Software Requirements Specification

for

Domain Specific Language with IDE for Commision Plan Designers

Version 2.0 approved

Prepared by
19z325 - Kavin Manikandan
20Z302 - Abinay
20Z345 - Shruthi Uday
20Z357 - Anukul Singh
20Z362 - Irfan

Guided by : Dr. Kavitha C
Assistant Professor (Sl.Gr.)
Department of Computer Science & Engineering

_____

Signature of the Guide

Contents

Revision History

Name	Date	Reason For Changes	Version
ip-srs-v1	10/2/23	First Version	1
ip-srs-v2	13/03/23	Formatting	2

1. Requirements

A domain specific language(DSL) with an integrated development environment(IDE) for a commission plan designer would provide a way to create and modify commission plans using a specialized language that is tailored to the specific needs of the commission plan designer.

1.1 Hardware Requirements

A pc/laptop with stable internet connection which has the following specifications:

- Processor: A fast, multi-core processor would be recommended to support the computational demands of the IDE and DSL.
- Memory: A minimum of 4 GB of RAM would be required, with 8 GB or more recommended for larger, more complex commission plan designs.
- Storage: A solid-state drive (SSD) or hard disk drive (HDD) with a minimum of 250 GB of storage capacity would be required, with more storage recommended for larger projects or if the IDE includes version control functionality.
- Display: A high-resolution display with a minimum resolution of 1920x1080
- Peripherals: A keyboard and mouse or trackpad would be required for input.

1.2 Software Requirements:

- Operating System: A modern operating system such as Windows 10, macOS, or Linux would be required.
- Development Environment: VisualStudio Code
- Dependent Libraries: The implementation may require several third-party libraries and frameworks, such as a lexer/parser generator library for the DSL, a GUI library for the IDE, and a database driver library to interface with the DBMS.

2. Software Requirement Specifications

2.1 Introduction

The purpose of this software requirement specification (SRS) is to define the functional and non-functional requirements for a domain-specific language (DSL) built using ANTLR lexer and parser for commission plans. The DSL will allow users to define commission plans using infix expressions and if-else conditions. Additionally, an integrated development environment (IDE) will be developed to provide a user-friendly interface for writing, testing, and debugging commission plans. The DSL should also support infix to prefix conversion and JSON conversion.

2.2 Functional Requirements

2.2.1 Commission Plan Expression Syntax

- The DSL should allow users to define commission plans using infix expressions. The syntax should include the following:
- Basic arithmetic operations (e.g., addition, subtraction, multiplication, division)
- Parentheses for grouping expressions
- Variables for referring to sales data (e.g., sales amount, sales quantity)
- Comparison operators (e.g., greater than, less than, equal to)
- Logical operators (e.g., AND, OR, NOT)

2.2.2 If-Else Condition Syntax

The DSL should allow users to define if-else conditions. The syntax should include the following:

- The 'if' keyword
- Comparison expressions using the syntax defined

- The 'then' keyword
- The commission plan expression to be executed if the condition is true
- The 'else' keyword (optional)
- The commission plan expression to be executed if the condition is false (optional)

2.2.3 Infix to Prefix Converter

- The infix to prefix converter will convert the commission plan expression from infix notation to prefix notation.
- The converter will use a stack-based algorithm to traverse the expression in infix notation and generate the equivalent expression in prefix notation.
- The output of the converter will be used as input for the JSON converter component.

2.2.4 Lexer and Parser

- The lexer will tokenize the input commission plan expression into a stream of tokens.
- The parser will generate a parse tree from the stream of tokens, based on the commission plan language grammar.
- The lexer and parser will work together to ensure that the commission plan expression is syntactically valid and can be parsed correctly.
- The parse tree generated by the parser will be used as input for the infix to prefix converter component.

2.2.5 JSON Converter

- The JSON converter will convert the commission plan expression and structure to JSON format.

- The JSON format will include the commission plan expression in prefix notation, the commission plan structure, and the variables used in the expression.
- The output of the JSON converter will be used by the IDE component for displaying the commission plan structure and variables and for evaluation by the backend.

2.2.6 IDE Features

The IDE should provide the following features to the users:

- Syntax highlighting and code completion for commission plans
- The IDE will provide a graphical user interface for writing, testing, and debugging commission plans.
- The IDE will communicate with the lexer, parser, infix to prefix converter, and JSON converter components to provide real-time syntax highlighting, code completion, and error highlighting.
- When a user enters a commission plan expression, the IDE will send it to the lexer component for tokenization.
- The lexer will then send the token stream to the parser component for generating a parse tree.
- The parse tree will be used by the infix to prefix converter component to generate the equivalent expression in prefix notation.
- The output of the JSON converter will be displayed in the IDE.
- The IDE will allow users to set breakpoints, inspect variables, and step through commission plan expressions during debugging.
- The IDE will integrate with source control systems, such as Git, for versioning and collaboration.
- The IDE will allow users to export and import commission plans in a variety of formats, such as CSV, JSON, and XML.

2.2.7 Error Handling

The DSL should provide informative error messages when users provide invalid input. The error messages should help users identify and correct errors in their commission plans.

2.3 Non-Functional Requirements

2.3.1 Performance

The DSL and IDE should be designed to handle large datasets of sales data efficiently. The time taken to evaluate a commission plan expression or if-else condition should be reasonable even for large datasets.

2.3.2 Security

The DSL and IDE should be designed with security in mind. They should not allow users to inject malicious code or access data that they should not have access to.

2.3.3 Usability

The DSL and IDE should be user-friendly and easy to learn. The syntax should be intuitive, and the error messages should be informative.

2.3.4 Conclusion

The DSL built using ANTLR lexer and parser for commission plans with infix expressions and if-else conditions should meet the functional and non-functional requirements defined in this software requirement specification. Additionally, the IDE should provide a user-friendly interface for writing, testing, and debugging

2.4. System Design

2.4.1 Introduction

The purpose of this system design is to provide an overview of the architecture for a domain-specific language (DSL) built using ANTLR lexer and parser for commission plans, and an integrated development environment (IDE) for writing, testing, and debugging commission plans. The DSL should support infix to prefix conversion and JSON conversion.

2.4.2 Architecture Overview

The architecture for the system will consist of the following components:

- Lexer and Parser: ANTLR lexer and parser will be used to parse commission plan expressions and if-else conditions defined by the users.
- Compiler: The compiler will compile the commission plan expressions and conditions into an intermediate representation, which can then be evaluated by the evaluator.
- Infix to Prefix Converter: The infix to prefix converter will convert infix commission plan expressions to prefix notation.
- JSON Converter: The JSON converter will convert commission plans to JSON format.
- IDE: The IDE will provide a user-friendly interface for writing, testing, and debugging commission plans. It will communicate with the Lexer and Parser, Compiler, Evaluator, Infix to Prefix Converter, and JSON Converter as needed.

2.4.3 Data Flow Diagram

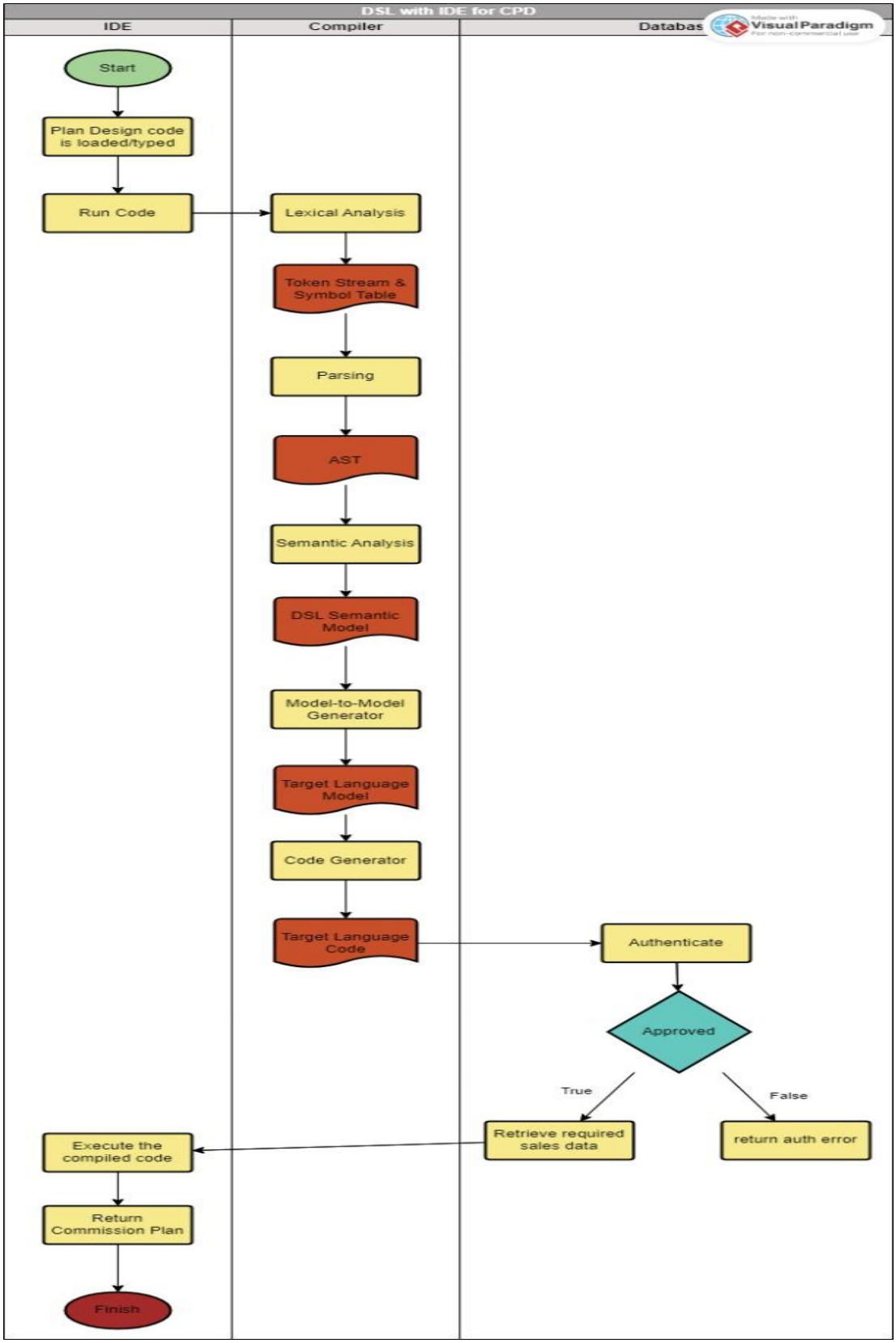

Figure 2.1

The data flows through several stages, each stage processing the input data in a specific way to produce the output data. Here is a high-level overview of the flow of data through a compiler:

- Lexical Analysis:

  The first stage of the compiler is the Lexical Analysis phase, where the input code is broken down into a stream of tokens. A token is a sequence of characters that represents a single entity, such as a keyword, identifier, or operator.

- Syntax Analysis:

  The second stage is the Syntax Analysis phase, where the tokens generated by the Lexical Analysis phase are parsed into a tree-like structure called the Abstract Syntax Tree (AST). The AST represents the structure of the input code in a way that is easier to process than the original source code.

- Semantic Analysis:

  The third stage is the Semantic Analysis phase, where the AST is checked for semantic errors. This includes checking for correct variable declarations and usage, type checking, and other language-specific checks.

- Intermediate Code Generation:

  The fourth stage is the Intermediate Code Generation phase, where the AST is translated into an intermediate representation that is closer to the machine language. The intermediate code may be in the form of bytecode, assembly code, or some other form that is easier to execute.

- Code Generation:

  The final stage is the Code Generation phase, where the optimized intermediate code is translated into machine code that can be executed by the target processor. The machine code may be in the form of object files or executable files, depending on the specific compiler and target platform.

2.4.4 ER Diagram

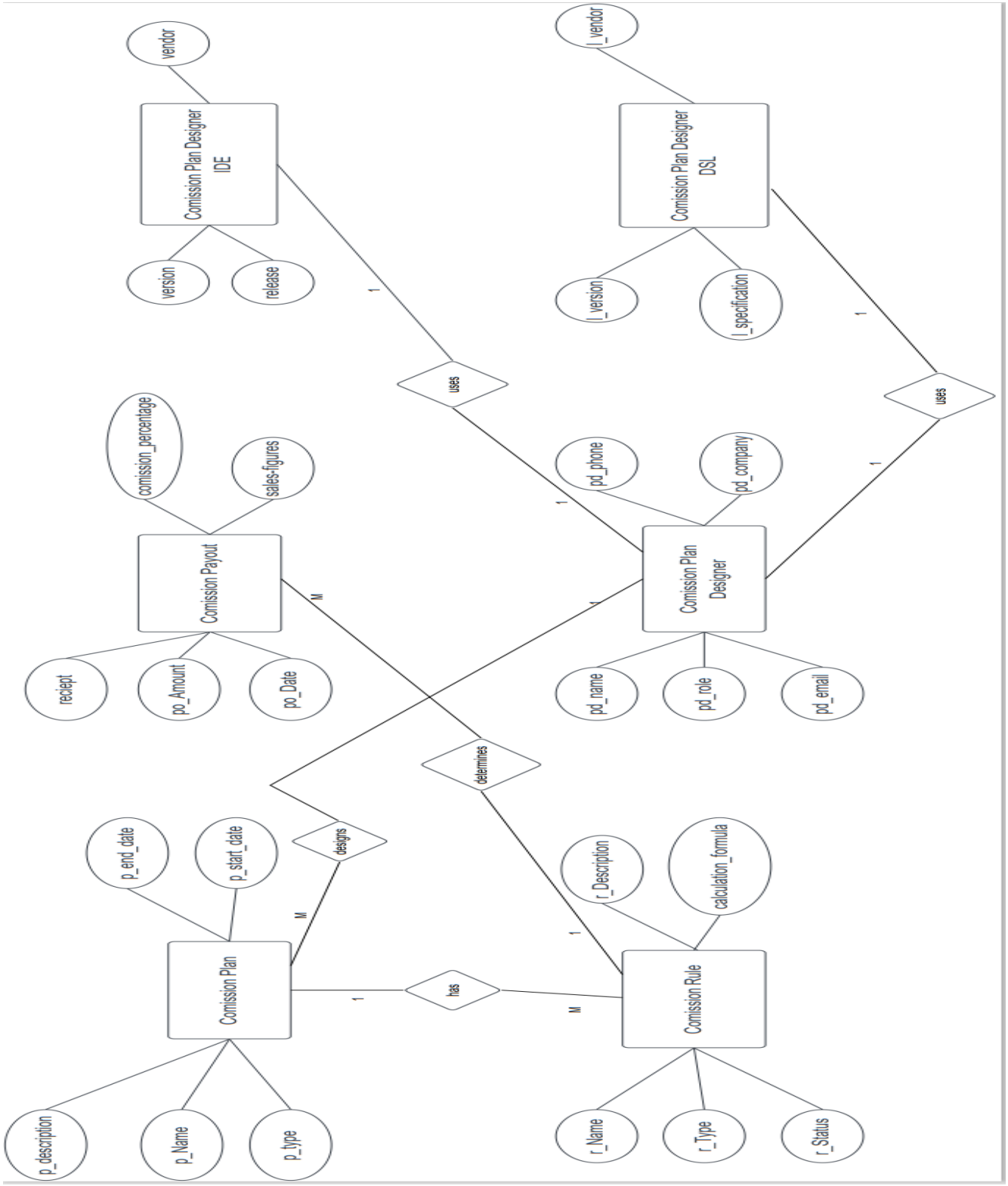

Figure 2.2

2.4.5. Component Diagram

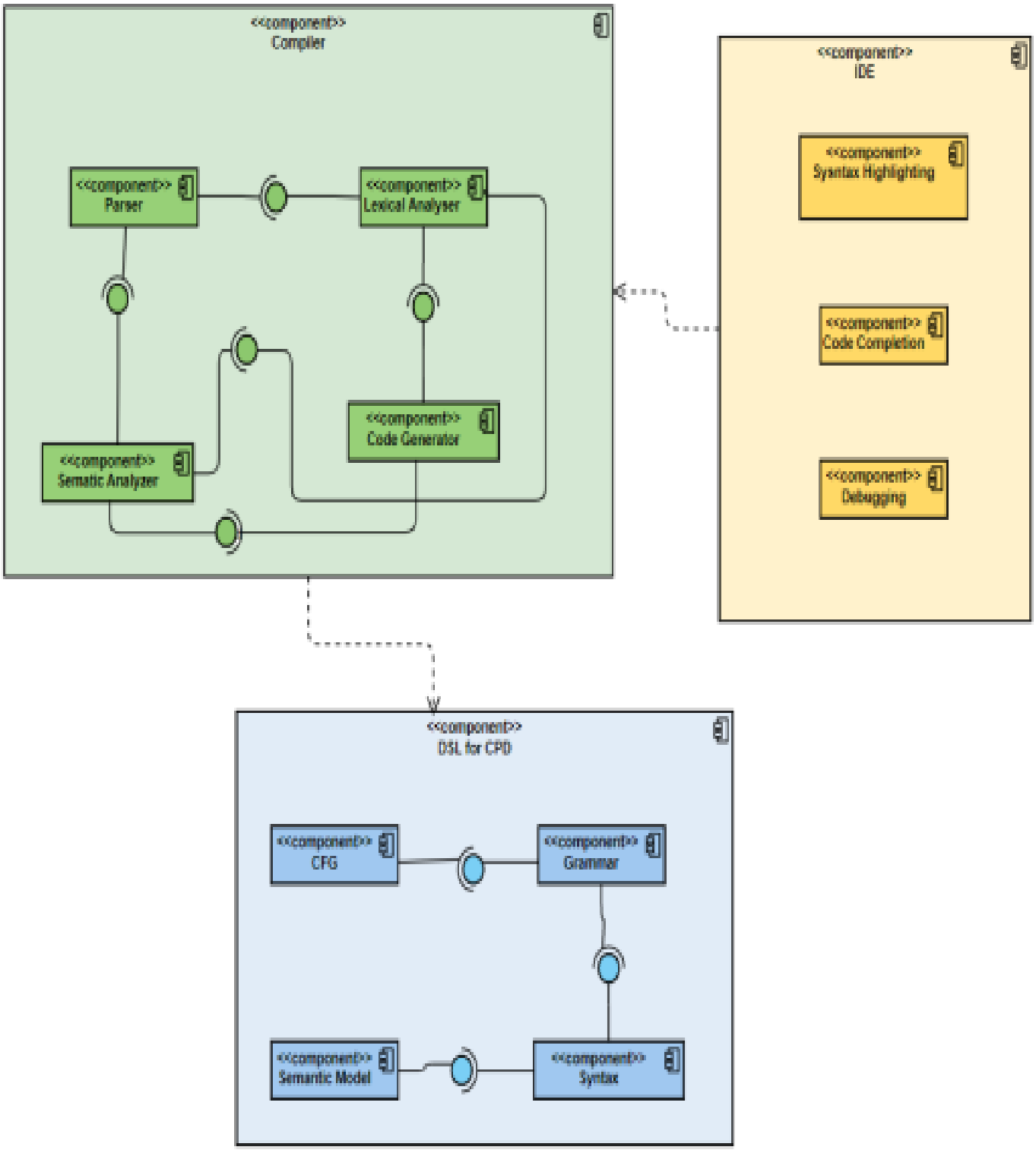

Figure 2.3

- Lexer and Parser Component:

  The Lexer and Parser components are responsible for tokenizing the input code and generating the corresponding abstract syntax tree (AST). The component includes subcomponents such as the Lexer and Parser, which work together to tokenize the input code and generate the AST.

- Syntax Component:

  The Syntax component represents the rules that govern the structure and syntax of the DSL. This includes the grammar and syntax rules that define the valid syntax for the language.

- Semantic Model Component:

  The Semantic Model component represents the semantic rules and constructs of the DSL. This includes the entities and relationships that are defined by the language, such as variables, functions, and data types.

- Semantic Analyzer Component:

  The Semantic Analyzer component is responsible for checking the AST for semantic errors, such as incorrect variable declarations and usage, type checking, and other language-specific checks. This component includes subcomponents such as the Symbol Table, which keeps track of variables and their types, and the Type Checker, which verifies that types are used correctly.

- Code Generator Component:

  The Code Generator component is responsible for translating the AST into executable code. This includes subcomponents such as the Instruction Selector, which selects the appropriate machine instructions for each node in the AST, and the Register Allocator, which assigns registers to variables in the code.

- Syntax Highlighting Component:

  The Syntax Highlighting component is responsible for providing visual cues to the user about the syntax of the code. This includes highlighting keywords, variables, and other language-specific constructs.

- Debugging Component:

  The Debugging component is responsible for providing tools for debugging the

code. This includes subcomponents such as the Debugger, which allows the user to step through the code and inspect variables, and the Profiler, which measures the performance of the code.

2.4.6 Infix to Prefix Converter

- The Infix to Prefix Converter will receive the parse tree generated by the Parser and convert the commission plan expression from infix notation to prefix notation.
- The Converter will use a stack-based algorithm to traverse the expression in infix notation and generate the equivalent expression in prefix notation.

2.4.7 Lexer and Parser

- The Lexer will receive the input commission plan expression and tokenize it into a stream of tokens.
- The Parser will receive the token stream generated by the Lexer and generate a parse tree from it, based on the Commission Plan language grammar.
- The Lexer and Parser will work together to ensure that the commission plan expression is syntactically valid and can be parsed correctly.
- The parse tree generated by the Parser will be sent to the Infix to Prefix Converter component.

2.4.8 JSON Converter

- The JSON Converter will receive the commission plan expression in prefix notation from the Infix to Prefix Converter.
- The Converter will generate the commission plan structure and variables in JSON format.
- The output of the JSON Converter will be sent to the IDE component for displaying the commission plan structure and variables.

2.4.9 IDE

- The IDE will receive the commission plan expression from the user and send it to the Lexer component for tokenization.
- The Lexer will then send the token stream to the Parser component for generating a parse tree.
- The parse tree will be sent to the Infix to Prefix Converter component for generating the equivalent expression in prefix notation.
- The prefix notation expression will be sent to the JSON Converter component for generating the commission plan structure and variables in JSON format.
- The commission plan structure and variables in JSON format will be sent to the IDE for displaying.
- The IDE will allow users to set breakpoints, inspect variables, and step through commission plan expressions during debugging.
- The IDE will integrate with source control systems, such as Git, for versioning and collaboration.
- The IDE will allow users to export and import commission plans in a variety of formats, such as CSV, JSON, and XML.

Conclusion

The system design for the domain-specific language (DSL) built using ANTLR lexer and parser for parser commission plans, and an integrated development environment (IDE) for writing, testing, and debugging commission plans should meet the functional and non-functional requirements defined in the software requirement specification. The architecture of the system should be scalable and maintainable.

3. Implementation

3.1 Lexer and Parser

The Lexer and Parser will be implemented using the ANTLR lexer and parser generator tool. ANTLR will be used to generate lexer and parser code in Java, which will be used to tokenize the input commission plan expression and generate the parse tree respectively.

The commission plan language grammar will be defined using a Context-Free Grammar (CFG) in ANTLR. The CFG will specify the syntax rules for the commission plan language and will be used by ANTLR to generate the parser code.

3.2 Infix to Prefix Converter

The Infix to Prefix Converter will be implemented as a Java class that takes in a parse tree as input and generates the equivalent commission plan expression in prefix notation.

The Converter will use a stack-based algorithm to traverse the parse tree in infix notation and generate the equivalent expression in prefix notation.

3.3 JSON Converter

The JSON Converter will be implemented as a Java class that takes in the commission plan expression in prefix notation as input and generates the commission plan structure and variables in JSON format.

3.4 IDE

The IDE will be implemented as a Visual Studio Code (VSC) plugin using the VSC extension API.

The plugin will provide an editor interface for users to enter commission plan expressions and debug them.

The plugin will use ANTLR to generate the lexer and parser code and the CFG for the commission plan language grammar.

The plugin will use the Infix to Prefix Converter and JSON Converter classes to convert the commission plan expression to prefix notation and generate the commission plan structure and variables in JSON format.

The plugin will use the VSC debugging API to provide debugging functionality, such as setting breakpoints, stepping through expressions, and inspecting variables.

The plugin will also integrate with source control systems, such as Git, for versioning and collaboration.

Finally, the plugin will allow users to export and import commission plans in a variety of formats, such as CSV, JSON, and XML.

Conclusion

Overall, the implementation will require a solid understanding of ANTLR, data structures, algorithms, and programming paradigms. The implementation should be modular, scalable, and testable, with appropriate unit tests and integration tests. Code reviews and continuous integration practices should be used to ensure high-quality and maintainable code.

5. Conclusion and Future Enhancements

In conclusion, building a commission plan domain-specific language (DSL) and integrated development environment (IDE) can be a challenging task that requires a solid understanding of ANTLR, data structures, algorithms, and programming paradigms. However, by following the key considerations mentioned above, the system can be designed to be modular, scalable, and maintainable, with appropriate unit and integration tests, continuous integration practices, and code reviews to ensure high-quality and maintainable code. This will ultimately result in a user-friendly IDE that allows developers to write, test, and debug commission plans efficiently, while also being able to handle large volumes of commission plans and users. Overall, a

well-designed and well-implemented system will provide an excellent tool for managing and calculating commissions in a variety of industries.

Future Enhancements

- A robust grammar for a DSL will ensure that language is clear,unambiguous and easy to understand for the users.
- An interactive IDE (Integrated Development Environment) for a DSL which provides a powerful tool for users to create and manage their programs.
- The DSL could include built-in visualization tools to help businesses better understand how their commission plans are impacting their sales team's performance.
- Future enhancements to a commission plan design DSL could help businesses further optimize their commission plans and better manage their sales teams, ultimately driving increased sales performance and revenue growth.