DOMAIN SPECIFIC LANGUAGE FOR ACCOUNTING

Egor BABCINEȚCHI, Daniel POGOREVICI, Iulia ȚĂRUȘ, Rafaela CERLAT*

Department of Software Engineering and Automatics, FAF-202, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova

*Corresponding author: Rafaela Cerlat, rafaela.cerlat@isa.utm.md

Abstract: *This paper analyzes and describes the implementation of a Domain Specific Language (DSL) in the accounting sector. These days entrepreneurs and business owners spend thousands of euros a year on tax consultants. This DSL would allow users to establish new taxes and to do calculations using real data very quickly with no additional cost.*
Keywords: *domain specific language, grammar, syntax, parser, lexer, taxes, accounting, finance*

Introduction

A Domain Specific Language is a programming language with a higher level of abstraction optimized for a specific class of problems. IT uses concepts and rules from the field or domain [1].

Domain-Specific Languages are used for different reasons and by different types of users. Some DSLs are intended for programmers, and for that reason are more technical, while others are targeted towards people with no experience in programming and therefore, they have less technical concepts and syntax. DSLs are viewed as User Interfaces (UIs) because they bridge the gap between the domain experts and the computation platforms.

Accounting is the process of recording financial transactions related to a business. This process includes summarizing, analysing and reporting these transactions to inspection agencies, regulators, and tax collection entities. The financial statements used in accounting are a concise summary of financial transactions over a period, encapsulating a company's operations, financial position, and cash flows.

Accounting may be handled by a bookkeeper or an accountant at a small firm, or by large finance departments with dozens of employees at bigger companies. Because of the increasing market and customer demands, the accounting and finance sector has to explore new ways and strategies to cut back costs and improve efficiency. Thus, accounting automation would take the most manual components of an accountant's work day and perform them instantly [2].

This DSL would represent a financial model, an abstract representation of the real-world financial situation, designed to illustrate a simplified version of a financial asset or portfolio of a business, project, or any other investment.

Introduction in Financial Accounting

Financial statements are written records that contain the financial performance and business activities of a company. Financial statements are often inspected by government agencies, accountants and firms to ensure accuracy and for tax, financing, or investing purposes. These statements include: balance sheet, income or cash flow statement.

A balance sheet shows what a company owns (its "assets") and owes (its "liabilities") as of a particular date, along with its shareholders' equity.

Assets are divided into current assets, which can be converted to cash in one year or less, and non-current or long-term assets, which cannot. A liability is any money that a company owes to outside parties, from bills it has to pay to suppliers to interest on bonds issued to creditors to rent, utilities and salaries [3].

Grammar

This DSL has a grammar defined by the 4-tuple $G = \{V_N, V_T, S, P\}$, where:

V_N - is a finite set of non-terminal variables;

V_T - is a finite set of terminal variables;
S - is the start variable (or start symbol);
P - is a finite set of production rules of the grammar.

V_N = {<program>, <declaration>, <classDecl>, <funcDecl>, <varDecl>, <statement>, <finDecl>, <balanceSheet>, <incomeState>, <bal_equity>, <bal_assets>, <bal_liab>, <assets_ex>, < liabilities_ex>, < equity_ex>, <bal_sheet_assets_ex> , <special_identifer>, etc.}
V_T = {class, fun, of, for, Balance Sheet, Income Statement, SRL, Individual, assets, liabilities, equity, if, print, return, while, and, not equal, true, false, cash, expenses, inventory, accounts, long_term, accounts-liab, others, long-term-liab, capital, retained, exportBalanceSheet, importBalanceSheet, =, >, >=, <, <=, -, +, !, *, 0..9, etc.}

P = { <program> → <declaration>
<declaration> → <classDecl>
<declaration> → <funcDecl>
<declaration> → <varDecl>
<declaration> → <statement>
<declaration> → <finDecl>
<declaration> → <balanceSheet>
<declaration> → <incomeState>
<classDecl> → **class** <identifier> **{ }**
<classDecl> → **class** <identifier> **extends** <identifier> **{ }**
<classDecl> → **class** <identifier> **extends** <identifier> **{<**function> **+}**
<funDecl> → **fun** <function>
<finDecl> → <identifier> **of** <entities> **{<**function>***}**
<balanceSheet> → **Balance Sheet** <identifier> **for** <identifier> **{** <function>* **}**
<balanceSheet> → **Balance Sheet** <identifier> **for** <identifier> **{**<balProps>* <function>*}
<incomeState> → **Income Statement** <identifier> **for** <identifier> **{ <**function>***}**
<balProps> → <bal_equity> | <bal_assets> | <bal_liab>
<bal_assets> → **assets {**<assets_ex> **+}**
<bal_assets> → **assets** <identifier> **{**<assets_ex> **+}**
<bal_liab> → **liabilities {**< liabilities_ex> **+}**
<bal_liab> → **liabilities** <identifier> **{**< liabilities_ex> **+}**
<bal_equity> → **equity {**< equity_ex> **+}**
<bal_equity> → **equity** <identifier> **{**< equity_ex> **+}**
<asset_ex> → <bal_sheet_assets_ex> **=** <expression>
< liabilities_ex> → <bal_sheet_liabilities_ex> = <expression>
< equity_ex> → <bal_sheet_equity_ex> = <expression>
<statement>→ <exprStmt> | <forStmt> | <ifStmt> | <printStmt> | <returnStmt> | <whileStmt> | <block>|<expression>
<exprStmt> → <expression>
<forStmt> → **for (** (<varDec> | <exprStmt> | **;**) expression? **;** expression?**)** < statement>
<ifStmt> → **if (**<expression>) < statement>
<ifStmt> → **if (**<expression>) < statement> **else** <statement>
<printStmt> → **print** <expression>

<returnStmt> → **return** <expression>
<whileStmt > → **while (** <expression> **)** <statement>
<block> → **{** <declaration>* **}**
<expression> → <assignment>
<assignment> → <identifier> = <assignment> | <logic_or >
<assignment> → <call> **.** <identifier> = <assignment> | <logic_or>
<logic_or> → <logic_and> (**and** <logic_and>)*
<logic_and> → <equality> (**and** <equality>)*
<equality> → <comparison> ((**not equal** | ==) <comparison>)*
<comparison> → <term> ((> | >= | < | <=) <term>)*
<term> → <factor> ((**-** | +) <factor>)*
<factor> → <unary> ((**/** | *****) <unary>)*
<unary> → (**!** | **-**) <unary> | <call>
<call> → <primary> (**( )** | **.** <identifier>)*
<call> → <primary> (**(** <arguments> **)** | **.** <identifier>)*
<primary> → **true** | **false** | **null** | **this** | <number> | <string> | <identifier> | <special_identifier> | **(**<expression>**)**
<function> → <identifier> **( )** <block>
<function> → <identifier> **(**<identifier>**)** <block>
<parameters> → <identifier> (**,** <identifier>) *
<arguments> → <expression> (**,** <expression>)*
<identifier> → <letter> (<letter> | <digit>) *
<digit> → **0** | ... | **9**
<letter> → [a-z … A-Z]
<entities> → **SRL** | **Individual**
<bal_sheet_assets_ex> → <bal_sheet_assets>
<bal_sheet_liab_ex> → <bal_sheet_liab>
<bal_sheet_equity_ex> → <bal_sheet_equity>
<bal_sheet_assets> → **cash** | **expenses** | **inventory** | **accounts** | **long_term**
<bal_sheet_liab> → **accounts-liab** | **others** | **long-term-liab**
<bal_sheet_equity> → **capital** | **retained**
<special_identifer> → **exportBalanceSheet** | **importBalanceSheet**

Lexer and Parser

Building a DSL consists of multiple stages. Lexical Analysis is the first step in the compiler designing. Therefore, this DSL will need a lexer that takes the source code (a sequence of characters) and converts it into a set of **tokens**. The lexer will contain a **tokenizer**, also called a **scanner**. In the process of scanning, if it detects that a token is invalid, it generates an error message. So, the role of the lexer would be to read the characters from the source code, check for valid tokens, and pass this data to the next step-the syntax analyzer.

Syntax Analysis is the second step of this process, when the given input of tokens is checked against the production rules and structure of the formal grammar. The parser would analyse the structure of the input, check if it is the correct syntax for the DSL, and detect any errors.

Semantic analysis would be the task of ensuring that the statements and declarations of a program are semantically correct, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

The input for the code generator usually consists of an abstract syntax tree or a parse tree. This tree is converted into a linear sequence of instructions, usually in an intermediate language such as three-address code [4-6].

For this DSL the user would write programs by typing in commands. So, the DSL will require the user to specify the type of fees, taxes or any other services he wants to be evaluated.

After building a parser and an interpreter for this domain specific language, the interpreter would just calculate the values of any field for any entity and the amount of taxes for each entity affected and print them on the screen. So, the output would be the calculated numbers and statistics.

The DSL will evaluate the code from the first line to last line, left to right, if there are no other constraints such as loops or conditionals.

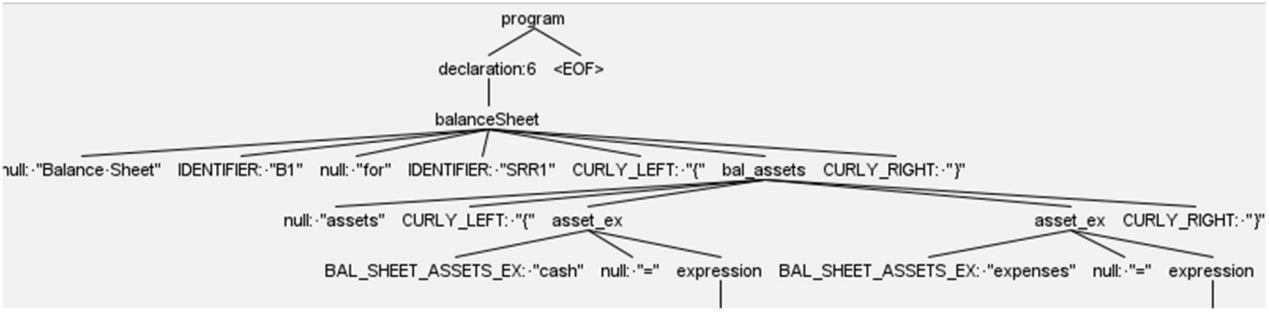

Figure 1. Parsing tree

Conclusions

In this paper the use of DSL in the financial field was analyzed. Such a tool could be used by small companies, agencies, start-ups and business managers with no specific knowledge of accounting, because it would use domain-relevant abstractions and notations.

Automating a range of accounting and financial operations will result in smoother operations, saved time and an overall better work-flow. In other words, the presented DSL would bring greater opportunities for the accounting sector, because it would reduce the complicated, time and money consuming work, as well as allow those interested in accounting and finance to focus more on value and results.

References:

1. *Domain-Specific Languages*. [online] [visited 28.02.2022]. Available: https://www.jetbrains.com/mps/concepts/domain-specific-languages/
2. Jason FERNANDO, *Guide to Accounting.* [online] [visited 28.02.2022]. Available: https://www.investopedia.com/terms/a/accounting.asp.
3. Chris B. MURPHY, *Corporate Finance & Accounting. Financial Statements.* [online] [visited 28.02.2022]. Available: https://www.investopedia.com/terms/f/financial-statements.asp.
4. Robert NYSTROM, *Crafting Interpreters*, 2021. [online] [visited 28.02.2022]. Available: https://craftinginterpreters.com.
5. Mirian HALFELD-FERRARI, *Compilers. Lexical Analysis*. [online] [visited 28.02.2022]. Available: https://www.univ-orleans.fr/lifo/Members/Mirian.Halfeld
6. John SMITH, *Lexical Analysis (Analyzer) in Compiler Design*. [online] [visited 28.02.2022]. Available: https://www.guru99.com/compiler-design-lexical-analysis.html.