Design and Implementation of a Domain-Specific Language and Compiler for Financial Calculations

Andrés Felipe Vanegas Bogotá, Jason Stevens Solarte Herrera
Faculty of Engineering – Systems Engineering Program
Universidad Distrital Francisco Jose de Caldas

CONTENTS

LIST OF FIGURES

Design and Implementation of a Domain-Specific Language and Compiler for Financial Calculations

***Abstract*—This paper presents the design and implementation of a domain-specific language (DSL) and its corresponding compiler tailored for financial calculations. The DSL simplifies the process of defining and computing financial operations such as interest calculations, maturity dates, and payment schedules. The compiler, implemented in Python, translates high-level financial instructions into intermediate code, which is then executed to produce accurate results. The system includes a lexical analyzer, a syntactic parser, a semantic analyzer, and an intermediate code generator. The paper discusses the architecture of the compiler, the design of the DSL, and the results of testing the system with various financial scenarios. The implementation demonstrates the effectiveness of using DSLs and compilers to automate complex financial computations, providing a robust tool for financial analysts and developers.**

I. Introduction

The complexity of financial calculations often requires specialized tools to ensure accuracy and efficiency. Traditional programming languages, while powerful, can be cumbersome for domain-specific tasks. This paper introduces a domain-specific language (DSL) designed specifically for financial calculations, along with a compiler that translates high-level financial instructions into executable code. The DSL simplifies the definition of financial operations, making it accessible to users without extensive programming experience. The compiler, implemented in Python, ensures that the instructions are correctly interpreted and executed.

II. Problem Definition

Financial calculations, such as interest computation, maturity date determination, and payment scheduling, are critical in various financial applications. However, these calculations can be error-prone when performed manually or using general-purpose programming languages. The lack of a dedicated tool for financial computations leads to inefficiencies and potential inaccuracies. This paper addresses the need for a specialized language and compiler that can automate and simplify these calculations.

III. Proposed Solutions

A. Domain-Specific Language Design

The DSL is designed with a syntax that closely resembles natural financial terminology. It includes keywords such as `CALCULATE`, `INTEREST`, `AMOUNT`, `RATE`, and `TIME`, making it intuitive for users to define financial operations.

B. Compiler Architecture

The compiler is composed of several key components that work together to process and execute financial instructions. These components include the Lexical Analyzer (Lexer), Syntactic Parser (Parser), Semantic Analyzer, and Intermediate Code Generator. Below, we describe each component in detail.

1) Lexical Analyzer (Lexer): The Lexical Analyzer is responsible for breaking down the input text into meaningful tokens. It uses regular expressions to identify keywords, numbers, dates, and other patterns while ignoring whitespace and comments. The Lexer ensures that the input text is correctly tokenized, providing the foundation for further processing.

```
- INTEREST: \binterest\b
- RATE: \brate\b
- AMOUNT: \bamount\b
- DATE: \bdate\b
- NUMBER: \b\d+(\.\d+)?\b
```

Listing 1. Example Token Patterns

The Lexer processes the input text and generates a list of tokens, which are then passed to the Syntactic Parser for further analysis.

2) Syntactic Parser (Parser): The Syntactic Parser validates the structure of the tokens according to a predefined grammar. It ensures that the tokens follow the correct syntax for financial calculations, such as interest, maturity, payments, and balances. The Parser uses a set of rules to validate the sequence of tokens and raises custom errors if the syntax is invalid.

```
<PROGRAM>       -> <INSTRUCTION>*
<INSTRUCTION>   -> <CALCULATE> | <DEFINE_RATE>
    | <PAYMENT> | <BALANCE>
<CALCULATE>     -> "CALCULATE" (<INTEREST> | <
    MATURITY>)
<INTEREST>      -> "INTEREST" "AMOUNT" <NUMBER>
     "RATE" <NUMBER> "TIME" <NUMBER> "YEARS"
<MATURITY>      -> "MATURITY" "DATE" "
    DATE_VALUE" "PERIOD" <NUMBER> "DAYS"
```

Listing 2. Grammar Rules

The Parser ensures that the instructions are correctly structured and delegates the processing of each instruction to the appropriate handler.

3) Semantic Analyzer: The Semantic Analyzer validates the meaning of the tokens generated by the Lexer and Parser. It ensures that the tokens follow logical rules, such as positive amounts, valid rates, and correct time values. If any semantic rule is violated, it raises a custom `SemanticError` with detailed feedback.

```
- Amount must be positive.
- Rate must be between 0 and 100.
- Time must be positive.
```

Listing 3. Semantic Rules

The Semantic Analyzer ensures that the instructions are not only syntactically correct but also logically valid.

4) Intermediate Code Generator: The Intermediate Code Generator translates the validated instructions into an intermediate representation. This intermediate code is then passed to the Assembler, which converts it into executable instructions. The Intermediate Code Generator ensures that the high-level financial instructions are correctly translated into a format that can be executed by the system.

```
- INTEREST_CALCULATION
- AMOUNT 1000
- RATE 5
- TIME 2
- YEARS
```

Listing 4. Example Intermediate Code

The Intermediate Code Generator plays a crucial role in bridging the gap between high-level instructions and executable code.

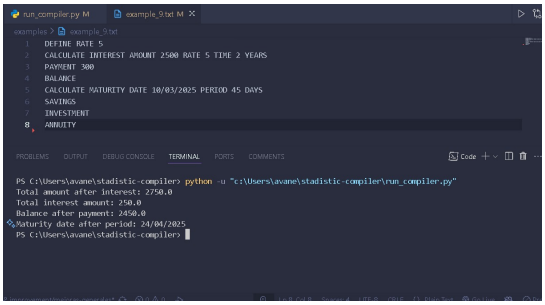

Fig. 1. Compiler Architecture Overview

IV. Criteria for Assessing Solutions

The effectiveness of the proposed solutions is assessed based on the following criteria:

- **Accuracy**: The system must produce correct financial calculations.
- **Efficiency**: The compiler should process instructions quickly.
- **Usability**: The DSL should be easy to learn and use.
- **Extensibility**: The system should allow for the addition of new financial operations.

V. Research Methodology

The research methodology involved the following steps:

- **Literature Review**: Study of existing DSLs and compilers for financial applications.
- **System Design**: Design of the DSL syntax and compiler architecture.
- **Implementation**: Development of the compiler components in Python.
- **Testing**: Validation of the system with various financial scenarios.

VI. Analysis and Interpretation

The system was tested with several financial scenarios, including interest calculations and maturity date determinations. The results demonstrated that the DSL and compiler accurately and efficiently processed the instructions. The usability of the DSL was confirmed through user testing, with participants finding the syntax intuitive and easy to use. The system's extensibility was demonstrated by adding new financial operations without significant modifications to the compiler.

VII. Conclusions and Recommendations

The design and implementation of a domain-specific language and compiler for financial calculations have proven to be effective in automating complex financial computations. The system provides a robust tool for financial analysts and developers, reducing the potential for errors and improving efficiency. Future work could focus on expanding the DSL to include more advanced financial operations and integrating the compiler with existing financial software.

VIII. Compiler Source Code

The source code for the compiler is available in the project repository.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] D. Horowitz, *End of Time*. New York, NY, USA: Encounter Books, 2005. [E-book] Available: ebrary, http://site.ebrary.com/lib/sait/Doc?id=10080005. Accessed on: Oct. 8, 2008.