Prototype: Create Reusable DAX Parsers using ANTLR 4

Xue Jin

CS421 - Programming Languages & Compilers | Summer 2023

Final Project Report

Overview

Project Motivation

This project is a prototype to create DAX parsers using ANTLR 4. As a software engineer for Power BI, a data analysis and visualization tool for business intelligence, I aim to leverage the knowledge gained from this course (CS421 - Programming Languages & Compilers) to enhance my work. After discussing the course content and our future backlog with my manager Taylor Clark, I selected this task for its independence from our product code (to avoid confidentiality violations), scalability for reuse, and alignment with the course content.

Power BI uses DAX (Data Analysis Expression) language as its primary customer-facing query language. DAX has syntax, functions, and context as its primary framework elements, and has its own set of grammars [1]. Currently, Power BI's codebase is written in C# (for modeling and query handling) and TypeScript (for application hosting and UI). Each side employs a DAX parser written in the respective target languages, with the actual DAX grammar integrated into the code that handles the parse results. This approach of duplicating the same grammar, lexer, and parser rules is not ideal, as it lacks extensibility to new languages, requires a complete rewrite when changing the target language, and increases the likelihood of introducing errors during code duplication.

Project Goal

The goal of this project is to prototype and potentially create reusable DAX parsers in C# and TypeScript from a single (set of) grammar file(s). This approach would extract the DAX grammar code from the functional code and enable language-agnostic parser generation to a certain extent. ANTLR 4 is a promising tool to accomplish this objective, as it can generate a parser to construct a parse tree and provide a listener/visitor interface to respond to phase recognition if

provided with a given set of grammars, with support for 10 target languages including C# and Typescript [2]. It could also spin into a good open-source project for DAX parser users.

Broad Accomplishments

The project is intended to serve as a brain teaser and prototype effort to test the feasibility of the idea. A major accomplishment, although not very tangible, is for me to learn about DAX language and ANTLR 4, so that I have the basic understanding and necessary tools to carry out the task. In terms of actual code produced, the project establishes an essential framework for DAX lexer rules, DAX parser rules, and a process to create the parser in target languages with basic unit testing.

As described above, I hope to carry this project beyond the timeline of this course. Future tasks would include completing the DAX grammar file set and cross check with Power BI's existing code to make sure it is comprehensive and accurate, creating parsers in C# and Typescript, and refactoring Power BI's code to replace the existing parsers.

Implementation

Status of the Project

The scope of the project, as stated in the project proposal, consists of two parts:

1. Required: Create a single grammar file containing all lexer and parser rules for DAX (*.g4 file).
2. Prototype: Generate DAX parsers in C# and TypeScript using ANTLR 4.

Because DAX has a set number of functions with defined parameters, I originally planned to complete the easy labor of finishing the grammar file(s) first (Part 1), because it is for sure needed and has less ambiguity to implement. However since DAX has over 250 functions [3], the repetitive labor gradually diminishes the prototyping purpose of this project. So I decided to switch gear and go deeper rather than broader.

DAX functions are currently classified into 14 categories and an unclassified "new function" category [3]. Functions in each category are similar in their purpose and input / output types, e.g. aggregation functions, date and time functions, financial functions, etc. I chose "**aggregation functions**" and "**information functions**", totalling 54 functions, as the set to write grammar for for this project. It is because these two categories use similar parameter tokens, for example column reference, scalar value, table expressions, while other categories such as "date and time functions" use completely different sets of input such as datetime.

What Went Well?

Start small and simplify the project along the way. I would describe the work process as a spiral growth: as I gained enough knowledge to kick start, I tested how ANTLR 4 works with two DAX functions and associated lexer rules (or simplified mocked up rules). After it worked, I learned more about how DAX functions work, then expanded the effort to the entire function category, then to the second category. During the expansion, I noticed the original lexer rules were either insufficient or inaccurate, so I looped back and revised them, referencing what we have in current Power BI DAX parsers. Along the way, I made the call to only focus on two function categories and made them work, rathen than coding in all functions but with broken pieces to fix later. I think this is the right approach to follow because, although the amount of code was less, it meets my goal to prototype and set up a clear structure for the grammar file. As the existing two function categories are proven working with correct ANTLR 4 parse results, I could bridge them with Power BI's parser to make the flow works, before expanding to a complete function list.

Referencing existing resources. Since both DAX and ANTLR 4 are new tools for me, I utilized many existing resources to educate myself. Below is a primary list of resources I referenced:

- DAX:
    - Official reference: Data Analysis Expressions (DAX) Reference
    - DAX Guide
- ANTLR 4:
    - Official website: ANTLR

    - ANTLR 4 Documentation: https://github.com/antlr/antlr4/blob/master/doc/index.md
    - Dr. Terence Parr (author) talking about ANTLR 4: Terence Parr - ANTLR4 on Vimeo
    - The ANTLR Mega Tutorial
- ANTLR 4 grammars: GitHub - antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions.

  There are many ANTLR 4 grammar projects for different languages in production. I referenced MySQL for ideas to set up my grammar file since it was similar to DAX in terms of structure and its focus on query. It will be a useful resource as I progress to the next part of this project as well.

Referencing what's covered in this course. Although this project does not involve writing the actual parser, but rather collecting and translating DAX grammars into a parser generating tool, I encountered several problems that were solved by what we learned in class. One of them was on how to define the lexer rule for atomic tokens like IDENTIFIER, NUMBER, or WHITESPACE. I used regular expressions and learned that ANTLR 4, similar to regex, also tries to match to the largest possible token [4], or what comes first, so the ordering of the lexer rules matters - more specific rules should be presented first. Similarly, the parser rules can also be written with regex if there are optional inputs with 0, 1 or more repetitions.

Another important aspect is that ANTLR 4 supports direct left recursions, but not indirect ones or mutually left-recursive rules [5]. When I got an error complaining "The following sets of rules are mutually left-recursive [expression, column, table, variant, identifiers]", I initially looked into resolving them through fixing common prefixes, left factoring, or epsilon production. But taking a step back, I realized the mutually left recursion was introduced by an unclear / incorrect hierarchy of lexer rule definitions. As we learned in class, if the nonterminals have actual meanings, resolving the issue with theoretical approaches may not work. The real solution lies in a more accurate lexer rule set.

What Works Partially?

Definition of lexer rules. I struggled a lot in this aspect because in the official DAX reference[1], parameters are described in very vague terms. For example, columns could appear as "columnName" "column" "name", some cannot be expressions but some can either be a name or yield to a name. Further there are restrictions of using only fully qualified column reference,[2] or whether a column is in or related to a table.[3] Same applies to tables, values, etc. I initially defined the necessary lexer and parser rules for parameter types as needed, but this approach introduced overlapping definitions and ambiguity. What serves the purpose better would be a comprehensive picture with all lexer rules for the parameters precisely defined, from where I could simplify to rules I needed. So I turned to how Power BI defines the parameter roles and built the lexer rules from there. Yet because the parameter roles for Power BI's DAX parser are overly complex, which in the current stage of my code I do not need, I set the "root" atomic lexer rule to be IDENTIFIER (basically any input starting with a letter and contains letters and numbers), so that the parse tree still contains the correct token types but is not bounded by strict token rules.

What Is Not Implemented?

Part 2 of the project. I carried the project so far as to create the auto-generated C# project with Antlr4.Runtime.Standard and wrote tests in C# around parse result syntax, but did not write the actual parser code to make it work in C#. I also did not try to integrate it with Power BI's existing parser code. Time constraint was the major reason, but I also hoped to keep this project isolated from confidential project code. I think what I have is sufficient to start a good conversation with domain experts in DAX parser to collect their insights, so that the refactor work is the most efficient.

Components of the Code

In the project repository, "\DaxGrammar\AntlrCSharp\DAXGrammar.g4" and "\DaxGrammar\AntlrCSharpTests\DAXGrammarTests.cs" contain code I wrote. Other files are

[1] Link to the DAX Reference: https://learn.microsoft.com/en-us/dax/
[2] Example: https://learn.microsoft.com/en-us/dax/best-practices/dax-column-measure-references#columns
[3] Example: https://learn.microsoft.com/en-us/dax/issubtotal-function-dax

either auto generated or created as necessary placeholders. “DAXGrammarTests.cs” contains unit tests which will be discussed later.

The “DAXGrammar.g4” file is the ANTLR 4 grammar file, and is structured as below:

- Parser Rules
    - startRule
    - functionCategory
        - aggregationFunctions
            - Actual parser rules for the aggregation functions
        - informationFunctions
            - Actual parser rules for the information functions
        - (TO BE EXPANDED)
    - ParamsRoles
        - ParamsRoles (that actually appeared as input to functions)
        - ParamsTokens (that construct the alternative types of the ParamRoles)
- Lexer Rules
    - Operators
        - Arithmetic Operators
        - Comparison Operators
        - Text Concatenation Operator
        - Logical Operators
        - Parentheses
        - (Others)
        - Enums
        - Letters (to define function tokens to be case insensitive)
        - Function Category
            - Symbols for aggregationFunctions
            - Symbols for informationFunctions
        - Utilities

- Enum Definitions
- Datatypes

Major Capabilities of the Code[4]

Using MAXX [6] as an example, whose function is:

MAXX(<table>,<expression>,[<variant>])

The example input string will be: `MAXX(tab1, expr1, TRUE)`

Parse input and show parse tree in text:

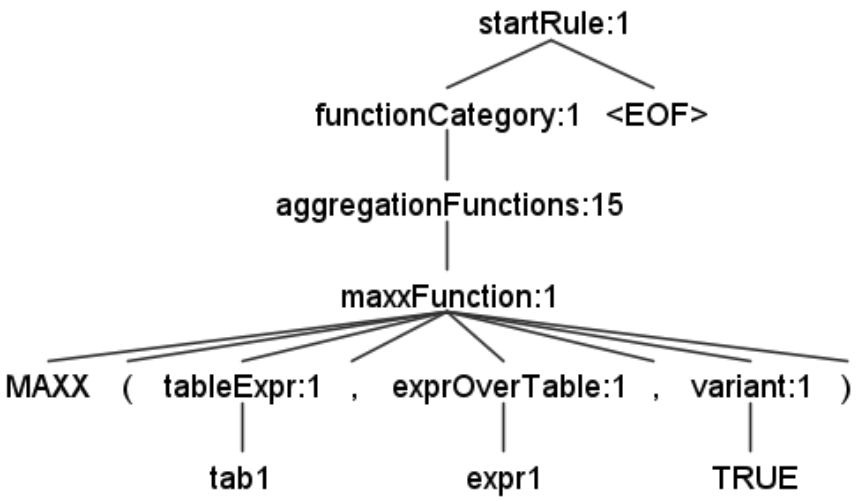
CMD `antlr4-parse DAXGrammar.g4 startRule -tree`

OUTPUT
```
(startRule:1 (functionCategory:1 (aggregationFunctions:15
(maxxFunction:1 MAXX ( (tableExpr:1 tab1) , (exprOverTable:1
expr1) , (variant:1 TRUE) )))) <EOF>)
```

Parse input and visualize the parse tree

CMD `antlr4-parse DAXGrammar.g4 startRule -gui`

OUTPUT



[4] Below I only listed the functional commands for each result. For detailed instruction on how to install ANTLR 4 and basic commands, please refer to: https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

Parse input and show the tokens and trace through the parse:

CMD

```
antlr4-parse DAXGrammar.g4 startRule -tokens -trace
```

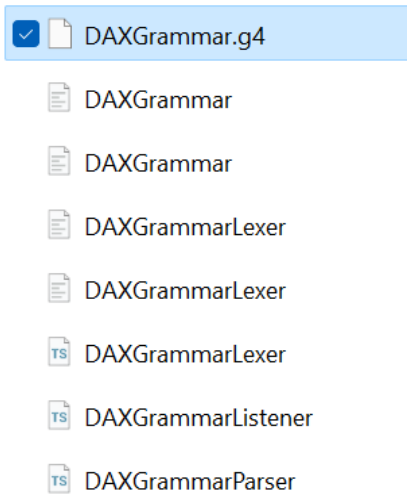
OUTPUT

```
[@0,1:4='MAXX',<MAXX_SYMBOL>,1:1]
[@1,5:5='(',<'('>,1:5]
[@2,6:9='tab1',<IDENTIFIER>,1:6]
[@3,10:10=',',<',',>,1:10]
[@4,12:16='expr1',<IDENTIFIER>,1:12]
[@5,17:17=',',<',',>,1:17]
[@6,19:22='TRUE',<BOOLEAN>,1:19]
[@7,23:23=')',<')'>,1:23]
[@8,26:25='<EOF>',<EOF>,2:0]
enter   startRule, LT(1)=MAXX
enter   functionCategory, LT(1)=MAXX
enter   aggregationFunctions, LT(1)=MAXX
enter   maxxFunction, LT(1)=MAXX
consume [@0,1:4='MAXX',<38>,1:1] rule maxxFunction
consume [@1,5:5='(',<17>,1:5] rule maxxFunction
enter   tableExpr, LT(1)=tab1
consume [@2,6:9='tab1',<84>,1:6] rule tableExpr
exit    tableExpr, LT(1)=,
consume [@3,10:10=',',<21>,1:10] rule maxxFunction
enter   exprOverTable, LT(1)=expr1
consume [@4,12:16='expr1',<84>,1:12] rule exprOverTable
exit    exprOverTable, LT(1)=,
consume [@5,17:17=',',<21>,1:17] rule maxxFunction
enter   variant, LT(1)=TRUE
consume [@6,19:22='TRUE',<22>,1:19] rule variant
exit    variant, LT(1)=)
consume [@7,23:23=')',<18>,1:23] rule maxxFunction
exit    maxxFunction, LT(1)=<EOF>
exit    aggregationFunctions, LT(1)=<EOF>
exit    functionCategory, LT(1)=<EOF>
consume [@8,26:25='<EOF>',<-1>,2:0] rule startRule
exit    startRule, LT(1)=<EOF>
```

Generate parser code in target language (e.g. TypeScript):

CMD `antlr4 -Dlanguage=TypeScript DAXGrammar.g4`

OUTPUT



Tests

Due to the nature of this project, it is quite difficult to come up with a good testing strategy. It is not at a stage suitable for feature tests and beyond, so I decided to focus on manual testing (while writing the code) and unit testing. Manual testing provided useful information to debug and revise the code, using ANTLR 4's native commands.

As for more automated and reusable unit tests, I found little information about testing the ANTLR 4 grammar code itself [4][7]. Besides, provided with the built-in parse tree and visualization, and detailed token and traces (as presented above), it is redundant to test the actual ANTLR 4 grammar. So I decided to research more into unit testing in the target language. I chose C# with the understanding that other target languages will follow a similar process. There is a very good tutorial on getting started and unit testing ANTLR 4 project in C#[5] that I referenced heavily.

[5] Tutorial: https://tomassetti.me/getting-started-with-antlr-in-csharp/

Specifically for my project, I mainly care about whether the parse result of the input function is valid. So my unit tests are all about checking for syntax errors. I established a pattern to create a parser with an input function string, call the start rule of the parser, and confirm that the parser is indeed created, called to generate output, and proceeded with no syntax error. I chose several functions that could have ambiguous and/or optional parameters to write tests for. "\DaxGrammar\AntlrCSharpTests\DAXGrammarTests.cs" contains these example tests. To run the test, run `dotnet test` in the test directory.

Should this project be carried further into the integration with Power BI's DAX parser, testing will be largely covered by Power BI's existing tests, which is also a benchmark of whether this refactor effort works.

Listing

Project repository: **https://github.com/adaxjin/CS421_FinalProject_DaxGrammar**

References

[1] "Learn DAX basics in Power BI Desktop - Power BI," Jan. 19, 2023. https://learn.microsoft.com/en-us/power-bi/transform-model/desktop-quickstart-learn-dax-basics (accessed Jul. 30, 2023).

[2] Terence Parr, "ANTLR v4 README.md," *GitHub*. https://github.com/antlr/antlr4/blob/dev/README.md (accessed Aug. 01, 2023).

[3] "DAX function reference - DAX," Jun. 21, 2022. https://learn.microsoft.com/en-us/dax/dax-function-reference (accessed Aug. 01, 2023).

[4] Ricardo, "Unit tests for ANTLR Lexer." https://ssricardo.github.io/2018/junit-antlr-lexer/ (accessed Aug. 01, 2023).

[5] parrt and beardlybread, "ANTLR 4 - Left-recursive rules," *GitHub*. https://github.com/antlr/antlr4/blob/master/doc/left-recursion.md (accessed Aug. 01, 2023).

[6] "MAXX function (DAX) - DAX," Jul. 12, 2023. https://learn.microsoft.com/en-us/dax/maxx-function-dax (accessed Aug. 01, 2023).

[7] Ricardo, "Unit tests for ANTLR parser." https://ssricardo.github.io/2018/junit-antlr-parser/ (accessed Aug. 01, 2023).

[8] G. Tomassetti, "Getting Started With ANTLR in C#," *Strumenta*, Jan. 16, 2018. https://tomassetti.me/getting-started-with-antlr-in-csharp/ (accessed Aug. 01, 2023).