

# COMP2212

## Programming Language Concept

**M Y K E Y**

May 2024

# 1 Introduction to The language MYKEY

MYKEY is a domain-specific query language designed specifically for querying and manipulating graph data stored in the Neo4j graph database. The syntax draws inspiration from Cypher and SQL's structure, with some elements from Java. The language and its features are tailored specifically for the tasks of analysing Neo4j files and performing specific queries and manipulations to the data, then outputting the result.

The syntax of the language is designed to be declarative, where the user specifies what data they want to retrieve or manipulate, rather than how to do it. This makes MYKEY queries easy to read and understand, even for programmers who are not experts in graph theory or database internals.

The language also provides a rich set of operations and functions for querying and manipulating graph data. This includes filtering data based on properties or labels, aggregating data, and performing graph traversal operations such as finding relationships between nodes.

Java-style comments are allowed. Programmers may use the two forward slashes (//) in their code to comment out lines. This can be done at the start of the line or inline.

MYKEY comes with its own interpreter, which lexes, parses, and type-checks both the graph data and the program before running. The interpreter will redirect the result to either std-out or std-error if an error occurs.

## 2 Language syntax and Main Features

A MYKEY query consists of a sequence of clauses, each separated by a new line. Each clause performs a specific operation, such as data retrieval or modification. The clauses are processed in the order they appear in the query. All the clauses conform to a structured syntax that follows this general pattern:

```
IMPORT "<file name>"
<Clause Name> <sequence of operations>
RETURN <identifier>
```

Below is a breakdown of the syntax:

### 2.1 IMPORT statement - IMPORT "<file name>"

**Every valid MYKEY program must begin with a single IMPORT statement** followed by a string indicating the name of the Neo4j file you wish to query. This file must be in the same directory with the program code file (.gql) and the interpreter (Gql.hs). Only one IMPORT statement is allowed per MYKEY program.

### 2.2 Query clauses - <Clause Name> <sequence of operations>

The main body of MYKEY's query consists of a sequence of clauses. There are **six** different clauses that can be written before the RETURN clause:

Clause name	Description of features
MATCH	Used to specify a pattern of nodes and relationships to search for in the database. It's similar to the SELECT statement in SQL but is used to describe how different nodes are related.
OPTION MATCH	Use filtered nodes or relationships to make new matches, or re-match new nodes or relationships
WHERE	Adds constraints to the patterns in a MATCH clause.
MERGE	Ensures that a relationship exists in the graph by creating it if it does not already exist.
SET	Add or update properties on nodes and relationships.
DELETE	Remove nodes and relationships

### 2.3 RETURN clause - RETURN <identifier>

Specifies what data to return from the query. **Every valid MYKEY program must only have one single RETURN clause and it needs to be at the end of the whole query.**

### 2.4 Expressions and Operations

Within **WHERE** and **SET** clause, several expressions and function operations are allowed to help with filtering or updating nodes and relationships. Expressions can have other expressions nested inside them as long they are the correct expected type. The types of expressions and functions are explained in detail below.

### 2.4.1 Arithmetic expressions

The arithmetic expression always returns a TypeInt type. It allows operations on numbers: addition, subtraction, division, and multiplication.

### 2.4.2 Boolean expressions

The boolean expression always returns a TypeBool type. It allows boolean arithmetic operations: comparing arithmetic expressions, boolean algebra operations (e.g. “AND”, “OR”) and null checks (e.g. ”IS NOT NULL”).

### 2.4.3 Complex expressions

**EACH <Arithmetic expression> BY <Relationship> <Direction of relationship>** - Perform arithmetic or comparison operations on each node and only the nodes related to it. The same applies to relationships.

**FILL <Property> WITH <value>** - Fill the property of an identifier with a specific value (if the property does not exist then create one).

**NOT <Label> IN <identifier>** - Filter out nodes that contains certain label.

### 2.4.4 Functions

MYKEY contains a useful set of operations and functions for querying and manipulating graph data. Below is a summary:

Function syntax	Description of features
id()	input a specific node identifier (e.g. n) to get the id of all nodes bind with that identifier
labels()	input a specific node identifier (e.g. n) to get the label of all nodes bind with that identifier
collect()	input 2 nodes identifier separated by ”+” (e.g. n + m) to get a union collection of all nodes that bind with the 2 node identifier
intersect()	input 2 nodes identifier separated by ”+” (e.g. n + m) to get an intersection of all nodes that bind with the 2 node identifier
exists()	input a specific node identifier with one of its properties (e.g. n.age) to check whether this property exists for every node that binds with this node identifier in the graph database

### 2.4.5 Null checks

Null checks are operations that can be performed on a node’s property (e.g. n.age IS NOT NULL). There are two types of null check operations:

**IS NOT NULL** - Used to filter out records where the specified property is null, retaining only those nodes or relationships where the specified property has a valid value.

**IS NULL** - Used to filter out records where the specified property is not null, retaining only those nodes or relationships where the specified property is null.

## 2.5 Lexical Rules

MYKEY’S tokens consist of 4 parts: **identifiers**, **literals**, **keywords**, and **symbols**.

**Identifiers** are names used to label nodes or relationships, properties, or node’s label/relationship’s type. They are unquoted and must begin with a letter followed by alphanumeric characters.

**Literals** can be number, string, Boolean, or null. Strings are enclosed in double quotes.

**Keywords** are reserved words that have special meaning, such as MATCH, RETURN, WHERE, and so on. These are always interpreted in their specific context within the query structure.

**Symbols** Symbols include punctuation and operators used in expressions, like commas (,), parentheses (()), brackets ([]), and curly braces ({}).

Expressions in MYKEY consist of combinations of identifiers, literals, function calls, and operators. These expressions can be used to perform mathematical calculations, string operations, logical comparisons, and more.

### 2.5.1 White space and Comments:

White spaces are ignored outside of literals and can include spaces, tabs, and line breaks. It is used to separate tokens but does not affect the interpretation of the query unless within a string literal.

Comments can be inserted in queries using two forward slashes (`//`). Anything following these slashes up to the end of the line is considered a comment and is ignored by the parser.

### 2.5.2 Case Sensitivity:

MYKEY is case-sensitive when it comes to identifiers and keywords. For example, `MATCH` and `match` are **NOT** treated the same, identifiers like `Person` and `person` would refer to different entities or properties if used in the same context.

## 2.6 Static Scoping

MYKEY uses static scoping, which means that the scope of variables is determined at the time the query is written, not at runtime. Variables are only accessible within the block where they are declared and any subsequent blocks that are logically part of the same query execution path. Once a variable is introduced using a clause like `MATCH`, or `OPTION MATCH`, it remains accessible throughout the query.

## 3 Execution model

Before running, the MYKEY program first get separated by the interpreter into **two** parts: the `IMPORT` statement and the rest of the program. There are several states that the interpreter will enter during compile-time and run-time.

### 3.1 File analysing state

The `IMPORT` statement part is analysed and the **file name** is extracted from the `IMPORT` statement (e.g. `"access.n4j"`). This file name will be used to find the target file within the current directory (file-related errors may occur during this state).

Once the file is successfully found, the content within the target file will be read and analysed by numerous compile-time units including a **file lexer** unit, a **file parser** unit, a **environment building** unit, and a **graph conversion** unit.

Firstly, the file lexer removes comments and white spaces from the file content and returns a parse tree made of identified tokens. The file parser then generates an abstract syntax tree based on the file grammar, which uses commas in line breaks to distinguish different headers, nodes, and relationships (for more detail, please refer to the file BNF in the Appendix). In case a token is unidentified, or the sentence is not part of the grammar a parse error is printed to `stderr` in this format:

```
Parse error at line:column (<line>, <column>)
```

#### 3.1.1 Environment building unit

After an abstract syntax tree has been generated, this AST will be passed into the environment building unit and generate a `TypeEnvironment`. `TypeEnvironment` is defined as below:

```
type TypeEnvironment = [(String, Maybe DataType)]
```

The `TypeEnvironment` generated contains type information about all the property names, node labels, and relationship types. For example, `("age", TypeInt)` indicates that the `"age"` field has the type of integer, `("Staff", TypeLabel)` indicates that the type of `Label`. This `TypeEnvironment` will be used later in the type-checking state to ensure type correctness within the program code.

#### 3.1.2 Graph conversion unit

The AST will also be passed into the graph conversion unit where the AST will be analysed and turned into a graph data structure which is defined as below:

```
data Graph = Graph
{ graphNodes :: Map.Map String Node,
  graphRelationships :: Map.Map String Relationship
}
```

The graph consists of two main components: `graphNodes` and `graphRelationships`. Each component is structured as a map, where each entry comprises a pair of a string and a `Node` or `Relationship`. The strings serve as unique identifiers: for nodes, they are simply the `<id>`; for relationships, they follow the format `<start_id>-<end_id>`, reflecting the IDs of the start and end nodes.

The Node and Relationship are the data structures defined to help with storing key information such as IDs, property values and their type, node labels, or relationship types (to see the full structure of the graph, please go to the Graph Data Structure section in the Appendix).

This graph conversion is very key as the graph generated will later be used as the source for the query. This design ensures efficient retrieval and manipulation of nodes and relationships, facilitating complex graph-based operations and analytics.

## 3.2 Program analysing state

The rest of the program is analysed separately with its program lexer, program parser, and type checker. The lexer removes comments and white spaces from the program and returns a parse tree made of identified tokens. The parser then looks for valid sentences in the MYKEY grammar to generate an abstract syntax tree (for more detail please refer to the BNF section in the Appendix). In case a token is unidentified, or the sentence is not part of the grammar a parse error is printed to std-err.

### 3.2.1 Type checking

The type checker is then run at compile time to make sure the program is well-typed and reports the type returned by the program. This type will either be `Left <error>` which indicates there's an error with the program's type, or `Right <type>` which indicates that the program has the right type.

The type checker will check the types of all expressions and operations using the pre-generated environment which contains type information about the graph database imported. This ensures that the types match with the expected type. For example, 'AND' boolean expression expects boolean inputs on both sides, `25 AND n.age > 5` will report a type checking error. Several types of errors can occur when type-checking is done, we will discuss them in a later section. No type errors can occur at runtime.

## 3.3 Program executing state

After the program passes the type checking, the result of parsing is passed to the run-time unit for execution, along with the pre-generated graph. The executor uses a series of evaluation functions to evaluate each query clause in order. The evaluation functions pattern matches the clauses and their operations/expressions and then performs the query by updating or manipulating the execution environment and the graph. This environment is initialised during the MATCH clause and is defined as below:

```
type Environment = Map.Map String (Either [Node] [Relationship])
```

The execution environment keeps track of variable bindings. Depending on the program, either the updated environment or the updated graph will be returned as the result of the query which is then passed into a CSV conversion unit that converts the data structure into CVS form to be outputted onto std-out. Multiple run-time errors can occur at this stage and we will cover them in the next sections.

## 4 Additional features

We created many additional features for our language. For example, **syntax sugar** and **syntax highlight** for programmer convenience, **type checking** and informative **error messages**.

### 4.1 Syntax sugar

**Pattern Matching** - MYKEY uses patterns resembling ASCII art to describe the structure within a graph. For example, `(a:Person)-[:KNOWS]->(b)` denotes finding a KNOWS relationship from a node labelled Person (node a) to any other node (b).

**Chain Calls** - MYKEY allows the construction of complex queries through chaining calls, making the writing of query statements more coherent and intuitive. For example, MYKEY allows the use of `","` within MATCH to achieve multiple matches rather than creating multiple MATCH clauses.

**OPTION** - This feature allows for optional matching within a query, without affecting the execution of the overall query.

**Omitting Node and Relationship Types** - In some cases, if the context in the query is already clear, you can omit the types of nodes or the names of relationships: `MATCH (p)-[:FRIENDS_WITH]->()`. This matches all people who have friends, without specifying the type of the friend nodes.

**Simplified Usage of the EXISTS Function** - The exists function can be used to check whether a specific property or relationship exists, For example: `exists(p.email)` matches all people who have an email address.

## 4.2 Type checking

MYKEY checks the type of every expression and operation matches with expected type before run-time to ensure that the program is well-typed before execution. This was explained in detail in the section above.

## 4.3 Error messages

During compilation and execution, several types of errors can occur and the interpreter will produce specific error messages on standard error (std-err). Below is a breakdown:

Compile-Time Errors	Description of the Error
Parse error at line:column ( <code>&lt;line&gt;</code> , <code>&lt;column&gt;</code> )	When syntax is incorrectly written (e.g., contains characters/sentences not part of the language grammar), a parse error message will be printed, stating the row and column in the file where the error occurred.
Where condition must be TypeBool	The condition specified after the WHERE clause does not return a TypeBool and caused this type error.
Identifier <code>&lt;identifier&gt;</code> exists but does not have <code>&lt;Type&gt;</code>	The identifier specified can be found in the environment but does not match with the type it claims to be.
Identifier <code>&lt;identifier&gt;</code> not found in environment	The identifier specified cannot be found in the environment.
Arithmetic operations require integer operands	Arithmetic operations were attempted with operands that are not integers.
Unknown identifier: <code>&lt;identifier&gt;</code>	Unable to recognise the given identifier.
Identifier does not correspond to a type: <code>&lt;identifier&gt;</code>	The identifier specified can be found in the environment but does not have a corresponding type.
Logical operations require boolean operands	Logical operations were given operands that are not type boolean.
Property not found: <code>&lt;identifier&gt;</code>	The property specified cannot be found in the environment.
Label not found: <code>&lt;identifier&gt;</code>	The label specified cannot be found in the environment.
Type not found: <code>&lt;identifier&gt;</code>	The relationship type specified cannot be found in the environment.
Invalid comparison types	Comparison expressions were given invalid operands (e.g. integer compared to a boolean is invalid in the MYKEY program).
Type mismatch in comparison operation	The types of operands within the comparison operation do not match.
MissMatch Regex	The regular expression was defined incorrectly. It needs to be in the form of <code>'^[a-zA-Z0-9_]*.'</code> . Where <code>[a-zA-Z0-9]*</code> is any alphanumeric string.
The <code>&lt;function.name&gt;</code> function requires a node identifier.	Many functions can only accept node identifiers. Other types of identifiers will cause this type-check error.

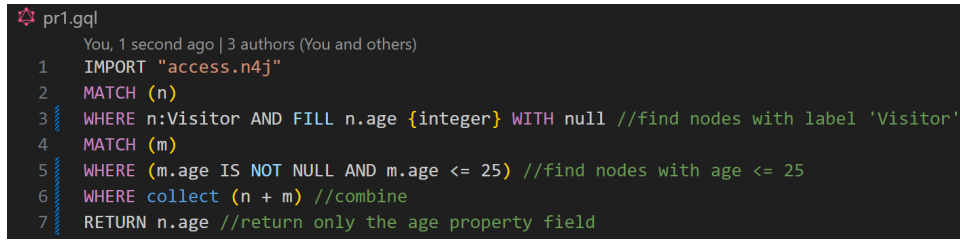
Run-Time Errors	Description of the Error
File Not Found Error	Target Neo4j file does not exist in the current directory
Empty file Error	The target Neo4j file's content is empty
Import Statement Error	No import statement found or improperly formatted import statement. Every valid MYKEY program must start with a single IMPORT statement in the correct format: <code>IMPORT "&lt;file name&gt;"</code> , where the file name is the name of the target Neo4j file.

## 4.4 Syntax Highlight

Due to the syntax highlight being hard to build in Haskell, our group decided only to build it in the VS code environment. We designed it with Node.js and Yeoman to generate a VS code extension. We highlighted input file names, syntax keywords, and comments with different colours (please see the Highlight section in the Appendix for the screenshot). We generated a . VSIX file through which all our group members can import MYKAY syntax highlight.

## 5 Appendix

### 5.1 Highlight



```
pr1.gql
You, 1 second ago | 3 authors (You and others)
1  IMPORT "access.n4j"
2  MATCH (n)
3  WHERE n:Visitor AND FILL n.age {integer} WITH null //find nodes with label 'Visitor'
4  MATCH (m)
5  WHERE (m.age IS NOT NULL AND m.age <= 25) //find nodes with age <= 25
6  WHERE collect (n + m) //combine
7  RETURN n.age //return only the age property field
```

Figure 1: Syntax highlighted MYKEY program within VS code

### 5.2 File BNF

```
<Input> ::= <InputItems>

<InputItems> ::= <InputItem> | <InputItems> <InputItem>

<InputItem> ::= <NodesRecord> | <RelationshipsRecord>

<NodesRecord> ::= <NodeHeader> <Nodes>

<RelationshipsRecord> ::= <RelationshipsHeader> <Relationships>

<NodeHeader> ::= ID "," LABEL | ID "," <PropertyFields> "," LABEL | ID "," <
    PropertyFields>

<Nodes> ::= | <Node> <Nodes>

<Node> ::= EntryValue | EntryValue "," <Node>

<RelationshipsHeader> ::= START_ID "," END_ID "," TYPE | START_ID "," <PropertyFields> ","
    END_ID "," TYPE

<Relationships> ::= | <Relationship> <Relationships>

<Relationship> ::= EntryValue | EntryValue "," <Relationship>

<PropertyFields> ::= <PropertyField> | <PropertyFields> "," <PropertyField>

<PropertyField> : EntryValue ":" <FieldType>

<FieldType> : string | integer | boolean
```

### 5.3 BNF

```
<Query> ::= QueryItems

<QueryItems> ::= <QueryItem> | <QueryItems> <QueryItem>
-----
<QueryItem> ::= <MatchClause> | <ReturnClause> | <WhereClause> | <MergeClause> | <
    SetClause> | <DeleteClause>
-----
<ReturnClause> ::= RETURN <ReturnItems> | RETURN <DistinctClause> | RETURN

<ReturnItems> ::= <ReturnItem> | <ReturnItem> "," <ReturnItems>
<ReturnItem> ::= <Identifier> "." <Identifier> | <Identifier>
-----
<MatchClause> ::= MATCH <MatchItems> | OPTION MATCH <OptionalMatchItems>

<MatchItems> ::= <MatchItem> | <MatchItems> "," <MatchItem>

<MatchItem> ::= "(" <Identifier> ")" | "(" <Identifier> ")" <Relationships> "(" <
    Identifier> ")" | <Relationships> "(" <OptionalIdentifiers> ")" | "(" ":" <Identifier>
```

```

    ")" | "(" <Pair> ")" | "(" <Pair> ")" <Relationships> "(" <Pair> ")" <Relationships>
    "(" <Pair> ")" | "(" <Identifier> ")" <Relationships> "(" <Identifier> ")" <
Relationships> "(" <Identifier> ")"

<OptionalMatchItems> ::= <OptionalMatchItem> | <OptionalMatchItems> "," <OptionalMatchItem
>

<OptionalMatchItem> ::= "(" <Identifier> ")" <Relationships> "(" <Identifier> ")" | "(" <
Identifier> ")" | "(" <Pair> ")"

<OptionalIdentifiers> ::= <Identifier> | epsilon

<Relationships> ::= "-[" <Identifier> "]"->" | "-[" ":" <Identifier> "]"->" | "-[" <
Identifier> ":" <Identifier> "]"->" | "<-[" <Identifier> "]"-" | "<-[" ":" <Identifier>
"]-" | "<-[" <Identifier> ":" <Identifier> "]"-"
-----
<WhereClause> ::= WHERE <Condition>

<Condition> ::= <Condition> OR <Condition> | <Condition> AND <Condition> | "(" <Condition>
)" | <Predicate> | <FunctionCall> | FILL <SimplePredicate>

<Predicate> ::= <Identifier> ":" <Identifier> | <Identifier> "." <Identifier> | <
Identifier> "." <Identifier> <NullChecks> | NOT <Identifier> IN <FunctionCall> | <
Predicate> <ComparisonOp> <Attribute> | <Predicate> <ComparisonOp> <SimplePredicate> |
EACH <Predicate> <ComparisonOp> <SimplePredicate> BY <Identifier> <Direction> | EACH
<Predicate> <Operations> <SimplePredicate> BY <Identifier> <Direction> | <Predicate>
"=~" <Regex>

<NullChecks> : isNotNull | isNull

<ComparisonOp> ::= "<=" | ">=" | "<" | ">" | "==" | "!="

<Attribute> ::= <Number> | <Str> | true | false | null

<SimplePredicate> ::= <Identifier> ":" <Identifier> | <Identifier> "." <Identifier>

<MatchValue> ::= "=~"
-----
<MergeClause> ::= MERGE <MergeItems> | MERGE BY "{" <Identifier>"," <Identifier> "}"
Direction

<MergeItems> ::= : "(" <Identifier> ")" <Relationships> "(" <Identifier> ")" | "(" <
Identifier> ")" "-[" <NodeClause> "]"->" "(" <Identifier> ")" | "(" ":" <Identifier> ")"
" <Relationships> "(" ":" <Identifier> ")"

<NodeClause> ::= "(" <Identifier> ":" <Identifier> "{" <Pair> "," <Pair> "}" ")" | "(" <
Identifier> ":" <Identifier> "{" <Pair> "}" ")"

<Pair> ::= <Identifier> ":" <Identifier>
-----
<SetClause> ::= SET <SetItems>

<SetItems> ::= <SetItem> | <SetItems> "," <SetItem>

<SetItem> ::= <Arithmetic>

<Arithmetic> ::= <Value> <Operations> <Value> | <Arithmetic> <Operations> <Value>

<Operations> ::= "+" | "-" | "*" | "/" | "="

<Value> : <Identifier> "." <Identifier>
-----
<DeleteClause> ::= DELETE <DeleteItems>

<DeleteItems> ::= <DeleteItem> | <DeleteItems> "," <DeleteItem>

<DeleteItem> ::= <Identifier>
-----
<FunctionCall> ::= id "(" <Identifier> ")" | labels "(" <Identifier> ")" | collect "(" <

```

```
Identifier> ")" | collect "(" <Identifier> "+" <Identifier>")" | exists "(" <Identifier>
> "." <Identifier> ")" | inter "(" <Identifier> "+" <Identifier> ")"
```

```
<Identifier> ::= Identifiers
```

```
<Direction> ::= Positive | Negative
```

## 5.4 Graph Data Structure

```
data PropertyValue
= PropInt Int
| PropString String
| PropBool Bool
| PropIntNull
| PropStrNull
| PropBoolNull
deriving (Show, Eq, Ord)
```

```
data Node = DataNode
{ nodeId :: String,
  nodeLabels :: [String],
  nodeProperties :: Map.Map String PropertyValue
}
deriving (Show, Eq)
```

```
data Relationship = DataRelationship
{ startNodeId :: String,
  endNodeId :: String,
  relationshipType :: String,
  relationshipProperties :: Map.Map String PropertyValue
}
deriving (Show, Eq)
```

```
data Graph = Graph
{ graphNodes :: Map.Map String Node,
  graphRelationships :: Map.Map String Relationship
}
deriving (Show, Eq)
```