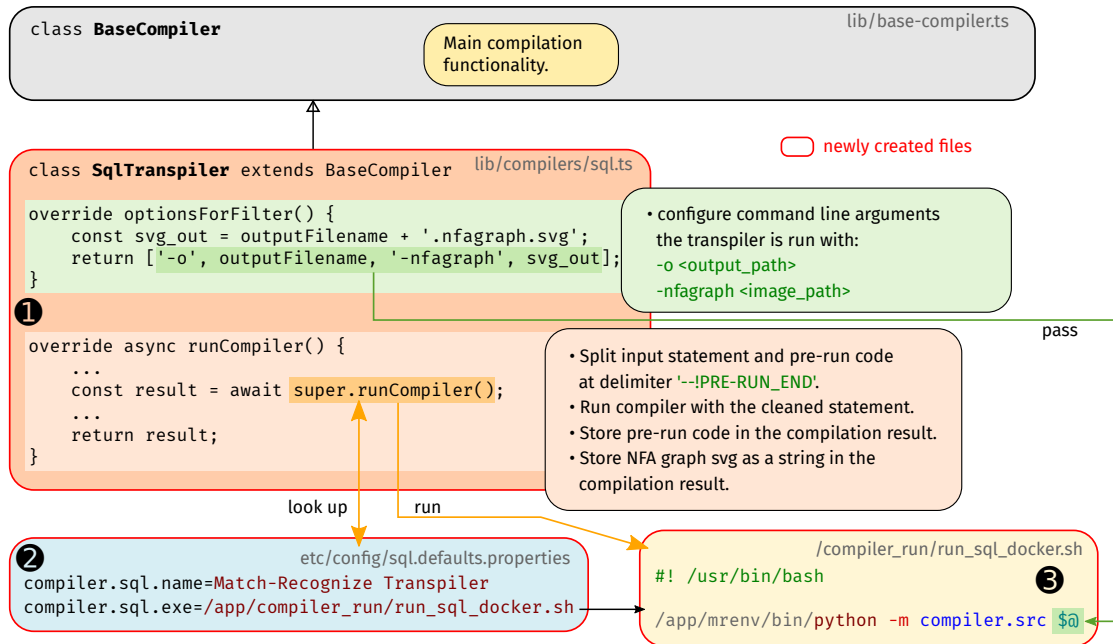


Figure 10: Files and code created to integrate the transpiler.

We first needed to create a new compiler class ①, inheriting from a super class `BaseCompiler`, which contained the main compilation functionality. Calling our transpiler is not very special or complicated, so we could keep most of the existing infrastructure. However, it was necessary to override two methods. The transpiler takes two command line arguments. The first argument is where to write the compilation result, the second argument is where to store the SVG image of the NFA. These are configured using the `optionsForFilter()` method. By default, the compilation output is written to a location that is randomly generated by Compiler Explorer. After compilation the folder and its contained files are deleted. We chose to write the SVG to the same location by adding just a suffix to the filename to ensure proper cleanup and not having to delve further into the naming mechanism.

We also overrode the main compilation method `runCompiler()` to be able to do input preprocessing and modify the compilation result. As mentioned earlier, it should be possible to insert initial SQL code just before the actual `MATCH_RECOGNIZE` statement. This *Pre-run code* must

be placed before a delimiter in the form of a magic comment (`--!PRE-RUN_END`) and must be excluded from compilation. So the question arose whether to do the code separation in the frontend, where it could be passed directly to the DuckDB shell, or to send the untouched editor source to the backend and take care of the separation over there.

Our choice was to do the latter. This way, we did not have to modify Compiler Explorer's infrastructure, just our already custom made `SqlCompiler` class. We remove the Pre-run code from the input and run the compiler with the cleaned statement. After receiving the compilation result, we add the Pre-run code to it as class data, so it can be easily retrieved by the frontend.

The transpiler also puts out the image of the NFA. We add this to the compilation result as well, storing it as a string.

The two other files that were created are a configuration file ❷ for the transpiler, in which its executable is specified, and a bash script ❸ which is executed to run the transpiler using a Python environment on the server.

5.2.2. Integrating an Image Viewer

To integrate an image viewer to display the graph of the NFA generated by the transpiler, we modified a window pane `cfg-view.ts` that was originally used to display a *control flow graph* of assembler code. However, we were only interested in the pane's drag and zoom functionality, so most of the complex graph building functions were disabled.

The pane acts on an `onCompileResult` event. The SVG stored in the compile result is then parsed back from the string representation and added to the original container as an HTML SVG element, being its only child.

5.2.3. Fitting a DBMS into Compiler Explorer's Frontend

The purpose of including a DBMS is to allow a transpiled statement to be tested directly by running it on an included DBMS. Therefore, DuckDB, in combination with DuckDB Shell²⁵, was chosen as a representative. A key reason is, that DuckDB has been compiled to WebAssembly (*Wasm*), enabling it to be run in the browser, *i.e.* in the frontend of our web application.

5.2.3.1. WebAssembly

WebAssembly is a specification for bytecode with the aim to achieve high performance, compact program size due to a binary format, execution safety, and high portability through hardware and platform independence [18]. It can be executed by most modern web browsers and provides a compilation target for languages like C/C++, C# and Rust. Bytecode is optimized ahead of time, so significant performance improvements can be expected over JavaScript which is human-readable code that has to be interpreted first. However, WebAssembly is not designed to replace JavaScript, but to work together with it [19]. For our purpose of providing a test DBMS, WebAssembly is well suited, as it allows us to deliver the DuckDB program to the client, which then runs an instance itself and does not generate any load on the server.

²⁵DuckDB Shell: <https://shell.duckdb.org/>.

5.2.3.2. Integrating DuckDB Wasm

The DuckDB DBMS and the DuckDB Web Shell are two separate programs, both available as Wasm packages. Figure 11 shows where these packages come from and where they are located within the web application. We will first look into the integration of the actual DuckDB DBMS ❶.

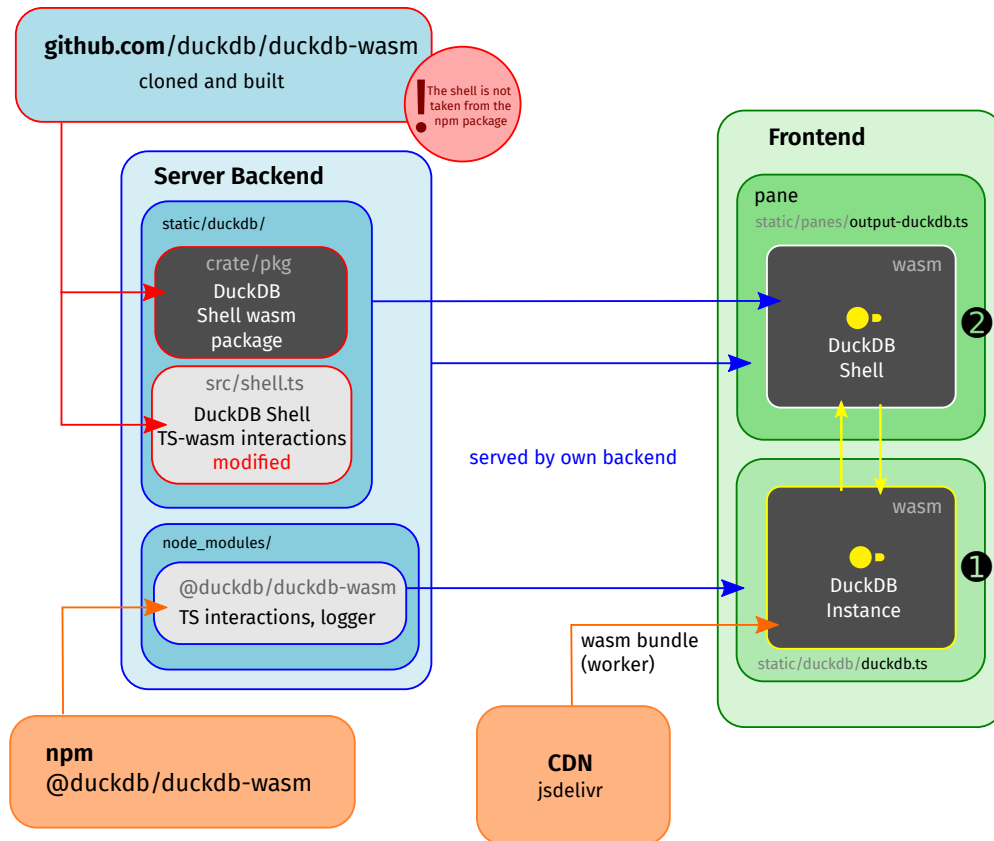


Figure 11: DuckDB WASM: Dependencies and package origin.

An initialization method, which is directly adopted from the DuckDB instantiation instructions²⁶ fetches the Wasm bundle directly to the client from the content delivery network (cdn) **jsdelivr**. For helper functions and TypeScript interaction logic, it is additionally required to import the package `@duckdb/duckdb-wasm` from npm²⁷ into the project. We are then handed a database object which provides an API that can be queried using JavaScript and returns data in the Apache Arrow²⁸ format. We do not intend to query this API directly, but to connect a DuckDB shell ❷ to this running instance of DuckDB.

²⁶DuckDB Instantiation Instructions: <https://www.npmjs.com/package/@duckdb/duckdb-wasm>

²⁷npm: A JavaScript package manager and registry. (<https://www.npmjs.com/>)

²⁸Apache Arrow: <https://arrow.apache.org/>.

5.2.3.3. DuckDB Shell

To integrate DuckDB Shell, a new window pane `output-duckdb.ts` and its associated PUG file were created. The embedding of the shell accomplished by using the provided `embed` method and passing the shell Wasm module, a `<div>` element as a container and our initialized running DuckDB instance.

After this step, the shell could be used by entering statements with the keyboard. However, our intent was to be able to execute statements programmatically by sending them to the shell from the outside. It appeared that the shell was not designed for this use case. The first hurdle was that the package available from npm did not expose any functionality other than the `embed()` method. More methods could be exposed by cloning DuckDB's Github repository, rebuilding DuckDB shell from the Rust sources and copying the recompiled Wasm module and binding files from there. However, the only method that could be utilized to reach the shell from the outside was a method originally intended to start the shell with statements encoded in the URL.²⁹ Because this worked only during the initialization of the shell, a workaround was implemented by reconnecting the shell to the running database each time a request for query execution is received from Compiler Explorer. No significant delay was observed.

To achieve this, the module `shell.ts` needed to be modified. `shell.ts` is provided in the shell's repository and includes TypeScript "glue" code to setup and embed a shell Wasm module. [Listing 28](#) shows the code additions: Two methods of the shell Wasm module had to be forwarded by adding them to the shell runtime object. The runtime object itself had to be returned by the `embed` method, so that it could be accessed from the window pane's module. [Listing 29](#) shows how to execute a query from there, including reattaching the shell.

²⁹To provide a link to DuckDB Shell with initial statements included. Example: <https://shell.duckdb.org/#queries=v0,SELECT-42~>

```

1  export class ShellRuntime { TypeScript
2
3      ...
4
5      // From the outside, the only possibility to pass queries is
6      // through passInitQueries, which hands
7      // URL encoded queries to the shell before it is loaded
8      public async passInitQueries(queries: Array<string>) {
9          shell.passInitQueries(queries);
10     }
11
12     // But passing URL encoded queries works only on attaching the database.
13     // So, it has to be attached again afterwards using this method.
14     public async configureDatabase(db: duckdb.AsyncDuckDB) {
15         shell.configureDatabase(db);
16     }
17 }
18
19 export async function embed(props: ShellProps) {
20     // Initialize and embed the shell.
21     // DB instance, container and wasm module are provided in props
22
23     ...
24
25     // Added returning of the runtime object
26     // created during this process
27     return runtime;
28 }

```

Listing 28: Additions to shell.ts: Providing access to the shell Wasm module.

```

1  // Run a SQL statement in the DuckDB shell. TypeScript
2  async runStmtInShell(stmt: string) {
3      ...
4      const rt = this.shellRuntime;
5      await rt.shellPassInitQueries(Array(stmt));
6      // re-attach database to run initial queries
7      await rt.configureDatabase(await this.db);
8  }

```

Listing 29: Running a statement in the shell from the pane module output-duckdb.ts.

5.3. Deployment

The web application is deployed using **Docker**³⁰. Docker allowed us to create a container holding an isolated environment and all the dependencies needed to run the application, along with the application's web assets themselves. In this way, configuration on the host machine was mainly limited to running the Docker image and forwarding a port.

5.3.1. Docker Container Setup

A Docker image is created by writing a Dockerfile: A sequence of Docker-specific commands that do tasks like running linux commands, setting up environment variables or copying files into the Docker image.

It is worth noting that each command creates a new layer in the docker image. It is therefore advisable to bundle steps and clean up unneeded files in the same step they were generated in so that they don't become part of the image, in order to keep the image size small.

The following is an examination of the Dockerfile, which sets up the image for our complete web application and is shown in [Listing 30](#).

We start by basing our image on an existing one: A Debian linux with an already pre-installed Node.js, which is retrieved from **Docker Hub**³¹ by using the **FROM** command (❶). Next we install Python, which we need to run the transpiler, and Python-related packages (❷). For security reasons, a non-root user is created who is intended to run the application (❸). All required source files of the web app and the transpiler are copied into the docker image (❹). Note that despite specifying the entire directories as copy sources, many unnecessary files such as caches and configurations will be ignored by the use of a separate `.dockerignore` file.

We then set up a Python environment for the transpiler and use `pip` to install all requirements needed for the project (e.g. the ANTLR runtime environment, `pglast`, etc. (❺). Proper user rights are set and the non-root user takes over (❻). We have to move a cache folder location to be reachable by the non-root user (❼).

The application is then built inside the Docker image (❽). Because we aimed for simplicity and reproducibility, this procedure was chosen, so that the build and runtime environment were kept identical and the docker image could easily be built on another developer's machine. During this step, cached files from the building process are removed to keep the image small.

Finally, the application is started using Compiler Explorer's provided `make` file (❾).

³⁰Docker, a containerization software: <https://www.docker.com/>

³¹Docker Hub, Docker's official container image library: <https://hub.docker.com/>.

```
1 Docker
2 FROM node:20.18.0-bookworm-slim } 1
3 SHELL ["/bin/bash", "-c"]
4 WORKDIR /app
5
6 RUN apt update && \
7     apt install -y make python3 python3-pip python3.11-venv graphviz \
8     && rm -rf /var/lib/apt/lists/* } 2
9
10 RUN groupadd -r mruser && \
11     useradd -r -g mruser -d /app -s /bin/bash mruser } 3
12
13 COPY compiler-explorer /app/
14 COPY compiler /compiler/ } 4
15
16 RUN python3 -m venv mrenv && \
17     source mrenv/bin/activate && \
18     /app/mrenv/bin/python -m pip install -r /compiler/requirements.txt } 5
19
20 RUN chown -R mruser:mruser /app /compiler } 6
21 USER mruser } 6
22
23 ENV CYPRESS_CACHE_FOLDER=/app/.cache/Cypress } 7
24
25 RUN make prebuild && \
26     rm -rf /app/node_modules/.cache \
27     /app/.cache/Cypress \
28     /tmp/* } 8
29
30 CMD ["make", "run-only"] } 9
```

Listing 30: Dockerfile for the image of the web application.

5.4. Further Challenges

Besides the complexity of the code base and the shell integration mentioned in [Section 5.2.3.3](#), there were other challenges. One was the multiple emission of events. In certain situations, the `compileRequest` event, upon which a compilation is requested, was triggered more than once. These situations were, among others, the re-opening of the web page with an open shell window and the loading of an example statement. In the DuckDB shell, this caused multiple queries in quick succession, which broke the text display and syntax highlighting. Due to the complexity of Compiler Explorer a reason for the multiple event triggering could not be found. As a workaround, a check has been implemented that compares a request to its successor and ignores identical requests.

Further problems arose due to the asynchronous embedding and initialization of the DuckDB DBMS and DuckDB shell. The asynchronous execution of these long-running processes is desired because it allows the website to stay responsive while starting the DBMS and loading the shell in a non-blocking way. The shell initialization is started upon instantiation of its window pane. In some situations, a statement had to be passed to the shell immediately after creating its containing window pane, making it necessary to wait for the shell to be ready. Passing it before the shell was properly initialized, would have resulted in broken syntax highlighting and text display as well, or no query execution at all. However, in TypeScript the constructor of a class does not allow using the `await` keyword, thus, waiting for the shell creation method (`makeShell`) to finish during the window instantiation. I found myself with the problem of getting the method for passing a query (`runStmtInShell`) to wait for `makeShell` that was already running at the time the `runStmtInShell` was invoked. `makeShell` had not been called by `runStmtInShell` which could therefore not just await a promise returned by the `makeShell` to be settled by this same method, which is the standard way ([Listing 31](#)).

```
1 async runStmtInShell(stmt: String) { ts TypeScript
2     await makeShell() // Not possible, makeShell is called elsewhere (in class
   constructor)
3     ...
4 }
```

Listing 31: Waiting until the promise returned by `makeShell` gets resolved.

A solution was, to expose the promise ([Listing 32](#)), based on a description in the mdn web docs³². This way, the passing method `runStmtInShell` could wait for the promise to be externally set to *fulfilled* by the creation method `makeShell` before sending a query to the shell.

³²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/withResolvers

```

1  class OutputDuckDb extends Pane<OutputState> {
2      ...
3      shellEstablishedPromise: Promise<void>;
4      resolveShellPromise: () => void;
5
6      async makeShell() {
7          // Create a shared promise that is resolvable by calling a function
8          this.shellEstablishedPromise = new Promise<void>(resolve =>
9              (this.resolveShellPromise = resolve));
10         // create the shell
11         ...
12         // resolve the shared promise
13         this.resolveShellPromise();
14     }
15
16     async runStmtInShell(stmt: string) {
17         // wait until shared promise is resolved (shell is ready)
18         await this.shellEstablishedPromise;
19         // send query
20         ...
21     }
22 }

```

Listing 32: Using a shared promise to make `runStmtInShell` wait for `makeShell`. The callback function `resolve` is captured to be called from the outside.

5.5. Discussion of the Frontend

In this section, the decision to use Compiler Explorer as a pre-existing web application is critically evaluated. An alternative would have been to newly implement a web application, which is also considered in this discussion. In addition, the use of specific technologies will be assessed. The most significant advantage of using Compiler Explorer was its stable and comprehensive backend and frontend infrastructure. Implementing a comparable system from scratch would have required substantial development and testing efforts. It was also helpful that many convenient functions were already in place, which make the app appear professional, such as syntax highlighting, caching, dockable windows and user settings for display and handling. Given the complexity and time constraints of the project, many of these features might not have been implemented in a newly developed web application due to the significant workload involved. A major challenge, however, was the size of the application. Compiler Explorer provides a lot of functionality of which only a small part was required for our use case. Unneeded features had to either be removed, which was sometimes hardly possible due to their strong entanglement in the code, or kept and hidden, still contributing to keeping the codebase vast and complicated. Moreover, it has a complex technology stack, involving a lot of tools, each of which adds another

layer of complexity, resulting in a steep learning curve to familiarize oneself with the application. Other drawbacks of Compiler Explorer's size were compilation time and space consumption.

A new implementation of a web application could have been customized according to our requirements and could therefore have been kept simpler and more lightweight. Technologies that are primarily aimed at complex web apps (TypeScript, webpack bundler, pug template engine) could have been avoided. An initial test³³ showed, that even a website using only the basic technologies HTML, CSS, basic JavaScript and the Python transpiler connected via CGI³⁴, running on an Apache web server, was capable of accomplishing the basic task. Such a minimalistic approach might have also been easier to maintain and change for developers not familiar with Compiler Explorer and its frameworks. Some form of window management system like Golden Layout (also used by Compiler Explorer) would have been a good addition in this case, though. It would also have been conceivable for a new implementation to pursue a framework based approach using frameworks such as the already Python-based Flask or Django for the backend, allowing a smoother, WSGI-based³⁵ integration of the Python transpiler, and more modern candidates like React.js or Vue.js for the frontend.

However, due to the need to write a large part of the infrastructure and the GUI itself, a new implementation with a comparable effort would have been inferior to Compiler Explorer in terms of functionality, appearance and convenience.

Also, developing from the ground up requires a thorough understanding of the technologies used, whereas adapting Compiler Explorer only required a deeper understanding of the technologies actually encountered while making additions and modifications. The benefit of already integrated functionality superseded the initial hurdle of understanding the technology stack.

In conclusion, Compiler Explorer proved to be a suitable choice for implementing a web application as a front end for the `MATCH_RECOGNIZE` transpiler.

The use of Docker turned out to be helpful as well. It allowed the entire web application, including the JavaScript runtime, web server, Python installation, ANTLR runtime, and our transpiler, to be deployed in a container, whose (re-)generation could be automated. Otherwise, these dependencies would have had to be installed on the server system, causing higher administrative effort.

³³Included in the Appendix: [Listing 33](#)

³⁴For performance reasons, fastCGI could have been used with a manageable effort.

³⁵Standard interface between web servers and Python applications: <https://wsgi.readthedocs.io/en/latest/what.html>

Conclusion

This thesis contributed to the development of a transpiler to convert SQL code containing `MATCH_RECOGNIZE` into standard SQL.

In the first part of the thesis, we covered the creation of a parser. We analyzed the syntax of a `MATCH_RECOGNIZE` statement and possible inputs and compared it to the given *AST interface*. We found that some parts, e.g. the pattern, had a well formed representation in nodes, other parts, such as expressions, had to be recognized in their exact form and passed on to *pglast*. After comparing different parser generators, we chose ANTLR because of features like the automatic generation of traversing tools, which promised to reduce work load. Based on a syntax description provided by Oracle, we defined a grammar in EBNF. We extended the grammar by adding Lexer rules and refining and restructuring parser rules to better reflect the interface and to make more fine grained decisions. The generated parser based on this grammar was able to successfully parse `MATCH_RECOGNIZE` statements within the defined constraints. We modified a *visitor* tool, generated by ANTLR, to traverse the parse tree output by ANTLR and construct an *abstract syntax tree* in the desired representation. The *visitor* proved to be effective as it only required to override and rewrite the relevant rule methods. Challenges arose in the handling of the parts to be forwarded to *pglast* for which we still had to define a sufficient number of tokens to be recognized correctly. The reconstruction of strings out of the recognized tokens needed additional logic as a workaround. Findings were, that although ANTLR needed a familiarization period, the chosen workflow effectively handled the parsing of a `MATCH_RECOGNIZE` statement.

In the second part of this thesis, we developed a web frontend for the purpose of demonstrating the `MATCH_RECOGNIZE` transpiler.

We modified the existing web application Compiler Explorer expecting to benefit from its stable infrastructure and comprehensive GUI features. We started by removing all unnecessary elements from the Compiler Explorer GUI. We then added the SQL language and integrated the `MATCH_RECOGNIZE` transpiler as a new compiler for this language. A mechanism was implemented to separate pre-run SQL code from the actual `MATCH_RECOGNIZE` statement to the compiler infrastructure in the backend. We modified an existing window pane to display an image of the NFA graph. For the purpose of a direct execution of the transpiled result, we integrated a WebAssembly version of the DuckDB DBMS into Compiler Explorer's frontend. We then embedded DuckDB shell as a new window pane into the frontend, attaching it with the DBMS. One significant problem was the missing functionality of the shell to pass queries from the outside. We addressed this by exposing more methods and utilizing a method for the execution of URL encoded queries during initial shell startup for this purpose. Other challenges arose from the asynchronous nature of the shell, requiring careful synchronization between query execution

and the shell embedding process. We deployed the web application using a Docker container to ensure portability and reproducibility of the development environment. Although the extensive code base and complex technology stack of Compiler Explorer was an initial hurdle, we found that the benefits of the existing functionality outweighed this, and that the workflow of modifying Compiler Explorer was a suitable choice for implementing a web frontend for the `MATCH_RECOGNIZE` transpiler.

6.1. Future Work

There are some adjustments that can be made to add robustness to the application: The parser could be extended by semantic checks, ensuring referenced variables have been defined (see [Section 3.4.1](#)). For aliases referenced in the `SELECT`, `PATTERN` and `MEASURES` clauses, this could be implemented with manageable effort. However, for references inside `expressions` and `conditions`, an inspection of the pglast tree will be necessary.

Regarding the frontend, the current workaround to pass queries to DuckDB Shell from outside, involving the provisional use of the function to load URL encoded queries, could use a cleaner solution. It would be conceivable to implement a specialized function directly in DuckDB Shell by forking and modifying its `Rust` source code.

Bibliography

- [1] Oracle, “SQL for Pattern Matching.” Accessed: Dec. 30, 2024. [Online]. Available: <https://docs.oracle.com/en/database/oracle/oracle-database/21/dwhsg/sql-pattern-matching-data-warehouses.html>
- [2] PostgreSQL, “Cyclic Tag System.” Accessed: Dec. 30, 2024. [Online]. Available: https://wiki.postgresql.org/index.php?title=Cyclic_Tag_System&oldid=15106
- [3] Wikipedia, “Parsing.” Accessed: Dec. 30, 2024. [Online]. Available: <https://en.wikipedia.org/wiki/Parsing>
- [4] P. Documentation, “The Parser Stage.” Accessed: Feb. 15, 2025. [Online]. Available: <https://www.postgresql.org/docs/current/parser-stage.html>
- [5] libpg_query, “C library for accessing the PostgreSQL parser outside of the server.” Accessed: Feb. 15, 2025. [Online]. Available: https://github.com/pganalyze/libpg_query
- [6] L. Gaifax, “PostgreSQL Languages AST and statements prettifier.” Accessed: Feb. 15, 2025. [Online]. Available: <https://pglast.readthedocs.io/en/latest/index.html>
- [7] T. Æ. Mogensen, “Introduction to Compiler Design.” Springer, pp. 68–73, 2024.
- [8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers Principles, Techniques & Tools.” Springer, pp. 43, 224, 2007.
- [9] Christian Wagenknecht, Michael Hielscher, “Formale Sprachen, abstrakte Automaten und Compiler.” Springer, p. 33, 2022.
- [10] D. M. Beazley, “PLY (Python Lex-Yacc).” Accessed: Nov. 15, 2024. [Online]. Available: <https://www.dabeaz.com/ply/ply.html>
- [11] E. Shinan, “Lark - a parsing toolkit for Python.” Accessed: Nov. 15, 2024. [Online]. Available: <https://github.com/lark-parser/lark>
- [12] wwwantlr.org, “ANTLR.” Accessed: Nov. 15, 2024. [Online]. Available: <https://www.antlr.org/>
- [13] T. Parr, “ANTLR Documentation.” Accessed: Nov. 15, 2024. [Online]. Available: <https://github.com/antlr/antlr4/tree/dev/doc>
- [14] K. F. Terence Parr Sam Harwell, “Adaptive LL(*) Parsing: The Power of Dynamic Analysis.” 2014.
- [15] Francisco Ortin, Jose Quiroga, Oscar Rodriguez-Prieto, Miguel Garcia, “An empirical evaluation of Lex/Yacc and ANTLR parser generation tools.” 2021.
- [16] J. Jones, “Abstract Syntax Tree Implementation Idioms.” 2003.
- [17] Elisabeth Robson, Eric Freeman, “Entwurfsmuster von Kopf bis Fuß.” O'REILLY, pp. 614–615, 2022.

- [18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, F Bastien, "Bringing the Web up to Speed with WebAssembly." 2017. doi: <http://dx.doi.org/10.1145/3062341.3062363>.
- [19] MDN Web Docs, "WebAssembly." Accessed: Jan. 28, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/WebAssembly>

List of Figures

Figure 1: Structure of the transpiler, topics covered in this thesis .	3
Figure 2: Example grammar with two production rules.	6
Figure 3: Class structure of the MR-AST interface	8
Figure 4: ANTLR scheme in the context of the thesis' use case.	12
Figure 5: Storage of expressions in MEASURES clause.	16
Figure 6: ANTLR parse tree (section) representing a minimal MATCH_RECOGNIZE subclause.	18
Figure 7: Modified Compiler Explorer. Panes from left to right: Source editor, compiler, DuckDB shell, NFA graph viewer.	30
Figure 8: Simplified architecture of Compiler Explorer.	31
Figure 9: Additions and changes that need to be made to Compiler Explorer.	33
Figure 10: Files and code created to integrate the transpiler.	34
Figure 11: DuckDB WASM: Dependencies and package origin.	36

All figures in this thesis were created by the author.

Technologies and Tools used

The following tools were used for writing this theses:

- Typst³⁶ - Writing and typesetting.
- Inkscape³⁷ - All figures.
- DeepL Write³⁸ - English language support.
- codly³⁹ - All code listings.

³⁶Typst: <https://typst.app/>

³⁷Inkscape: <https://inkscape.org/>

³⁸DeepL Write: <https://www.deepl.com/de/write>

³⁹codly: <https://typst.app/universe/package/codly/>



Appendix

A.1. Grammar for Parsing MATCH_RECOGNIZE

```
1  grammar Mr; ANTLR
2
3  options { caseInsensitive = true; }
4
5  query :
6      'SELECT' expression(',' expression)* 'FROM'
7      table_expression
8      rest?
9      ';'
10 ;
11
12 table_expression :
13     table_name
14     row_pattern_recognition_clause
15     t_alias?
16 ;
17
18 rest:
19     rest_part+
20 ;
21
22 rest_part:
23     ( 'WHERE' | 'GROUP BY' | 'HAVING' | 'ORDER BY' | 'FETCH FIRST' | 'LIMIT')
24     expression(',' expression)*
25 ;
26
27 row_pattern_recognition_clause :
28     'MATCH_RECOGNIZE' '('
29     row_pattern_partition_by?
30     row_pattern_order_by?
31     row_pattern_measures?
32     row_pattern_rows_per_match?
33     row_pattern_skip_to?
34     'PATTERN' '(' row_pattern ')'
35     row_pattern_subset_clause?
```

```

36     'DEFINE' row_pattern_definition_list
37     ')'
38     ;
39
40 row_pattern_partition_by :
41     'PARTITION BY' column(',') column)* ;
42
43 row_pattern_order_by :
44     'ORDER BY' column(',') column)* ;
45
46 row_pattern_measures :
47     'MEASURES' row_pattern_measure_column(',') row_pattern_measure_column)* ;
48
49 row_pattern_measure_column :
50     expression 'AS' c_alias ;
51
52 row_pattern_rows_per_match :
53     row_pattern_one_row_per_match
54     | row_pattern_all_rows_per_match
55     ;
56
57 row_pattern_one_row_per_match : 'ONE ROW PER MATCH' ;
58
59 row_pattern_all_rows_per_match : 'ALL ROWS PER MATCH' ;
60
61 row_pattern_skip_to :
62     'AFTER MATCH'
63     ( skip_to_next_row
64     | skip_past_last_row
65     | skip_to_first
66     | skip_to_last
67     | skip_to)
68     ;
69
70 skip_to_next_row : 'SKIP TO NEXT ROW' ;
71
72 skip_past_last_row : 'SKIP PAST LAST ROW';
73
74 skip_to_first : 'SKIP TO FIRST' variable_name ;
75
76 skip_to_last : 'SKIP TO LAST' variable_name ;
77
78 skip_to : 'SKIP TO' variable_name ;
79

```

```

80 row_pattern :
81     row_pattern_term
82     | row_pattern '|' row_pattern_term
83     ;
84
85 row_pattern_term :
86     row_pattern_factor
87     | row_pattern_term row_pattern_factor
88     ;
89
90 row_pattern_factor :
91     row_pattern_primary row_pattern_quantifier? ;
92
93 row_pattern_quantifier :
94     row_pattern_symbol_quantifier
95     | row_pattern_range_quantifier
96     ;
97
98 row_pattern_symbol_quantifier :
99     '*' '?'?
100    | '+' '?'?
101    | '?' '?'?
102    ;
103
104 row_pattern_range_quantifier :
105     ( '{' NUMBER?', 'NUMBER?'}'
106     | '{' NUMBER '}' ) '?'?
107     ;
108
109 row_pattern_primary :
110     variable_name
111     | SYMBOL
112     | '(' row_pattern? ')'
113     | '{-' row_pattern '-}'
114     | row_pattern_permute
115     ;
116
117 row_pattern_permute :
118     'PERMUTE' '(' row_pattern (',' row_pattern)* ')' ;
119
120 row_pattern_subset_clause :
121     'SUBSET' row_pattern_subset_item (',' row_pattern_subset_item)* ;
122
123 row_pattern_subset_item :

```

```

124     variable_name '=' '(' variable_name(',' variable_name)* ')' ;
125
126 row_pattern_definition_list :
127     row_pattern_definition(',' row_pattern_definition)* ;
128
129 row_pattern_definition :
130     variable_name 'AS' condition ;
131
132 expression : expression_part+ ;
133 condition : condition_part+ ;
134
135 column: TEXT ;
136 variable_name : TEXT ;
137 expression_part : TEXT | NUMBER | SQL_STRING | '=' | '*' | '/' | '+' | '-' | '('
| ')' | 'AS' ;
138 condition_part : expression_part | '<' | '>' | '<=' | '>=' | '<>' ;
139 c_alias: TEXT ;
140 t_alias : 'AS' TEXT;
141 table_name: TEXT ;
142
143 SYMBOL : [$|^] ;
144 TEXT : ([a-z_#@.])([a-z0-9_#@.])*;
145 NUMBER : [0-9]+ ; // [0-9]+(.[0-9]+)?
146 SQL_STRING : '\'' ( ~'\'' | '\''\'' )* '\'' ;
147
148 COMMENT : '--' ~[\n]* -> skip;
149 WS: [ \t\r\n]+ -> skip;
150

```

A.2. Minimal Web Frontend Test

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Match Recognize Transpiler Test</title>
7   <script>
8     function transpile(event) {
9       event.preventDefault();
10      const mr_statement = document.getElementById("mr").value;
11      const url = "/cgi-bin/mr_transpile_dummy.py?mr=" + mr_statement;
12      fetch(url)
13        .then(response => response.text())
14        .then(data => {
15          document.getElementById("transpiled_result").innerHTML = data;
16        });
17    }
18  </script>
19 </head>
20 <body>
21   <form onsubmit="transpile(event)">
22     <label for="mr">Enter a MATCH RECOGNIZE statement:</label><br>
23     <textarea name="mr" id="mr" rows="20" cols="40"></textarea>
24     <input type="submit" value="Transpile">
25   </form>
26   <div>Result:</div>
27   <div id="transpiled_result"></div>
28 </body>
29 </html>
```

Listing 33: A minimal web front end test.

```
1  #!/usr/bin/python3
2  import cgi
3  form = cgi.FieldStorage(encoding="utf-8")
4  mr_statement = form.getvalue("mr")
5
6  # Run transpiler with input statement #
7
8  result = "WITH RECURSIVE ..."
9
10 print("""
11     Input Statement:<br>
12     {0}<br>
13     Transpilation Result:<br>
14     {1}<br>
15     """.format(mr_statement, result))
```

Listing 34: Python CGI script for running the transpiler (requires Apache web server configured for CGI).