Technical report, Åland University of applied sciences, educational program for IT24

# FROM TEXT TO DIAGRAMS

## - Design and implementation of a lexer and parser for a custom diagram language.

Björn Granholm


<2025:05>

Supervisor: Tiina Mäkinen

# TABLE OF CONTENTS.

# 1. INTRODUCTION

## 1.1 Background

In systems that interpret or analyze textual input, there is a need to convert human-readable text to structured representations that can be processed by machines. This process often begins with **lexical analysis** (lexing)(Improve, 2015a) and **syntactic analysis** (parsing)(Improve, 2015b). A **lexer** scans input strings into tokens, identifying meaningful elements like keywords and symbols. **Parsers** then use these tokens to build structured data, often in the form of **abstract syntax trees (**AST)(Wikipedia contributors, 2025a) or domain-specific objects.

The project presented in this report focuses on a **domain-specific language** (DSL)(Wikipedia contributors, 2025b) designed to describe diagrams. This language allows users to define nodes, edges, attributes(such as color and layout) using a readable textual syntax. The lexer and parser are implemented in Go(*Documentation*, n.d.) and are designed to convert this syntax into a structured diagram object, suitable for visualization.

This lexer and parser were originally developed as part of the Laboration 1 course and are now used as the basis for this technical report.

## 1.2 Purpose

The purpose of this report is to examine the implementation of a lexer and parser for a DSL designed to define diagrams. By analyzing how text input is converted to structured representations, the report aims to highlight key aspects of language design lexical analysis and syntactic parsing.

The lexer and parser here serve as an educational tool for understanding how languages can be processed programmatically and this report covers how these components work together to transform  code into a format for visualization.

```
Shell
diagram flowchart (layout=vertical) {
       node A "Start" (color=lightgreen, shape=ellipse, text=black)
       node B "Process"
       node C "END"

       A -> B "Step 1" (color=red, width=3)
       B -> C "Step 2"
}
```

*Figure 1. Code  of an example diagram (Granholm, n.d.-c)*

# 2. LEXER DESIGN

The lexer is responsible for scanning the raw input string and converting it into a sequence of tokens that represent the basic elements of the diagram language Each token corresponds to a meaningful unit, such as a keyword, identifier, symbol or literal value.

## 2.1 Token types

The lexer categorizes characters and strings into several predefined token types:

- `TOKEN_KEYWORD` Reserved for words like diagram and node.
- `TOKEN_IDENTIFIER` Names of diagrams, node IDs or values.
- `TOKEN_STRING` Text enclosed in quotation marks.
- `TOKEN_ARROW` The arrow used to define connections between nodes.
- `TOKEN_LBRACE` Brace marking the beginning.
- `TOKEN_RBRACE` Brace marking the end.
- `TOKEN_SYMBOL` Other single-character symbols such as `=`, `(`, `)`, and `,`.
- `TOKEN_EOF` Signifies the end of the input.

This categorization enables the parser to interpret the input with a clear syntactic structure.(Granholm, n.d.-b)

## 2.2 Tokenization logic

The lexer operates using a loop that iterates through the input string character by character. It uses Unicode-aware functions to identify letters, digits and whitespace. Based on the current character, the lexer decides how to group characters into a token.

Key aspects of the tokenization process:

- Whitespace is ignored.
- Words are identified as either **keywords** or **identifiers**, depending on whether they match predefined words.
- String literals are captured between quotation marks.
- Multi-character symbols like `->` are detected using lookahead.
- Single-character symbols are handled explicitly.

Each time a token is recognized, it is added to a token list which is returned to the parser once the input has been processed.

```
Shell

0: diagram (KEYWORD)
1: flowchart (IDENTIFIER)
2: ( (SYMBOL)
3: layout (IDENTIFIER)
4: = (SYMBOL)
5: vertical (IDENTIFIER)
6: ) (SYMBOL)
7: { (LBRACE)
8: node (KEYWORD)
9: A (IDENTIFIER)
10: Start (STRING)
11: ( (SYMBOL)
12: color (IDENTIFIER)
13: = (SYMBOL)
14: lightgreen (IDENTIFIER)
15: , (SYMBOL)
16: shape (IDENTIFIER)
17: = (SYMBOL)
18: ellipse (IDENTIFIER)
19: , (SYMBOL)
20: text (IDENTIFIER)
21: = (SYMBOL)
22: black (IDENTIFIER)
```

```
23: ) (SYMBOL)
24: node (KEYWORD)
25: B (IDENTIFIER)
26: Process (STRING)
27: node (KEYWORD)
28: C (IDENTIFIER)
29: END (STRING)
30: A (IDENTIFIER)
31: -> (ARROW)
32: B (IDENTIFIER)
33: Step 1 (STRING)
34: ( (SYMBOL)
35: color (IDENTIFIER)
36: = (SYMBOL)
37: red (IDENTIFIER)
38: , (SYMBOL)
39: width (IDENTIFIER)
40: = (SYMBOL)
41: 3 (IDENTIFIER)
42: ) (SYMBOL)
43: B (IDENTIFIER)
44: -> (ARROW)
45: C (IDENTIFIER)
46: Step 2 (STRING)
47: } (RBRACE)
48:  (EOF)
```

*Figure 2. Terminal output of the diagram in figure 1 tokenized.*

# 3. PARSER DESIGN

The parser takes the list of tokens produced by the lexer and constructs a structured representation of the input diagram. It does so by analyzing the sequence of tokens according to the rules of the DSL. The parser is implemented using a recursive descent strategy(Wikipedia contributors, 2024), where each syntactic rule is represented by a function.

## 3.1 Overall structure

The parser is encapsulated in a struct that maintains the list of tokens and a pointer to the current position in the token stream. Parsing begins with a call to the **Parse()** function, which delegates to **parseDiagram()** the main entry point for diagram definitions.(Granholm, n.d.-a)

The parser progresses by checking the current token, matching expected types or values; advancing the pointer. If a token does not match the expected pattern an error message is returned.

## 3.2 Grammar overview

The expected syntax follow this general structure:

```
Shell
diagram <type> (optional attributes) {
    node <id> "<label>" (optional node attributes)
    <id> -> <id> "<optional edge label>" (optional edge attributes)
}
```

*Figure 3. Structure of a diagram in the language.*

Each part of this structure is handled by separate parsing logic:

- **Diagram declaration:** Expects the diagram keyword, type,  optional attributes and a block enclosed with braces.
- **Nodes:** Defined using the node keyword followed by an ID, a label in quotes and an optional style in parentheses.
- **Edges:** Defined by using the format `<from> -> <to>`, optionally followed by a label in quotes and attributes such as color or width.

## 3.3 Attribute parsing

Attributes for both nodes and edges are parsed using key-value pairs within parentheses. The parser checks for the opening parenthesis, then iteratively extracts attribute name-value pairs and delimited by commas.
The parser uses default values for optional attributes if they are not explicitly provided.

## 3.4 Error handling

The parser includes basic error handling to detect:

- Unexpected token types or missing expected tokens.

- Invalid diagram types or malformed attribute syntax.
- Premature end of input.

# 4. IMPLEMENTATION HIGHLIGHTS

This section outlines specific aspects of the lexer and parser implementation that demonstrate design choices, structures and flexibility for future extensions.

## 4.1 Unicode-aware lexical analysis

The lexer uses Go's built in unicode package to handle characters in a robust and language-agnostic way. This allows the lexer to correctly recognize  letters and digits even beyond the basic ASCII range, making it more resilient to diverse input.

```Go
if unicode.IsLetter(c) { ... }
if unicode.IsDigit(c) { ... }
```

*Figure 4. Examples of the built in functions using(Unicode, n.d.)*

## 4.2 Minimalistic yet expressive token types

The token types are intentionally kept minimal, just enough to distinguish between keywords, identifiers, strings, symbols and structural elements like arrows and braces.
This design reduces the complexity while still supporting a fully functional DSL for diagrams.

## 4.3 Support for optional attributes with defaults

Both node and edge definitions allow for optional attributes. The parser provides default when these attributes are omitted allowing users to write simpler, more concise input without sacrificing flexibility.(Granholm, n.d.-a)

```Go
color := "#e0f7fa"
textColor := "#004d40"
shape := "rect"
```

*Figure 5. Diagram default options for a parsed node (Granholm, n.d.-a)*

## 4.4 Recursive descent parsing

The parser is structured using recursive descent, where each grammatical construct is mapped to a dedicated function. This makes it easy to follow and modify, making it ideal for future enhancements.

## 4.5 Descriptive error reporting

Error messages are constructed using **Errorf()**(*Fmt Package - Fmt - Go Packages*, n.d.) from the built-in fmt package and describe what was expected at a given point in the input. This is helpful for both debugging during development and for user-facing feedback.

```Go
if p.currentToken().Value != "=" {
         return d, fmt.Errorf("expected '=' after attributename")
 }
```

*Figure 6. Example of error handling.(Granholm, n.d.-a)*

## 4.6 Token driven control flow

The parser avoids any external grammar engine and relies purely on token-driven control flow. It uses help functions like **match()** and **advance()** to move through the token stream clearly and predictably.

# 5. DISCUSSION

The implementation of the lexer and parser proved to be both efficient and flexible for the scope of the project. The decision to use a handmade parser allowed for clear and maintainable code, making it easier to trace and understand the parsing logic.

## 5.1 Strengths

- **Clarity and simplicity**: The code is easy to follow and well-structured.
- **Extensibility**: Adding support for new attributes or diagram features would require only minor additions to the existing parsing logic.
- **Error reporting**: The parser provides helpful error messages, which aid both development and user experience when handling incorrect input.

## 5.2 Challenges

- **Error recovery**: While the parser detects and reports errors clearly, it does not attempt to recover. In a more advanced implementation, partial recovery could improve resilience**.**
- **Manual token handling**: The use of explicit token-by-token control is straightforward but can become verbose at a larger scale.

## 5.3 Opportunities for improvement

- Introducing more robust unit tests for edge cases and malformed input would ensure long term sustainability
- Implementing support for comments or whitespace-only lines would make the language more user-friendly.

Overall the lexer and parser performs well within their scope and serve as a foundation for understanding text processing in DSL.

# 6. CONCLUSION

This report has explored the design and implementation of a lexer and parser for a domain-specific language used to define diagrams. Starting with character-based input, the lexer successfully breaks the text into tokens while the parser interprets these tokens to construct a structured diagram consisting of nodes, edges and optional attributes.
Through the use of Unicode character handling and recursive descent parsing the implementation is balanced between simplicity and expressiveness. The system supports flexible syntax with optional styling features making it suitable for lightweight diagram definition tasks.
The project demonstrates how custom language tools can be built from the ground up using basic programming constructs. It also highlights the educational value of implementing language processors manually, offering insight into the core principles behind modern programming tools.

# REFERENCES


*Documentation*. (n.d.). Retrieved October 9, 2024, from https://go.dev/doc/

*fmt package - fmt - Go Packages*. (n.d.). Retrieved May 14, 2025, from https://pkg.go.dev/fmt

Granholm, B. (n.d.-a). *interpreter/parser.go at main · BeornG/LexerParserReferences*. Github. Retrieved May 14, 2025, from https://github.com/BeornG/LexerParserReferences/blob/main/interpreter/parser.go

Granholm, B. (n.d.-b). *interpreter/types.go at main · BeornG/LexerParserReferences*. Github. Retrieved May 14, 2025, from https://github.com/BeornG/LexerParserReferences/blob/main/interpreter/types.go

Granholm, B. (n.d.-c). *LexerParserReferences*. Github. Retrieved May 14, 2025, from https://github.com/BeornG/LexerParserReferences

Improve, K. F. (2015a, July 13). *Introduction of lexical analysis*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

Improve, K. F. (2015b, September 22). *Introduction to syntax analysis in compiler design*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/

*Unicode*. (n.d.). Retrieved May 14, 2025, from https://pkg.go.dev/unicode

Wikipedia contributors. (2024, October 25). *Recursive descent parser*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Recursive_descent_parser&oldid=1253332543

Wikipedia contributors. (2025a, March 14). *Abstract syntax tree*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1280483463

Wikipedia contributors. (2025b, April 16). *Domain-specific language*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=1285942317