PLC Report

Nayden Nedev (nn1g22@soton.ac.uk)

Alejandro Andino (aam1g22@soton.ac.uk)
Patrick Jfremov-Kustov (pjk1g22@soton.ac.uk)

1 Our Inspirations

We began by researching different query languages, starting with SQL due to its widespread use in relational databases and previous experience. We quickly realised that that could have translated better with the Graph Data documents we were given as input. This was mainly related to the way it treated relationships. This made us look further into more suitable languages that worked well with Graph Data documents, which led us to Cypher.

While we were initially drawn to Cypher for its clear syntax and seamless integration with Neo4j files, we also gave due consideration to Gremlin. However, after careful evaluation, we decided against it due to its less intuitive readability. Our primary inspiration became Cypher, with influences from SQL, a language we were already familiar with. This allowed us to leverage our existing knowledge, saving valuable time that we could redirect towards developing our unique grammar rules and coding.
This approach boosted our progress and ensured that our language, Decypher, maintained clarity and ease of comprehension for users familiar with SQL and Cypher.

2 Main Language Features

Token Definitions: Tokens are defined for keywords such as FIND, FROM, RETURN, WHERE, etc., each associated with a specific constructor. Tokens are symbols like parentheses, brackets, commas, etc. Additionally, tokens are defined for string literals (TokenString), identifiers (TokenIdentifier), and integers (TokenInteger).

Regular Expressions: Regular expressions are used to define token patterns, such as for string literals ($\ddot{(}$($alphanum — $white — [$\dot{]}$)+$\ddot{)}$ and identifiers ($alpha [$alpha $digit _ $\acute{]}$*). These patterns capture combinations of alphanumeric characters, whitespace, and specific symbols.

Lexer Rules: Lexer rules are specified using regular expressions and corresponding actions to construct tokens. For example, the lexer recognizes keywords like FIND, FROM, RETURN, etc., and constructs tokens with appropriate constructors and position information. String literals and identifiers are also recognized, and tokens are constructed accordingly.

Position Information: Each token is associated with position information (line and column) indicating its location in the input stream. The AlexPosn type is used to represent position information.

Token Data Type: Tokens are represented using the Token data type, which includes constructors for different kinds of tokens (keywords, symbols, strings, identifiers, integers). Each token constructor is associated with position information and, where applicable, additional data (such as the string value for string literals and the integer value for integers).
Token Position Function: A function tokenPosn is a way to extract position information from tokens. This function takes a token and returns a string representing its position (line and column).

Once that was done, we started moving into the grammar. This part was better since the queries had already been thought out better. Again, we started by implementing the basic grammar for the queries we thought out for the tokens section. Similarly, the next step was to think of possible variations of these queries and debate whether or not it made sense to adapt the grammar to allow said variations. The ones we thought made sense were added, and the rest were discarded.

Query Structure: Queries contain several clauses: FindClause, FromClause, ReturnClause, WhereClause, UpdateClause, CreateClause, and DeleteClause. Queries can be constructed with combinations of these clauses, such as Find-Where, Find-Create, Update-Return, etc.

Tokens: The DSL recognizes various tokens such as keywords (FIND, FROM, RETURN, WHERE, etc.), identifiers (IDENT), strings (STRING), integers (INTEGER), and symbols (such as parentheses, commas, etc.).
Grammar Rules: The grammar rules define the syntax and structure of different clauses and elements within the DSL. For example, the grammar specifies how to parse a FindClause, FromClause, ReturnClause, etc. It defines the precedence and associativity of operators like AND, OR, NOT, etc., using %left, %right, and %nonassoc directives.

Data Types: The parser generates abstract syntax trees (ASTs) representing different elements of the DSL, such as queries, clauses, identifiers, relationships, conditions, comparisons, etc. Each data type in the AST corresponds to a specific construct in the DSL, facilitating further processing or interpretation of the queries.

Error Handling: The parser includes error handling mechanisms to report parsing errors, such as encountering an unexpected token or failing to match a grammar rule. The parseError function handles error cases and provides meaningful error messages, including the token's position where the error occurred. This will be expanded on the Informative Error Messages section.

Semantic Actions: Semantic actions are associated with grammar rules to construct AST nodes during parsing. For instance, when parsing a FindClause, FromClause, or any other clause, specific actions are taken to build the corresponding AST nodes.
DSL Operations: The DSL supports various operations like finding, updating, creating, and deleting data entities. These operations are represented through different combinations of clauses and elements within the queries.

3 How to use our Interpreter

Firstly, you must call Alex on the Tokens.x file. After that, call happy on the Grammar.y file. Once both operations have been done, the next step is to compile the interpreter using ghc so that the entire command will be ghc Gql.hs. After that, call the main method using the name of the input file as an argument. For example, if the file to read from is access.n4j, the call will be "./Gql access.nj4".

4 Scoping and Lexical Rules

Regarding getting the tokens, we first sat down to read and understand the specifications and get to the point where we knew what we were asked to do. Once we knew what the tasks required us to do, we decided what tokens to implement. We thought of how the queries we would need would look, and from there, we started writing all of the tokens. After that, we thought of what variations of our tokens could be used. We then debated as to whether or not these variations were worth adding to, and the ones we decided were worth adding did so.

5 Syntax

Our syntax is straightforward:
- You MUST have a FIND query
- You MUST have a FROM query
- You MUST have a RETURN query
WHERE, UPDATE and DELETE can be at most once and in that order.

6 Additional Features

The main additional feature we implemented was syntax highlighting, which will be described in more detail in its respective sections. We also allowed the user to use comments in their code to annotate their code as they see fit. These comments consist of two dashes, "–" and are only single lines.

We planned on adding more features, but unfortunately, we could not implement them fully or at all due to time constraints. These features include type checking, subtyping and type safety, string interpolation, refactoring tools, auto-completion and code suggestions, debugging and tracing, integration with existing tools, domain-specific optimisations, extensibility, recursive queries and code folding.

7 Type Checking

Type Checking could have been used to check if the query the user is inputting is valid. This would ensure that no invalid query is inputted into the system or allow the program to be more robust in these situations.

8 Syntax Highlighting

We decided to incorporate syntax highlighting into Decypher as it helps distinguish and identify key elements within the language. We found it very helpful while writing the programs in Decypher. Given that syntax highlighting is specific to IDEs, we decided to support Visual Studio Code, a widely used IDE we utilised throughout our project.

Creating syntax highlighting for a new language was quite complex, requiring the development of a Visual Studio extension. To simplify the process, we used a predefined Visual Studio Code extension template provided by the Yeoman generator. This template provided us with a structured file system, including a package.json file, which takes a separate file where we define our language tokens in JSON format and use the rules to colour the syntax appropriately. After compiling and running the configuration, the currently running Visual Studio Code instance supported syntax highlighting.

We also utilised vsce to package our extension into a .vsix file, making it distributable and installable on other machines. The final stage was to upload the extension on the extensions store. However, we should have done this as we thought that was optional at this stage of development.

In Decypher, keywords are highlighted in magenta, parentheses in a yellowish hue, and identifiers, properties, and labels in bright blue. Our language's grammar determined this colour scheme. One thing to note is that the grammar does not differentiate between identifiers, properties, and labels because we treated all of them as a single type to make the interpreter easier to implement. Other symbols remain in white. An example of the syntax highlighting is provided in Appendix A.

9 String Interpolation

String interpolation could have been added to our language by introducing a syntax that allows variables or expressions to be embedded directly within a string literal. This would mean that the tokens and the grammar must be updated to reflect this and handle said cases. This feature is helpful because it simplifies creating dynamic strings, making the code more readable and maintainable. Instead of concatenating strings and variables or using complex formatting functions, users can directly insert variables or expressions into the string, improving code clarity and reducing errors.

10 Refactoring Tools

Refactoring tools help automate code restructuring processes while keeping the program's behaviour intact. To implement them, we would have needed to analyse the code to identify patterns, dependencies and some opportunities for improvement, like removing duplicate code, understanding the relationship between different code elements, etc.

We need to implement the basic functions of code refactoring, such as renaming variables. These should be done automatically so the user's experience is the best. It will save time and reduce the likelihood of introducing errors, as these are repetitive tasks.

We also need to implement interactive suggestions for code improvements. This goes hand in hand with both code suggestions and syntax highlighting, as syntax highlighting can be used to hint to the user of potential issues or suggest alternative implementations with better practices or coding standards.

We would also need to add a preview and some safety checks before applying the changes. This ensures that no compilation errors or unintended side effects come from adapting and changing the code. This will help the user make informed

decisions and will prevent accidental code breakage.

One more thing that could be added is custom refactorings. This will allow the user to have refactorings tailored to their needs or the project requirements. It makes it more flexible and allows users to automate repetitive tasks on their code.

11 Auto-completion and Code Suggestions

Auto-completion and code suggestions could have been integrated into the IDE by writing algorithms that analyze the code context and provide suggestions for completing code snippets, variable names, method names, and more. This feature is handy as it increases productivity by reducing typing errors, providing quick access to documentation, and suggesting best practices. It also makes the experience more enjoyable.

Similarly, with auto-completion, developers can write code faster, explore APIs more efficiently, and discover potential coding mistakes early in the development process. This improves overall code quality and reduces debugging time, which also contributes to the user's experience with our language.

12 Debugging and Tracing

We would need to implement two separate things to get this working in a way that is helpful for the user: query logging and visual tracing tools.

Query logging provides facilities to log queries and their execution details. This includes execution times, paths traversed, and nodes accessed. These can help the user identify performance bottlenecks or logical errors.

We need to develop visual tools that allow users to step through their queries and visualize the graph operations step-by-step. This can be useful in understanding complex joins and pathfinding operations in the graph so that if the query does not give the desired output, the user can troubleshoot and find out why.

13 Integration with Existing Tools

We must ensure that our language can be easily integrated with other popular programming languages. This can be done through well-documented and robust APIs that will allow our language to execute through these other programming languages.

We would also need to develop plugins for popular IDEs that support syntax highlighting, code completion, inline error checking, etc. This will improve the development experience.

14 Domain-Specific Optimisations

First, We need to enhance the query optimiser to recognise common patterns and structures specific to certain domains. Then, the execution path will need to be optimised accordingly.

We should also implement domain-specific indexing strategies, which help speed up the processing of queries. One example is specialised indexes for frequently queried properties or relationships.

15 Extensibility

We must allow the users to define their functions for our language's queries. This will help to encapsulate complex logic into components that are easily reusable.

We would also need to develop a plugin architecture that allows third parties to extend our language with custom syntax, functions, or new types of graph algorithms optimized for specific use cases.

16 Recursive Queries

To implement them, we need to update the tokens and the grammar to allow such queries. These queries will have a base case and a recursive step, which will then be combined, and the resulting combination will be used to look at the graph for the desired information.

17 Code Folding

This one is more complicated, but it would have been a great addition to our language. First, we must analyse the syntax so the code structure is parsed. Said structure includes the different blocks of queries. We could have written a separate lexer and parser to make this work.

Once the syntax is analysed, we would have needed to identify regions of the code that can be folded. This could have been easily done by looking for sections inside curly braces or spaces between chunks of the query so that those can be folded. It also includes queries inside queries.

After that, the UI would need to be updated so that there is a visual indicator to show the user where folding is possible. This would work such that if the user interacts with that indicator via a click with their mouse or a keyboard shortcut, the markers would collapse or expand to hide/show the corresponding code in the corresponding core regions.

Another crucial element would be persistence. The editor needs to remember the folding state of each code region to keep consistency between sessions. This could be stored in memory, and the user can adapt to other settings.

We also need to handle the nested folds such that collapsing one region does not collapse other things within said region. This is more targeted to the enjoyability of the code rather than something that would make code folding work.

Relating to enjoyability, we need to optimise this implementation by using efficient parsing, lazy loading of code regions, minimising redraws when folding/unfolding code, etc. This is so that if the user has many queries, they will not notice slowing down when using the program.

Still relating to the ease of use aspect, we could add customisable options, such as the user's ability to define custom foldable regions or adjust the folding behaviour based on their preferences. This will enhance the flexibility and usability of the feature as a whole.

18 Informative Error Messages

We decided to use the posn wrapper for our tokens and grammar. This allowed us to understand better where the code was going wrong and gave the user more detailed and helpful error messages, as the line and column of the parse error, for example, is displayed. This makes it so the user can figure out what part of the query they inputted is invalid.

19 Limitations / Improvements

One of the most significant limitations we found was time. A big learning point for us is time management and learning to use our time to its best potential. We also found that, since the bulk of this coursework was over the Easter break and everyone in the team was abroad, the communication and the motivation to do more was also a weak point. If we had more time, we would have fully developed the grammar. We would have also implemented and fully fleshed out the additional features mentioned before, as we believe that, although not a core requirement, they make the program more robust. It could positively affect the user's enjoyment overall if a more complete implementation were developed.

A Appendix A

```
1   FIND (n)
2   FROM "access.n4j"
3   WHERE (n:Visitor) OR n.age <= 25
4   RETURN n.id, n.age, n.label
```

Figure 1: Syntax Highlighting