8-2-2024

A Domain-Specific Language for Accounting

Janhavi Doshi
jd6583@rit.edu

RIT

A Domain-Specific Language for Accounting

by

Janhavi Doshi

A Thesis Submitted in Partial Fulfillment of the Requirements for the

Degree of Master of Science
in Computer Science

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology

Rochester, NY

August 2, 2024

Committee Approval

Dr. Matthew Fluet Date

Associate Professor

Chief Thesis Advisor

Dr. Hans-Peter Bischof Date

Graduate Program Coordinator

Committee Member

Dr. Aaron Deever Date

Undergraduate Program Coordinator

Committee Member

Abstract

Accounting can be loosely described as a set of rules and best practices for recording and reporting financial transactions. Despite the seemingly straightforward nature of these principles, implementing them through programming has proven to be complex. Currently, over 550 accounting software solutions exist, developed using more than 17 programming languages. These diverse and often fragmented solutions have led to numerous instances of fraud and errors that elude early detection due to inadequate validity checks. This study investigates the need for a unified approach by developing a Domain-Specific Language (DSL) for Accounting. The proposed DSL aims to standardize accounting practices, enhance the accuracy of financial reporting, and facilitate the detection of inconsistencies and fraudulent activities. By incorporating accounting-specific syntax and semantics, the DSL is designed to bridge the gap between accountants and programmers, making programming more accessible to accounting professionals. This study will explore the design principles, integration strategies, and potential benefits of implementing such a language.

Contents

Chapter 1

Introduction

At its core, accounting can be seen as a structured framework of rules and best practices for accurately recording and reporting financial transactions. While at first glance the foundational concepts might appear straightforward, translating these principles into programming systems is far from simple. The current landscape of accounting solutions is populated with over 550 distinct accounting software offerings, each crafted using a variety of more than 17 different programming languages. This vast diversity has resulted in fragmented systems that are implementing the same logic without any generalized standards. The consequences of this fragmentation are significant and pervasive. Errors and instances of fraud often slip through the cracks undetected for a long time due to the lack of standardized validity. Such vulnerabilities can lead to substantial financial misstatements and undermine the trust in financial reporting systems. The pressing need for a cohesive approach to accounting software is evident. This study proposes the development of a Domain-Specific Language (DSL) for accounting, aptly named the Accounting Storage Language (ASL). ASL aims to create a unified framework that standardizes accounting practices across different platforms, thereby enhancing the accuracy and reliability of financial reports. By integrating accounting-specific syntax and semantics, ASL seeks to bridge the gap between the domain knowledge of accountants and the technical expertise of programmers. ASL is a designed to be purely accounting-focused data format language. The introduction of ASL promises several key benefits. Firstly, it would provide a consistent and transparent method for recording financial transactions and "flagging" suspicious or unusual transactions for timely review, reducing the likelihood of errors and fraud. Secondly, it would make the programming of accounting systems more accessible to professionals in the field of accounting, who may not have extensive programming knowledge. Lastly, a standardized data format language would facilitate better integration

and uniformity between different accounting systems, fostering a more robust financial reporting environment. This paper will delve into the intricacies of designing ASL, examining the principles that underpin its development, the strategies for its integration into existing systems, and the broader implications for the accounting industry. By addressing these areas, the study aims to demonstrate how ASL can serve as a pivotal tool in modernizing and securing the field of accounting.

1.1 Current Landscape of Accounting Software

While many businesses prefer customized software, these are still limited in their capabilities depending on the scope of the software applications. An average accountant's toolkit often includes a customized Enterprise Resource Planning (ERP) solution alongside Excel, the most widely used tool in the accounting domain. While this gets the job done, empowering accountants with programming knowledge and capabilities will make the work not only more efficient but also more secure and less error-prone. While a customised ERP solution is fit for the task, accountants rarely have an clear idea of what goes on behind the application layer. The application logic is solely managed by programmers and there is little to no concept of pair-programming present between accountants and programmers. For changing one any logic or any functionality, accountants have to rely solely on the programming team. Tasks like this would be much more efficient with the use of programming skills, especially with a domain-specific language equipped to handle accounting specific data. However, general programming languages and the multitude of them are proving to be a barrier to entry for the accounting community. This combined with technical jargon that often comes with programming, has resulted in there being little to no pair programming between software engineers and accountants. A domain-specific language designed such that key accounting terms are a part of its syntax, would help bridge this gap and make communication between domain experts and programmers easier.

1.2 Concept of Domain-Specific Languages

General purpose languages like Java and Python are designed for a wide range of applications. In contrast, Domain-Specific Languages (DSLs) are programming languages that are designed specifically for a particular application domain, area of interest or industry. It is commonly believed that DSLs significantly improve ease of use and increase efficiency. While there is much diversity in various DSLs, a notable feature for

this discussion would be the syntax and semantics tailored to the domain for which the DSL was created. Accounting specific syntax would be instrumental in encouraging more accounting professionals to learn programming. However, DSL development is very challenging since having both domain knowledge and language design expertise are key to a successful attempt. There are very few people, if any, who possess this knowledge overlap. In this thesis, we will be developing a domain-specific language designed for accounting.

1.3 ASL Design Principles

In general-purpose languages, the most common goal of a program is to arrive at a computation and a final result or value through a series of computations. The intermediate steps leading to this final result are often not of utmost concern. However, the primary purpose of accounting is to keep a detailed and thorough record of everyday financial transactions. The closing balances of accounts are important, but they are secondary to the detailed steps that led to these balances, since these steps are instrumental in providing a clear audit trail.

Another distinction between general-purpose languages and ASL lies in the depth of integration of validation and verification logic. General-purpose libraries typically handle validation at the application level, treating the database as a passive data store, with checks performed before and after database operations. In contrast, ASL embeds validation directly into the database language, ensuring data integrity during the storage process. This deep integration allows ASL to enforce domain-specific rules actively and maintain data validity natively within the database layer.

1.4 Flags

In recent years, numerous financial frauds have been concealed through various manipulations of accounting records. To counteract this, accounting has many rules and regulations in place. However, it depends solely on its end-user (accountants) to uphold these rules. This leads to human errors and does not facilitate early detection of frauds or mistakes. In ASL, we have incorporated a ‘flag system’ designed after concurrent auditing, in which an auditor checks the books of accounts on a rolling basis. Similar to this, we have ‘flags’ in ASL, which are user-defined and act as warning for unusual or suspicious transactions. All transactions

that are 'flagged' either have to provide an explanation or evidence to be a valid transaction in ASL. This explanation is then logged as a part of the audit trail which helps facilitate transaparency and early detection of frauds and errors.

1.5 Corrections

Another interesting aspect of accounting that is also derived from its dependence on its end-users to uphold the rules is allowing for mistakes and human errors. Accounting has provisions in place to rectify mistakes and errors via a multitude of entries such as reverse entries or contra entries, which lets accountants reverse any previous wrong entries. To emulate this behaviour, we have also allowed for "corrections" in ASL. Our goal has been to allow for corrections while maintaining the validity and integrity of the financial data. We have acheived this by allowing only back-dated corrections which do no cause any other past or future transactions to become invalid as a result.

1.6 Limitations

During this study, we tried to envision accounts as 'objects' and attempted to incorporate OOP (Object Oriented Programming) concepts into ASL. However, this proved to be more challenging than we initially realised. Accounting has many tasks which are closely related to it but outside its direct scope. We tried embedding functionalities like Inventory Valuation, Management Reporting, enforcing Accrual Accounting and others in ASL. However, using OOP principles, modelling these functionalities required collecting, storing and processing a variety different attributes. This would have significantly increased the complexity of ASL. To model these behaviours, ASL would have had to be equipped with the ability to understand and process the context of each transaction, which can vary very significantly.

Hence, while ASL can model all core accounting tasks effectively, it is currently not equipped to provide support for accounting-adjacent tasks, and these will be a part of future work towards making ASL more versatile.

We will now take a closer look at Domain-Specific Languages, ASL's Design and its Syntax and Semantics.

Chapter 2

Understanding Domain-Specific Languages

2.1 Introduction to Domain-Specific Languages

A Domain-Specific Language (DSL) is usually a small programming language that focuses on providing optimal and direct solutions to a small set of problems within a specific domain or industry. Unlike General-Purpose Programming Languages (GPLs), Domain-Specific Languages may be well-suited for only a small number of tasks. However, the intent behind DSLs is to provide more efficient, intuitive, and straightforward solutions through specialization.

2.1.1 Comparison with General-Purpose Programming Languages

General-Purpose Programming Languages (GPLs), like Python and Java, are widely popular for being capable of providing solutions to most, if not all, programming problems or tasks. However, the consequent lack of specialization often renders generic solutions less than optimal. A Domain-Specific Language, on the other hand, focuses on a smaller host of problems and often, if done well, can result in a much better solution in terms of effectiveness and efficiency. While General-Purpose Languages (GPL) and Domain-Specific Languages (DSL) share many common features since they are essentially programming languages, they also differ significantly due to the motivation behind their creation. Some of the differences are:

- **Functionality**: General-Purpose languages offer much more functionality and flexibility. They can be utilized for many different purposes and are suitable for a wide range of tasks, whereas Domain-Specific Languages are highly specialised and their optimal use is limited to the tasks that they were

designed for.

- **Syntax and Usability**: Domain-Specific languages often have domain specific simple and intuitive syntax that makes them more accessible to domain experts.
- **Learning Curve**: The learning curve for DSLs is typically lower for domain experts since the language's constructs and terminology are usually aligned with the domain.

2.1.2 Advantages and Limitations of DSLs

Advantages

- **Increased Productivity**: DSLs often boost productivity since they tend to offer quicker solutions specific to the domain.
- **Better Communication**: The use of domain terminology boosts understanding between experts.
- **Ease of Use for Domain Experts**: DSLs are often more intuitive for domain experts, as they align with their specialized knowledge.
- **Verification**: Domain specialization can be leveraged to enforce or facilitate verification of properties in the domain.

Limitations

- **Scope and Flexibility**: DSLs are not suitable for broader applications beyond the tasks they were designed for.
- **Resource Availability**: DSLs often have fewer resources and community support compared to General-Purpose Languages.
- **Expertise in Both Domain and Language Design**: It is rare to find someone who has expertise in both Domain Knowledge and Language Design. But this is crucial to DSL development.

2.2 Role of DSLs in Different Industries

DSLs have been gaining popularity across many industries to tackle specific tasks or problems. Healthcare, Finance, Law, Machine Learning are just a few examples of industries where DSLs have made significant contributions to enhance efficiency and precision. DSLs have enabled professionals from diverse backgrounds to be equipped with programming skills pertaining to their domain.

2.2.1 SQL

The main strength of SQL is in how it handles complex operations on the state of the database with a relatively simple syntax. Not only does it provide a standard way to perform CRUD (Create, Read, Update, and DELETE) operations, but it also does the same for more advanced operations like aggregations, joins, and nested-queries. The modern ANSI SQL standard also includes support for operations on JSON (Javascript Object Notation) documents, the data format most commonly used in web applications and for describing configurations. Atzeni et al. highlight that SQL's success is largely due to its "simplicity, expressiveness, and closure properties," which have made it an enduring standard in database management [2]. These characteristics have enabled SQL to maintain its relevance even as data management needs have evolved over the decades. [3]

The last 2 decades saw the emergence of NoSQL databases - systems that eschew the SQL model to be more document-centric, to meet the demand of web services that mostly deal with data documents in formats like JSON. Despite this, SQL has continued to evolve and remain relevant. Recent developments have focused on extending SQL's capabilities to be more in line with requirements in modern enterprises. For example, Melton et al. mention the SQL:2011 standard's introduction of time-based features, mainly the PERIOD type and temporal predicates. This extension to the SQL standard enables efficient querying of time-varying data. We also see in the extension to the standard, new clauses such as SYSTEM VERSIONING for automatic tracking of row versions. These features enable complex time-based queries like "AS OF" to retrieve data as it was present at a specific point in time. Although these features vary in the extent to which they are implemented by database vendors, these extensions demonstrate that SQL can be flexible enough to evolve with the times [6].

Ong et al. explore SQL++'s advancements in supporting complex analytics within the database itself. They introduce features like nested data models and flexible schemas, allowing SQL++ to efficiently han-

dle semi-structured data alongside traditional relational data. The language incorporates path navigation expressions for traversing nested structures, and introduces a powerful FLATTEN operator for de-nesting complex data. Additionally, SQL++ extends aggregation capabilities with features like array aggregation and user-defined aggregates, enabling sophisticated in-database analytics that were previously only possible through external processing. These enhancements demonstrate SQL's evolution to meet the demands of modern, data-intensive applications that often involve complex, hierarchical data structures [8].

2.2.2 AWK

AWK, a text-processing language developed in the 1970s, is another example of a DSL that has found widespread use across industries - mainly via its inclusion in several operating systems as a command-line tool. It's name derives from its creators Aho, Weinberger, and Kernighan. AWK was designed specifically for pattern scanning and processing [1]. Its syntax is designed to simplify pattern matching and text manipulation. This makes it very effective for tasks involving parsing and manipulating large swathes of text. It finds application in log file analysis, and other common tasks in command-line workflows.

As Aho et al. explain in their seminal paper, a typical AWK program consists of pattern expressions and associated actions, the input file is processed line by line, if any line matches a pattern in the program, the associated action is executed [1]. This simple yet powerful system gives users the ability to express complex text processing tasks simply. In fact, Van Wyk demonstrates how AWK could be used to implement a variety of algorithms, including some typically associated with more complex programming languages, showing both its power and versatility within its domain [10].

Despite the fact that it was developed in the 70s, AWK continues to be a mainstay in modern computing environments. Schmitz et al. have shown how AWK can be effectively used in bioinformatics for processing large genomic datasets [9]. AWK's influence extends beyond its direct use. Kolovos et al. discuss how AWK's pattern-action paradigm has influenced the design of modern model-to-text transformation languages in model-driven engineering [5]. AWK shows how well-designed DSLs can not only solve domain-specific problems efficiently but also influence broader areas of computer science and engineering.

2.2.3 Catala

Catala is a Domain-Specific Language (DSL) designed specifically for computational law, developed by Merigoux, Chataing, and Protzenko (2021). It addresses the complexity in translating legal texts into executable code, particularly in cases such as tax preparation systems, contract analysis software, and intellectual property analysis tools [7]. The design of Catala is modeled after the unique structural and semantic properties of legal language, pursuing a ”general case first, special cases later” approach that mirrors the structure of legal texts. This is implemented via an override mechanism based on a prioritized default path, enabling post-facto modification of rules [7].

Technically, Catala implements a constrained form of prioritized default logic, where each definition is equipped with a static pre-order encoded directly in the syntax tree. This allows for efficient compilation while still capturing the nuanced logic of legal texts. The language features a sophisticated type system that includes types for dates, durations, and monetary values, and more complex structures like collections. Catala also implements a scope system that enables modular representation of legal texts, closely following their structure and allowing for complex interactions between different sections of law [7].

The Catala compiler performs several sophisticated transformations. It first de-sugars the syntax into a language representing various active scopes. This then gets compiled into a default calculus, and finally a lambda calculus with exceptions is generated. The final output is verified using the F* proof assistant. This ensures that the guarantees the language provides of its behavior are strong and valid [7]. Catala’s syntax is intentionally verbose and descriptive, developed in collaboration with legal professionals to enhance readability for non-programmers. It enforces explicit definition of concepts, even those implicitly understood in legal contexts, promoting clarity and reducing ambiguity [4].

In practice, Catala has been used to formalize complex legal texts such as the French family benefits computation. This formalization led to the discovery and correction of a bug in the official state-sponsored simulator, demonstrating Catala’s potential in improving the accuracy and reliability of legal computations [4]. The language’s design, which closely mirrors the structure of legal texts while providing the rigor of a formally verified programming language, positions Catala as a promising tool for bridging the gap between legal expertise and software engineering in the domain of computational law.

2.3 Relevance and Application of DSLs in Accounting

A program is a set of instructions and rules; accounting is based on a set of instructions and rules on how to record transactions. Hence, the translation from accounting to programming should flow naturally. But the key limiting factor here is combining the two knowledge bases, with accounting professionals hesitant to gain programming capabilities due to the perceived steep learning curve. To bridge this gap, we consider the relevance of a domain-specific language for accounting.

DSLs have been very successful in various other industries, this offers valuable insights for the accounting domain. Learning from sectors like database management and law, an accounting DSL can leverage similar principles to handle accounting-specific tasks with greater efficiency and precision.

There are many manual and repetitive tasks in accounting that can benefit from a programming language. An Accounting DSL can help improve efficiency, reduce human error and enhance accuracy.

Chapter 3

Survey Findings: Programming Among Accountants

The survey "Programming for Accountants" targeted accounting professionals from different industries with varying levels of expertise. The purpose of this survey was to gather knowledge regarding current levels of programming expertise, challenges faces in using technology and preferences and expectations for an Accounting DSL. The responses were collected from accountants with 1-10 years of experience in finance, consulting, insurance and retail sectors. Most of the respondents had little to no programming experience and were mainly using softwares like SAP, Excel, Oracle and Tally for work.

3.1 Survey Design

The survey comprised of three distinct sections to comprehensively understand the needs and the challenges faced by accounting professionals. They are:

1. Demographic Information: This section collected demographic information including years of experience and industry of work to understand diversity of the respondents.

2. Programming Experience: Questions in this section aimed at understanding the level of programming experience and identifying key challenges faced with the current technology.

3. Domain-Specific Language Preferences: This section delved into specific features and functionalities expected from an accounting DSL. The motivation was to identify types of accounting tasks that can

benefit from automation.

3.1.1 Key Focus Areas of Survey Questions

The survey was focused on several key areas to gather relevant data for the development of the domain-specific language:

- Existing Programming Knowledge and Tools
- Challenges with Current Technology
- Desired Features in the DSL
- Accounting Tasks Needing Automation
- Learning Preferences for the DSL
- Common Errors

3.2 Survey Results

The survey results provide critical insights into the current state of programming knowledge among accountants, the challenges they face with existing technology, and their preferences and expectations for a domain-specific language tailored to accounting. These findings highlight the necessity for a user-friendly, robust, and secure DSL that can address common issues and enhance the efficiency of accounting practices.

3.2.1 Current Level of Programming Experience

The survey made it clear that a majority of respondents had little to no programming experience. This backs up our hypothesis that there is not much overlap of programming knowledge and accounting skills in practice. To bridge this gap, we need a DSL that is accessible to accounting professionals without a strong background in traditional programming languages.

3.2.2 Challenges in using Existing Technology

Respondents identified the major challenges in using current technology.

- **Complexity and Technical Jargon:** Many found existing tools too complex and filled with technical jargon that is difficult to understand, making them less user-friendly for those without a technical background. This complexity often results in a steep learning curve, hindering effective use and adoption.
- **Integration Issues with Other Tools:** There is difficulty in integrating different software solutions used for various accounting tasks, leading to inefficiencies and data inconsistencies. The lack of seamless integration causes workflow disruptions and increases the risk of errors, making it harder for accountants to maintain accurate records.
- **Common and Repetitive User Errors:** High frequency of repetitive errors due to manual entry and other processes, highlighting the need for automated solutions to minimize these mistakes. These errors can significantly impact the accuracy of financial records and increase the workload of accounting professionals, leading to inaccuracy and inefficiency.
- **Lack of Certain Specific Features:** Many respondents indicated that current tools lack specific features essential for efficient accounting work, leading to gaps in functionality. This absence forces accountants to rely on multiple tools or manual processes, reducing overall efficiency and increasing the potential for errors.

3.2.3 Preferences and Expectations for an Accounting DSL

In terms of a new DSL for accounting, the survey responses showed clear preferences:

- **Simplified Syntax:** Respondents prefer a syntax that is easy to understand and use, even for those with minimal programming experience. This would ensure accessibility, allowing more accounting professionals to adopt and effectively use the DSL without extensive training.
- **Enhanced Security:** Strong security measures are essential to protect sensitive financial data, ensuring that unauthorized access and data breaches are prevented. This feature is crucial for maintaining the integrity and confidentiality of financial information.
- **Integrated Data Analysis Tools:** Tools that allow for seamless data analysis within the DSL are highly desired. These tools will enable accountants to derive insights and make informed decisions without needing additional software, streamlining their workflow.

- **Automation of Repetitive Tasks:** Features that automate common and repetitive accounting tasks can significantly reduce manual effort and the likelihood of errors, improving efficiency and accuracy. This automation will allow accountants to focus on more strategic tasks.

- **Customization:** The ability to customize the DSL to fit specific accounting needs is important for users. Customization would ensure that the language can adapt to various workflows and preferences, enhancing its utility across different accounting contexts.

- **Learning Preferences:** Respondents preferred learning through online tutorials and workshops. Providing comprehensive and accessible educational resources could help users to quickly become proficient in using the DSL, leading to better adoption and utilization.

Overall the survey results demonstrate that there is need for a Accounting DSL that is simple, easy to use, secure. integrated data analysis tools, provides automation, is scalable and offers customisation. The thesis work will focus on simplified syntax, automation of repetitive tasks and customization. The rest of the preferences and expectations are beyond the scope of this thesis and the resulting DSL will not be able to satisfy all of the preferred design features.

Chapter 4

Motivation & Design Principles for ASL

4.1 DSL vs library in a generalized programming language

The key difference between using a library in a general-purpose language like Python and designing a specialized domain-specific language like ASL lies in where the validation and verification logic resides and how deeply it is integrated into the system.

4.1.1 Key Differences

- **Integration Depth:** General purpose libraries keep validation at the application level. The database just stores data passively, while the app checks everything before and after database operations. With ASL, validation becomes part of the database language itself. The storage system actively ensures data validity during the storage process.
- **Data Integrity and Consistency:** While traditional database systems and statically typed languages afford us a generalized way to ensure that the data that we operate with is within the bounds of a type, defining a validity check that involves domain specific logic can still be complicated. For example, if we wanted to analyze the past n number of increments to a variable x and enforce the next increment to be a function of these historical changes (average, sum, etc.), it would not be a very straightforward check to implement. Baking the validation logic into the language itself allows us to actively enforce these rules on the database layer - thus avoiding the risks of storing invalid records.
- **Formal Verification:** Doing formal verification for something like a python library is cumbersome,

and requires a lot of effort to cover all edge cases correctly. In the case of ASL, formal verification techniques are part of the language's operational semantics itself, ensuring that all transactions comply with predefined invariants and rules. This can be more systematically implemented and verified.

- **Concurrency and Real-Time Auditing:** Handling concurrent transactions and maintaining audit logs in real-time in a library can be complex and error-prone when managed at the application level. In the case of ASL, the database system can natively support concurrent transaction management and real-time auditing, providing built-in mechanisms to ensure consistent and accurate audit logs. This approach facilitates systematic implementation and verification, ensuring that all transactions adhere to predefined invariants and rules.

4.1.2 ASL's Approach

1. **Integrated Validity Checks:**
    - **Syntax and Semantics:** The language syntax includes prototypes for defining transactions, flags, and audit logs. The operational semantics define how these constructs interact with the database.
    - **Built-In Constraints:** Invariants and constraints are embedded within the database language. For example, account balance checks and double-entry bookkeeping rules are enforced directly by the database.
2. **Concurrent Auditing:**
    - **Real-Time Verification:** Transactions are verified as they are processed, with audit logs generated concurrently. This ensures immediate detection of inconsistencies and potential fraud.
    - **Audit Logs:** Audit logs are maintained as part of the database, providing a tamper-evident record of all transactions.

4.2 ASL: A Database-Centric Approach to Accounting Systems

This section elaborates on the conceptualization of ASL as a language that operates on the database layer with integrated formal verification mechanisms, showing its potential as a good foundation for accounting

software development.

4.2.1 Rationale for 'Database' Approach

One of the key outcomes of the practice of accounting involves producing ledgers and audit trails. These audit-logs and transaction-logs form the corpus of data that accounting departments both consume and produce. We can then think of working with this data as similar to interacting with a database using SQL. We can think of our domain-specific-language as an analog to SQL - with accounting principles and rules baked into the language framework itself. It is more than syntactic sugar however, having these principles integrated into the language helps us formally reason about our data with confidence.

4.2.2 Key Advantages of the ASL Paradigm

1. **Centralized Validity Checks:** ASL embeds formal verification and validity checks within the database structure, ensuring data integrity independent of the accessing application. This centralization eliminates redundant validation logic across multiple applications, thereby reducing inconsistencies and errors.

2. **Uniform Data Integrity:** By enforcing a consistent set of rules and standards at the database level, ASL ensures data consistency and reliability across diverse systems and platforms.

3. **Standardization:** ASL promotes interoperability by imposing a standardized structure for accounting data. This common framework ensures uniformity in data formats and validation rules across all interacting applications.

4. **Enhanced Security and Reliability:** The centralization of validation and verification processes within the database enhances overall system security and reliability. ASL enables the implementation of rigorous access controls and audit trails, ensuring comprehensive logging and verification of all transactions.

5. **Reduced Development Overhead:** By handling complex validation rules at the database level, ASL allows developers to focus on business logic and user interface design, simplifying the overall development process.

4.2.3 Implementation Considerations

Schema Enforcement:

- We need to define a standardized schema for accounts, transactions, flags, and audit logs.
- We will need to implement database constraints to ensure that fundamental accounting rules such as non-negative balances for certain accounts and valid account types are enforced.

Centralized Auditing:

- Producing audit logs as a runtime artifact, ensuring real-time recording and verification of all transactions.

Triggers and Stored Procedures:

- Implementation of validation triggers - account specific fragments of code to verify transactions prior to commit.
- Utilization of stored procedures to wrap more complicated validation logic, ensuring consistency and reusability. This is beyond the scope of current work, but would be a good avenue for future work.

Formal Verification Tools:

- Define invariants that must be maintained across all valid states of the program.
- Application of a modified type system to ensure compile-time type-checking.

Chapter 5

Core Examples for Accounting

This chapter provides a fundamental overview of key accounting concepts through practical examples. It is designed to help readers understand the types of transactions and entries that are commonly encountered in accounting.

We start with the cash account, highlighting the critical rule that the net balance of a cash account can never be negative. This non-negativity constraint is embedded directly in our DSL to prevent invalid transactions. Next, we discuss depreciable assets and demonstrate how depreciation is calculated and recorded. Our DSL includes a flag system to handle depreciation, ensuring that assets are depreciated correctly over their useful life.

While we touch on inventory valuation and the accrual basis of accounting, these concepts are not yet fully implemented in ASL. Inventory valuation is acknowledged as a complex, accounting-adjacent topic that falls more within the realm of costing. As such, it is earmarked for potential future work rather than current implementation. Similarly, the accrual basis of accounting cannot be enforced as a ”check” within the current capabilities of our DSL. Adherence to accrual accounting principles remains the responsibility of the DSL user at this stage.

By the end of this chapter, readers will have a clear understanding of basic accounting entries and transactions, providing the necessary context for the implementation details and further discussions in subsequent chapters.

5.1 Cash

Cash account is probably the most common account found in all accounting ledgers. There is one very important universal rule associated with any Cash account. At any given point of time the net balance of a cash account can never be less than zero. This is not a flag, but rather a restriction placed upon the user such that a negative Cash balance should be impossible to achieve.

Example:

- Initial Cash Balance : 0
- Debit Cash 100 Credit Bank 100
- Current Cash Balance : 100
- Debit Inventory 10 Credit Cash 10
- Current Cash Balance : 90
- Debit Rent 120 Credit Cash 120
- ERROR : Insufficient Cash Balance

5.2 Depreciable Assets

A specific type of asset whose value declines over the period with use are called depreciable assets. The 'depreciation' on these is calculated using a specific formula. For this calculation we need details like the purchase price, the type of depreciation method to be used, the number of years over which to depreciate the asset, and the expected resale value of the asset. The journal entry for 'depreciation' can be thought of as a 'process' that is carried out every month over the lifespan of the asset.

Example:

- Asset: Company Vehicle
- Purchase Price: $50,000
- Depreciation Method: Straight-Line Depreciation

- Useful Life: 5 years
- Residual Value (Salvage Value): $5,000
- Depreciation Calculation:
  - Straight-Line Depreciation Formula:
  - Annual Depreciation Expense = (Purchase Price - Residual Value) / Useful Life
  - Monthly Depreciation:
  - Annual Depreciation= ($50,000 - $5,000) / 5 = $9,000
  - Monthly Depreciation = $9,000 / 12 = $750
- Monthly Entry:
- Debit Depreciation 750 Credit Vehicle 750

5.3 Inventory

While inventory greatly varies across various entities or businesses, there are only a handful of methods to value inventory at the end of the financial year. Some of these are First-In-First-Out, Last-In-First-Out, etc. We would like to collect this information (which valuation method does this inventory account follow) and store as a property of the account. The valuation methods are functions or methods that have an impact on the reported profit/loss of the entity for the year. The valuation of closing inventory is currently not supported by ASL, however this is an interesting avenue for future work.

Example: First-In-First-Out Laptop Business

- Day 1: Bought 10 laptops @ $500 each
- Day 15: Bought 5 laptops @ $550 each
- Sale Transactions:
- Day 20: Sold 12 laptops
- Sold Inventory (FIFO):

  - 10 laptops @ $500 (from Day 1 purchase)
  - 2 laptops @ $550 (from Day 15 purchase)

- Remaining Inventory:
- 3 laptops @ $550 = $1650

5.4 Accrual Basis of Accounting

Accounting follows 'accrual' method of recording transactions. This means that incomes, expenses and losses (not profits) are recorded in the books when and only when they are due. An example of this: expenses for the whole year paid all at once are recorded as 'Prepaid expenses' and are written off as an expense proportionately spread across the year. To formalise this behaviour in ASL, we would need special constructs in place that will allow us to collect all the necessary information, we would also need the ability to contextually determine the time period the expense pertains to. This is currently beyond the abilities of ASL and the onus of adhering to accrual basis of accounting falls on the user of ASL at this stage.

Expense: Annual Office Rent

- Total Payment: $12,000 (paid on 01/01)
- Rental Period: 12 months
- Monthly Rent Expense: $1,000 per month ($12,000 / 12 months)
- Journal Entries:
- At the time of payment:
  - Debit Rent Expense 1000 Credit Bank 1000
  - Debit Prepaid Rent 11000 Credit Bank 11000
- Monthly Journal entry from 02/01 to 12/01:
- Debit Rent Expense 1000 Credit Prepaid Rent 1000

5.5 End-to-End Accounting Ledger of a dummy enterprise

In this section, we present a comprehensive example of an accounting ledger for a hypothetical enterprise. This end-to-end illustration covers various transactions over a period of months, showing how different types of entries interact and affect the overall financial position of the business. The example aims to help solidify the understanding of the fundamental accounting concepts discussed earlier in the chapter.

5.5.1 October

- **Capital introduced Cash Balance**: $80,000.
- **Machinery Purchase with Cash**: Acquired manufacturing equipment for $20,000, to be depreciated over 10 years with a salvage value of $2,000.
- **Office Equipment Purchase with Cash**: Acquired office furniture and computers for $5,000, to be depreciated over 5 years with no salvage value.
- **Raw Materials Purchase with Cash**: Bought raw materials for production costing $3,000.
- **Rent Payment with Cash**: Paid annual office rent of $24,000 (recorded as prepaid rent for next year).
- **Salaries paid with Cash**: Paid $8,000 in monthly salaries to employees.
- **Utility Bills paid with Cash**: Paid $800.
- **Monthly Depreciation**:
  - **Machinery**: $150
  - **Office Equipment**: $83.33
- **Closing Cash Balance**: $80,000 - $20,000 - $5,000 - $3,000 - $24,000 - $8,000 - $800 = $19,200

5.5.2 November

- **Starting Cash Balance**: $19,200
- **Production**: Used $2,000 worth of raw materials to manufacture goods.
- **Customer Invoices**: Issued invoices totaling $20,000 for products delivered on credit.

- **Utility Bills paid with Cash**: Paid $800.
- **Salaries paid with Cash**: Paid $8,000 in monthly salaries.
- **Monthly Depreciation**:
  - **Machinery**: $150
  - **Office Equipment**: $83.33
- **Closing Cash Balance**: $19,200 - $800 - $8,000 = $10,400

5.5.3 December

- **Starting Cash Balance**: $10,400
- **Product Sales**: $40,000
- **Equipment Purchase with Cash**: $25,000, depreciated over 5 years with a salvage value of $5,000.
- **Salaries paid with cash**: $8,000
- **Employee Bonuses**: Issued bonuses totaling $4,000 to employees for exceeding production targets.
- **Cash Collection on Invoices**: Collected $15,000 against the invoices issued in November.
- **Utility Bills paid with Cash**: $800
- **Monthly Depreciation**:
  - **Machinery**: $150
  - **Office Equipment**: $83.33
  - **Delivery Vehicle**: $333.33
- **Tax Payment**: Paid quarterly taxes estimated at $3,000.

Closing Cash Balance: $10,400 + $40,000 - $25,000 - $8,000 - $4000 + $15000 - $800 - 3000 = $24,200.

5.6 Ledger Entries

5.6.1 October Ledger Entries

1. **Cash Entry**
    - **Debit**: Cash $10,000
    - **Credit:** Capital $10,000
2. **Machinery Purchase**
    - **Debit**: Machinery $20,000
    - **Credit**: Cash $20,000
3. **Office Equipment Purchase**
    - **Debit**: Office Equipment $5,000
    - **Credit**: Cash $5,000
4. **Raw Materials Purchase**
    - **Debit**: Inventory $3,000
    - **Credit**: Cash $3,000
5. **Prepaid Rent Payment**
    - **Debit**: Prepaid Rent $24,000
    - **Credit**: Cash $24,000
6. **Salaries Payment**
    - **Debit**: Salaries Expense $8,000
    - **Credit**: Cash $8,000
7. **Utility Bills Payment**
    - **Debit**: Utility Expense $800
    - **Credit**: Cash $800

8. **Machinery Depreciation**
    - **Debit**: Depreciation Expense $150
    - **Credit**: Accumulated Depreciation - Machinery $150
9. **Office Equipment Depreciation**
    - **Debit**: Depreciation Expense $83.33
    - **Credit**: Accumulated Depreciation - Office Equipment $83.33

5.6.2 November Ledger Entries

1. **Production Expense**
    - **Debit**: Cost of Goods Sold $2,000
    - **Credit**: Inventory $2,000
2. **Customer Invoices Issued**
    - **Debit**: Accounts Receivable $20,000
    - **Credit**: Sales Revenue $20,000
3. **Utility Bills Payment**
    - **Debit**: Utility Expense $800
    - **Credit**: Cash $800
4. **Salaries Payment**
    - **Debit**: Salaries Expense $8,000
    - **Credit**: Cash $8,000
5. **Machinery Depreciation**
    - **Debit**: Depreciation Expense $150
    - **Credit**: Accumulated Depreciation - Machinery $150

6. **Office Equipment Depreciation**

    - **Debit**: Depreciation Expense $83.33
    - **Credit**: Accumulated Depreciation - Office Equipment $83.33

5.6.3 December Ledger Entries

1. **Product Sales**

    - **Debit**: Cash $40,000
    - **Credit**: Sales Revenue $40,000

2. **Equipment Purchase**

    - **Debit**: Equipment $25,000
    - **Credit**: Cash $25,000

3. **Salaries Payment**

    - **Debit**: Salaries Expense $8,000
    - **Credit**: Cash $8,000

4. **Employee Bonuses**

    - **Debit**: Salaries Expense $4,000
    - **Credit**: Cash $4,000

5. **Cash Collection on Invoices**

    - **Debit**: Cash $15,000
    - **Credit**: Accounts Receivable $15,000

6. **Utility Bills Payment**

    - **Debit**: Utility Expense $800
    - **Credit**: Cash $800

7. **Machinery Depreciation**

    - **Debit**: Depreciation Expense $150
    - **Credit**: Accumulated Depreciation - Machinery $150

8. **Office Equipment Depreciation**

    - **Debit**: Depreciation Expense $83.33
    - **Credit**: Accumulated Depreciation - Office Equipment $83.33

9. **New Equipment Depreciation**

    - **Debit**: Depreciation Expense $333.33
    - **Credit**: Accumulated Depreciation - New Equipment $333.33

10. **Tax Payment**

    - **Debit**: Tax Expense $3,000
    - **Credit**: Cash $3,000

Chapter 6

Flags in Accounting

6.1 Motivation for Accounting Flags

Over the years, numerous frauds have been covered up by manipulating the books of accounts in various manners. Our goal is to design 'flags' at appropriate places. If the reporting of such flags and the explanations for the same were made a part of the public record, it can not only increase transparency but also assuredly bring discrepancies to notice.

Example: The Enron Scandal

In 2001, the then energy sector giant company Enron collapsed due to extensive corporate fraud and corruption. One of the main issues for such a massive collapse was the manipulation of complex accounting loopholes and special purpose entities to hide Enron's massive debt and inflate profits. Special Purpose Entities or Vehicles (SPV) are separate legal entities created by the parent company for a specific purpose that are mostly used to record transactions and keep them off the parent company's books. While there are accepted ways of using SPVs, in this case, they have been used to manipulate the financial books.

Key Issue: Hiding Debt and Inflating Profit **Manipulation:** Enron used off-balance-sheet special purpose vehicles (SPVs) to hide its debt and liabilities. This made the company look more profitable and less risky than reality. Enron also wrongly reported inflated revenues and profits.

Potential 'Flag' system to prevent such a scandal:

Flags for the following can be added to the language.

1. Unusually High Transactions with Unrelated Entities
2. Rapid Increase in Revenue Without Corresponding Cash Flow
3. Large Discrepancies Between Reported Profits and Cash Flows
4. Excessive Use of Complex Financial Instruments
5. Frequent Changes in Accounting Policies

Outcome: If such a flag system had been in place and actively monitored at Enron, it could have alerted stakeholders, auditors, shareholders, and regulatory bodies, to the discrepancies and unusual practices. This early detection could have potentially prevented the magnitude of the scandal.

The flags in the DSL would ensure transparency if they were made a mandatory part of the public record, along with appropriate explanations for violations of the flags. Since the nature of the flags is closely tied to the nature of the business, the flags will need to be customizable by the users. Flags can essentially be thought of as 'warnings' and not errors; they can either be corrected or provided appropriate explanation for.

6.2 Concurrent Auditing in the form of 'Flags'

Concurrent audit is a systematic examination of financial transactions on a regular basis to ensure accuracy, authenticity, compliance with procedures and guidelines. It is a modern take on the traditional audit process, where traditional auditing principles are combined with real-time data processing and analysis. In this method, an organization's transactions are audited in real-time or near-real-time to validate that they are accurate, authentic, and comply with all rules and regulations that have been defined.

Concurrent auditing allows to detect non-compliant transactions in real-time instead of after-the-fact. For example, if an account's home rental transaction suddenly goes up by 100% of its historical value, that anomaly should be flagged in the concurrent audit. Another example can be in the form of accounts that require a minimum cash balance to constantly be maintained - if the cash balance goes below a certain threshold, we want to be able to catch that in real-time using concurrent-auditing technique.

Concurrent auditing is a defining feature of ASL, implemented in the form of 'Flags'.

6.3 Concept of Flags

Flags in ASL are conceptualized as a first-class construct, analogous to type system in statically-typed programming languages. They serve as invariants that must hold true for each state transition in the accounting system.

For the context of this thesis, we will look at flags as an abstract function.

Each transaction in ASL is treated as a state transition. Before a transition is finalized, all relevant flags are evaluated.

The system maintains a set of active flags $F = \{f_1, f_2, \ldots, f_n\}$ for each account or entity.

For each state transition $T : S \to S'$, where S is the current state and S' is the proposed new state, the system evaluates: $\forall f \in F, f(S')$ must evaluate to true

If any flag in our set is raised (or gets evaluated as a false value), the program state is considered invalid.

In cases where a flag needs to be overridden, the system requires an explanation/evidence, which is then logged automatically as part of an audit trail.

6.4 Abstraction for the Flag Function

We have not defined the syntax for our Flag system as a part of this work - however we seek to implement flags initially as a predicate that operates on the entire transaction history and current transaction of the accounting system. In the context of ASL's syntax, a flag is a function that takes all the logs, sets and maps, and a transaction as an input and returns a boolean value as the output.

6.4.1 Why Comprehensive Access is Essential for Flags:

1. Historical Context and Trend Analysis

Flags need access to the entire transaction history to identify trends and historical patterns. For example, a flag can detect unusual increases in expenses or revenue that deviate from historical norms, suggesting possible manipulation or errors. Understanding past behavior is crucial for recognizing anomalies that could indicate fraudulent activities.

2. Multi-Dimensional Analysis

Accounting practices often involve complex relationships between various financial components. Flags must analyze these relationships across multiple dimensions, such as time, transaction type, and involved entities. Comprehensive access enables flags to cross-verify data, ensuring that financial statements accurately reflect the organization's financial position.

3. Real-Time Monitoring and Prompt Response

To be effective, flags must provide real-time monitoring capabilities. By having full access to the transaction history and current transactions, flags can promptly detect and report discrepancies, allowing for immediate corrective actions. This real-time aspect is critical in preventing minor issues from escalating into significant problems.

4. Ensuring Compliance and Policy Adherence

Organizations must adhere to various accounting standards and policies. Flags help ensure compliance by continuously checking transactions against these standards. For example, flags can verify that all transactions conform to GAAP (Generally Accepted Accounting Principles) or IFRS (International Financial Reporting Standards). They can also enforce internal policies, such as limits on expenditures or requirements for maintaining specific cash reserves.

5. Detecting Sophisticated Fraud Schemes

Fraudsters often use sophisticated methods to conceal their activities, such as layering transactions to obscure their origins or using complex financial instruments. Flags with comprehensive access can detect such schemes by analyzing the flow of funds and the structure of transactions, identifying inconsistencies or patterns typical of fraudulent behavior.

6. Facilitating Detailed Audit Trails

Audit trails are essential for investigating financial discrepancies and ensuring accountability. Flags that operate on the entire transaction history can create detailed logs of all financial activities, including flagged

transactions and the reasons for flagging them. These audit trails provide a transparent record that auditors and regulators can review to verify the accuracy and integrity of financial statements.

6.4.2 Conceptual Model of Flag Function

In the conceptual model, a flag is viewed as an invariant or rule that must hold true for all state transitions within the accounting system after the flag is introduced. Each state transition is evaluated against a set of predefined flags, ensuring that any deviation from the expected behavior is immediately identified and addressed.

6.4.3 Evaluation of Flags

For each state transition $T : S \xrightarrow{tx} S'$, where S is the current state, tx is the new transaction and S' is the resultant new state, the system evaluates: $\forall f \in F, f(S, tx)$ i.e. it evaluates all the flags against the current state and the new transaction and the resultant state S' must evaluate to true.

This means that for the transition to be valid, every flag in the set of active flags F must hold true in the new state S'. If any flag evaluates to false, the transition is considered invalid, and appropriate actions are taken, such as logging the issue, notifying relevant personnel, or even reverting the transaction. However, in ASL, flags are mainly in the form of 'warnings' and can be overriden. If a flag returns true, it must be added to the override log with an explanation.

6.4.4 Customization and Flexibility of Flags

The flag system must be flexible and customizable to cater to the unique requirements of different organizations and industries. Users should be able to define their own flags based on specific criteria relevant to their operations. This customization allows the flag system to be adaptable and effective across various contexts, ensuring that it remains a powerful tool for maintaining financial integrity.

The abstraction for the flag function emphasizes the importance of comprehensive access to all log components to ensure robust and effective monitoring of accounting practices. By evaluating each state transition against a set of predefined flags, the system can detect and address discrepancies in real-time, enhancing transparency, compliance, and the overall integrity of financial reporting.

6.5 Consistency (Future Work)

The system maintains a set of active flags $F = \{f_1, f_2, \ldots, f_n\}$.

Flags f_1 and f_2 are inconsistent if there is no state at which they are both true. This property should be evaluated every time a new flag gets registered, or a new transaction arrives in the system, and ensures that we do not have any flags that contradict each other. For example, the following set of flags are considered inconsistent in the system-

```
flag MinimumCashBalance {
  condition: account.cash >= 1000
  severity: ERROR
}

flag MaximumCashBalance {
  condition: account.cash < 1000
  severity: ERROR
}
```

This is currently not enforced by ASL since flags are an abstract function and we have not specified the syntax and semantics for flags' internal workings. However, this is an interesting avenue for future work.

Chapter 7

Logging System

7.1 Introduction

ASL's logging system is designed to bridge the gap between traditional programming paradigms and the unique requirements of accounting systems. While both domains share foundational principles of logic and data computation, their approaches to state management and error handling differ significantly. Unlike general-purpose programming languages, mostly concerned with the final state of the program, the ASL runtime maintains a complete record of all intermediate states. This is implemented as an append-only log structure.

While tools like compilers aim to reject invalid programs, ASL's logging system is designed to capture and record errors, misses, and mistakes. These mistakes or errors are not a part of what would make our program "invalid", but are rather anticipated "accounting errors" due to human error or such. This is achieved via the semantics for corrections.

Logs in ASL are not just for debugging or auditing; they are first-class data that the program operates on. This allows for powerful querying and analysis capabilities. The logging system ensures that all entries are temporally consistent, allowing for time-based analysis.

There are different types of logs that form the corpus of data that ASL operates on, we will discuss them in the upcoming section.

7.2 Types of Logs in ASL

Accounting logs serve multiple critical functions:

1. **Historical Record**: They provide a historical record of all financial transactions, which is essential for auditing, compliance, and financial analysis.
2. **Verification and Validation**: Logs ensure that transactions are recorded accurately and can be verified and validated against original documents and statements.
3. **Transparency and Accountability**: Detailed logs promote transparency and accountability, allowing stakeholders to trace and understand the financial activities and decisions.
4. **Error Detection and Correction**: Logs capture errors and allow for their correction through appropriate entries, maintaining the integrity of the financial data.

Let us look at the types of logs in ASL.

7.2.1 Transaction Input Stream

This simply refers to the raw transaction events that get fed into our program. Each transaction event in the input stream is recorded along with the timestamp on arrival.

All the subsequent logs that the ASL runtime generates is a by-product of this stream of data.

The transactions in this stream are entered by the end users (accountants) and are not guaranteed to be valid, and many may have "corrections" applied to them after, so simply using it does not guarantee that we have a valid system.

7.2.2 True Log

The 'true' log refers to the canonical log of transactions that have corrections and audits accounted for. It derives from the combination of Transaction Input stream, corrections logs, and Override logs. The true log places past corrections to transactions in the correct temporal order, ensuring that the final output is valid and consistent. This highlights an aspect of the ASL runtime that is significant - ASL allows for, and accepts the fact that accounting frequently involves errors, misses and out of order events. The existence of which

does not make the accounting system invalid. The true log can be considered as the final artifact produced by any program written in ASL.

7.2.3 Account Balance Logs

ASL supports different types of accounts:

T := { Asset, Liability, Income, Expense, Capital, Cash, Depreciable Asset, Depreciation, Inventory, User-defined }

Each account may have a 'balance' property whose value is derived from the transaction logs coming via the input stream. The balance logs can be formalized as follows

B := · | B,act $\mapsto$ (c,T) (Account Bal/Type Log)

7.2.4 Flag / Override Log

ASL allows creation and storage of a list of flags. The flags enable us to perform concurrent accounting as described in the previous section.

Each transaction in the input stream of an ASL program is evaluated against a list of registered Flags. A particular transaction is only considered valid within the context of the accounts it is operating within if all the predicates defined by the flags in our system pass.

In cases where the predicate is not satisfied, the user has the option of overriding the Flag predicate with appropriate explanation/ evidence.

All of the events described above are collated into ASL's runtime audit log.

Hence, the audit log comprises of the following events :

- **New Flag** Every time a new Flag construct is created within our context, we would need a log to document it.
- **Flag Raised For Transaction**: If a Flag is raised for a transaction, our program is considered invalid, however, this can be 'overridden'.
- **Flag Overriden**: If a flag gets overridden after it is raised, we document the explanation/evidence to support this claim.

Our flag/override logs can be formalized as follows:

Flag Logs (A new flag is introduced to our system)

F := · | F, f (Flag Log)

Override Logs (A new flag is overriden by the user)

Each override log is accompanied with a note - a string representing the explanation/evidence of the override.

V := · | V, (tx, f, note, DT) (Override Log)

Chapter 8

Concept of 'Time' in Accounting

Time is a cornerstone concept in accounting, serving as the foundation for numerous critical practices and principles and should be given careful consideration during the development of an Accounting DSL. The DSL should be able to handle time and the idea of 'time passing' not just as a peripheral concept but rather as an integral part of the language structure.

The concept of time is core to accounting, playing a crucial role in categorizing financial activities, delineating reporting periods, and underpinning the assessment of financial health over specified intervals. This temporal dimension influences all aspects of accounting, from daily transaction recording to strategic fiscal planning. It is instrumental in ensuring that financial statements provide an accurate and timely representation of an organization's economic activities, thereby facilitating informed decision-making by stakeholders.

The integration of time as a fundamental axis in Accounting Domain-Specific Languages (DSLs) is essential for accurately modeling the temporal dynamics inherent in accounting practices. By integrating time-based constructs directly into the DSL, developers and accountants can more effectively model financial processes, stick to reporting cycles, and execute time-sensitive financial transactions. This integration ensures that the DSL not only accommodates but also enhances the temporal aspects of accounting.

The following sub-sections will briefly cover principles of accounting which have a temporal aspect to them.

8.1 Temporal Significance in Accounting

8.1.1 Periodicity Principle

Accounting divides financial activities into specific time periods (e.g., months, quarters, years) for reporting and analysis purposes. This temporal segmentation is crucial for assessing financial performance and position over time.

8.1.2 Accrual Basis Accounting

This method recognizes economic events regardless of when cash transactions occur. It relies heavily on the concept of time to match revenues with expenses in the appropriate periods. Only the accrual accounting method is allowed by generally accepted accounting principles (GAAP). Public companies within the U.S. must adhere to the Generally Accepted Accounting Principles (GAAP) as decided by the Financial Accounting Standards Board (FASB). Therefore, such businesses are required to use accrual-basis accounting, since it meets GAAP standards.

Conversely, the cash basis recognizes revenues and expenses only when cash is received or paid. An Accounting DSL must be versatile enough to support both accounting bases, offering syntactic structures and functions that allow users to specify the timing of transaction recognition. This flexibility is crucial for accurately modeling the financial realities of different types of businesses and ensuring compliance with various accounting standards and principles.

8.1.3 Fiscal Years and Reporting Cycles

Organizations often operate on fiscal calendars that may not align with the calendar year, requiring sophisticated handling of time-based operations. Having the ability to define these calendars as part of your program in an easy, convenient manner would be very useful from a programming perspective.

8.1.4 Time-Dependent Valuations

The calculation of depreciation and amortization involves time-based schedules that reduce the value of assets over their useful lives. Accounting DSLs can include predefined functions for common depreciation methods, simplifying what would otherwise require complex setup in a general-purpose language.

8.1.5 Audit Trails and Compliance

Establishing an audit trail for compliance purposes requires that we must maintain extensive and accurate audit logs that have been correctly timestamped. Audits are performed not just for compliance purposes, but can be useful in rooting out inefficiencies, detecting fraud and goes about giving a full financial picture of the organization.

8.2 Time in ASL

In the Accounting Specification Language (ASL), time is not merely a feature but a fundamental construct integrated into the core of the language. This is evident from the ubiquitous presence of the DT (DateTime) variable across various log structures.

8.2.1 Time-Centric Log Structures

Input Stream (Δ): The input stream is fundamentally time-ordered in the order of when the transactions are 'recorded'

Transfer Transaction Log (T): The transfer transaction log maintains its records in a temporal order, in the order the transactions have 'occurred', unlike the input stream

This allows for precise tracking of when transfers occur, crucial for reconciliation and audit trails.

Override Log (V): The override log, also known as the audit-log is used to track every time a flag in our system gets raised for a transaction and is explicitly overridden by the user with a mandatory string of explanation or evidence. These logs also are timestamped to denote the time at which the Flag was raised for a transaction and was overridden

Chapter 9

Syntax and Semantics for Accounting DSL

In this chapter, we will introduce the syntax and semantics for our language, Accounting Storage Language (ASL). ASL is less a language for computation and more a data format language with guarantees about the well-formedness of the resulting data. The data is in the form of a permanent log recording all the transactions, the balances, the account types, the flags in place and the override notes. We will cover all these in detail in the later part of this chapter. The primary focus in designing ASL has been on defining when a permanent log is well-formed.

Accounting can be best described as an act of recording activities or 'transactions' that have a financial or monetary impact. Because of such a broad scope, it is very difficult to enforce rules that can apply to all possible transactions. A big chunk of effort towards the design of ASL has been trying to strike a balance between incorporating validity checks and keeping the language flexible.

Here, we discuss our definition of 'well-formedness' within the scope of ASL. Let us look at an example of two types of 'wrong' behaviors:

1. transfer Cash Machinery $1000, 07/11/2024: This transaction is reducing our Cash account and increasing our Machinery account by $1000 signifying cash purchase of the same amount. However, in reality the Machinery cost was $2500 and we paid all of that via Cash. So in this context, this is a "wrong" entry or transaction record.

2. transfer Cash $1000: This transaction is only focusing on reducing Cash account, however, there is no other account being mentioned into which this amount can be added. This is also a "wrong" entry or transaction record.

ASL follows double-entry accounting which has two fundamental rules:

1. Every debit will have a corresponding equal Credit.
2. Cash can never be negative.

Apart from these, the three ”Golden Rules” are:

1. Debit all expenses and losses, Credit all incomes and gains
2. Debit the receiver, credit the giver
3. Debit what comes in, Credit what goes out

After examining these rules, let us take another look at our examples of ”wrong” entries. In the first example, while it is still a wrong entry, it does not break any fundamental rules of accounting and can be attributed to human errors, which are fairly common in accounting practice. However, the second example does break fundamental rules of double-entry accounting. In ASL, we allow, expect and provision for the first type of ”mistake” and guarantee that the two fundamental rules of accounting are upheld as a part of our validity check.

While the two fundamental rules are straightforward to incorporate, the three ”Golden Rules” pose a bigger challenge. How does one classify expenses and losses vs incomes and gains? We attempted to introduce a type system modeled after type systems in statically typed languages to address this, but static types proved too rigid for this domain. In accounting, while it is true that it is a ”rule” that all expenses must be debited, there are several scenarios in which expenses need to be credited too. One example of such a scenario is when you have made a mistake and need to rectify it via a ”Contra” Entry, which is the reverse of the original entry to nullify its effects. For the second rule, a single entity or account can be the 'receiver' in one entry and can be the 'giver' in another entry. Here, again, we cannot assign a static type (giver/ receiver) to any account since its type can change based on the context. The third rule, ”Debit what comes in, credit what goes out,” is even more challenging. This rule dictates that any incoming goods, items, or funds should be debited, while anything paid out, sold, or given away should be credited. Incorporating this rule into the verification process is also very complex due to its broad applicability and the dynamic nature of transactions. To ensure compliance with this rule, ASL would need to track and classify all incoming and

outgoing entries. This requires a detailed understanding of the context of each transaction, which can vary significantly.

The complexity arises from the necessity to dynamically assign debit or credit status based on the specific circumstances of each transaction, which can result in making the language too rigid and complex. Hence, in ASL, we are only making guarantees about the fundamental rules of accounting and we are choosing to allow ”mistakes” in the form of human errors. This is because accounting as a discipline also has provisions for these mistakes, in the form of Adjusting Entries, Reversing Entries or Contra Entries (journal entries to reverse the effects of previous wrong entries). We are modelling this behaviour in ASL through our ”correction” transaction. A correction transaction in the context of ASL is a way of correcting the accounting logs to revert any previous ”mistakes” made by the end user (accountant). This correction transaction can be any transfer entry that either reverses a previous entry or complements a previous entry to reflect correct books of accounts.

If we look at our previous example again:

transfer Cash Machinery $1000: This transaction is reducing our Cash account and increasing our Machinery account by $1000 signifying cash purchase of the same amount. However, in reality the Machinery cost was $2500 and we paid all of that via Cash.

We can now pass the following entry to rectify this mistake:

correction transfer Cash Machinery $1500 07/11/2024, 07/24/2024

Here, 07/11/2024 is the date of the original entry which we are rectifying and 07/24/2024 is the date when this correction entry is written by the end user. Due to this correction entry, our books of accounts will now be aligned with the reality of the transaction. Later in this chapter, we will take a more detailed look at how ASL handles correction transactions.

Also, to minimize the risks due to malicious or fraudulent misrepresentation and undetected errors that may not reflect the true nature of transactions as they have happened in real life, we have introduced a ”flag” system in ASL. Flags are currently designed in ASL as a set of user-defined predicates that help in facilitating the detection of inconsistencies and irregularities. For the context of this thesis, we will look at flags as an abstract function that takes as input all the permanent accounting logs produced as a result of a set of processed transactions and a new transaction that is to be added to the log; and returns a boolean value. Currently, we have only focused on flags that are akin to ’warnings’ and can be overridden.

Now that we have seen the challenges and limitations faced in developing ASL, we can define what

soundness or well-formedness mean in the context of ASL. ASL gives the following guarantees about a valid program that does not get stuck:

1. Every debit will have a corresponding equal credit.
2. Cash will never be negative.
3. The books of accounts will always be balanced i.e. sum of all account balances will always be 0.
4. All transfer transactions will either
   (a) Not turn any flag present at the time of the transfer active OR
   (b) Will document the reason or explanation in the override log for any and all flags that have been turned active

We can now take a look at the syntax and operational semantics of ASL.

9.1 Expressions

Transfer Transaction $\mathbf{tf}$:= transfer act_1 act_2 amt | $(tf; tf)$

Transactions $\mathbf{tx}$:= tf | create_acc Ty act_1 | create_ty $type$ | create_flag f

| delete_flag f | tf; override_flag $(f_0, f_1, \ldots, f_n)$ note

| correction tf DT | current_time

Account Type $\mathbf{Ty}$:= Asset | Liability | Income | Expense

| Capital | Cash | $type$

Where:

$f : L \times B \times T \times F \times V \times tx \rightarrow$ Boolean

DT : DateTime Format

$act_1, act_2, flag, type \in$ AlphaNumeric Characters

$c \in \mathbb{Z}$

$amt \in \mathbb{Z}^+$

"tf" is a transfer transaction, where funds are transferred from one account (act_1) to another account (act_2) for a specified amount (amt). Multiple transfers can be grouped together using the $(tf; tf)$ syntax, allowing for simultaneously occurring transactions.

"tx" are various types of other ancillary transactions supported by this DSL, few of which are self-explanatory.

"flag" is a function that takes as input all the logs and a transaction tx, and returns a boolean.

With the override_flag transaction you can override any number of flags with an explanation note which will be a part of the override (concurrent audit) log.

The correction transaction accounts for human errors and allows for pre-dated 'corrections' which are transfer entries with DT representing the date and time to which the correction pertains.

"current_time" is a placeholder transaction which can be used to check all the flags at any time without any transfer entry.

Asset, Liability, Income, Expense, Capital and Cash are pre-defined account types. All accounts fall under at least one of these types. Having them as a part of our syntax guarantees that all basic account types are accessible to the flags. Flags can use types to perform checks on validity.

9.2 Logs, Maps and Set

Input Stream $\mathbf{\Delta} := \cdot \mid \Delta, (tx, DT)$

True Accounting Log $\mathbf{L} := \cdot \mid L, (tf, DT)$

Accounts Balance and Type Map $\mathbf{B} := \cdot \mid B, act \mapsto (c, Ty)$

Built-In and User Defined Types Set $\mathbf{T} :=$ Asset, Liability, Income, Expense,

Capital, Cash, Empty $\mid T, type$

Flags Set $\mathbf{F} := \cdot \mid F, f$

Override Log $\mathbf{V} := \cdot \mid V, (tf, f, note, DT)$

In general-purpose programming languages, the focus is primarily on computation and achieving a final result through various operations and transformations. The intermediate states and steps leading to the

final result are often secondary and may not be explicitly recorded. However, in accounting, the situation is fundamentally different. Accounting is not merely about computation; it is the act of creating a written record of financial transactions. This written record, or log, is both the action and the product of accounting. Therefore, logs are crucial in the accounting domain. Accounting logs are vital as they provide a historical record for auditing, compliance, and financial analysis; ensure accurate recording and validation of transactions; enhance transparency and accountability for stakeholders; and detect and correct errors to maintain data integrity.

ASL's logging system is designed to meet these needs by maintaining comprehensive and consistent records of all transactions and related actions. Unlike traditional programming paradigms that may only retain the final state, ASL logs all intermediate states and changes, providing a complete picture of the financial data's evolution over time. This approach aligns with the core principles of accounting and ensures that all entries are traceable, verifiable, and auditable.

9.2.1 Types of Logs in ASL

- **Input Stream (Δ)**

  The Input Stream log captures all incoming transactions in the exact order they are written or recorded by the user, with timestamps that reflect when the transaction was recorded (not necessarily when the event occurred.) This log serves as the initial raw data input for the ASL runtime, preserving the sequence of transaction events based on user input times.

- **True Accounting Log (L)**

  The True Accounting Log is the canonical log of transactions, accounting for all corrections and audits. It reflects the final, validated state of the transactions after considering any adjustments or corrections. This log ensures that the recorded transactions are consistent and accurate, representing the true financial state in chronological order.

- **Accounts Balance and Type Map (B)**

  This map maintains the balances and types of all accounts. It records the current balance of each account along with its type (e.g., Asset, Liability). This map is crucial for tracking the financial position and ensuring that all accounts are correctly categorized and balanced.

- **Built-In and User Defined Types Set (T)**

  The Types Set keeps track of all built-in and user-defined account types. It includes standard types such as Asset, Liability, and Income, as well as any custom types defined by the user. However, only the 'Cash' type has native support in ASL's semantics, the rest are included to make sure every account can be attributed a factually correct type since flags rely on types. This set also allows for flexibility in defining and using various account types within the ASL system.

- **Flags Set (F)**

  The Flags Set records all the flags defined within the system. Flags are predicates used to detect inconsistencies or irregularities in transactions. This collection helps in concurrent auditing by evaluating transactions against these flags and ensuring they meet predefined conditions.

- **Override Log (V)**

  The Override Log captures instances where flags have been overridden. It includes the transaction that triggered the flag, the specific flag, the note explaining the override, and the timestamp. This log ensures transparency and accountability when exceptions are made to the standard rules.

These logs form the foundation of ASL's logging system, ensuring that all transactions are accurately recorded, analyzed, and validated. They provide a robust framework for maintaining the integrity and reliability of the accounting data.

9.3 Judgements

9.3.1 Retrieving Accounts Balance and Type Information from Log B

$$\frac{B', act_1 \mapsto (c, Ty)@act1 \rightsquigarrow (c, Ty)}{B@act_1 \rightsquigarrow (c, Ty)}$$

The judgments defined above are used to retrieve the balance and type information of a specific account (act_1) from the map B, which maintains all account balances and their types.

The Hit judgment occurs when the account being queried (act_1) is found in the map. The notation $B', act_1 \mapsto (c, Ty)$ indicates that act_1 is associated with balance c and type Ty. When queried, this entry directly returns the balance and type, (c, Ty).

9.3.2 Input Stream Evaluating to All Logs

Empty Input

$$\frac{}{\cdot \Downarrow \cdot;\cdot;\cdot;\cdot;\cdot}$$

General

$$\frac{\forall dt_1 \in \Delta, DT \geq dt_1 \quad tx \neq \text{correction} \quad \Delta \Downarrow L'; B'; T'; F'; V' \qquad L'; B'; T'; F'; V'; (tx, DT) \Downarrow L''; B''; T''; F''; V''}{\Delta, (tx, DT) \Downarrow L''; B''; T''; F''; V''}$$

Correction

$$\frac{\forall dt_1 \in \Delta, DT' \geq dt_1 \quad DT < DT' \quad \Delta \Downarrow L''; B''; T''; F''; V'' \qquad \Delta \oplus (tf, DT) \Downarrow L'; B'; T'; F'; V'}{\Delta, (\text{correction } tf\ DT, DT') \Downarrow L'; B'; T'; F'; V'}$$

In ASL, the evaluation of the input stream to produce the various logs is a critical process that ensures the accuracy and consistency of financial records. This process involves interpreting each transaction in the input stream and updating the respective logs, including the True Accounting Log (L), Accounts Balance and Type Log (B), Types Log (T), Flags Log (F), and Override Log (V). The evaluation rules define how each type of transaction, whether regular or corrective, is processed to maintain a complete and accurate record of all financial activities. Here, we detail the specific rules and cases for how the input stream evaluates to produce these logs.

- **General Transaction Evaluation**

  The General rule handles the evaluation of regular transactions that are not corrections. The process involves:

  - **Initial Evaluation:** Evaluating the current state of the input stream Δ results in intermediate logs (L', B', T', F', V'). We ensure that tx is not a correction entry, and we also check for the DateTime stamp DT to be more than or equal to all the previous DateTime stamps in the Δ.
  - **Appending the Transaction:** The transaction (tx, DT) is appended to these intermediate logs, updating them to $(L'', B'', T'', F'', V'')$.

- **Final State:** The new state of the input stream, now including the transaction (tx, DT), results in the final logs $(L'', B'', T'', F'', V'')$.

This ensures that each transaction is sequentially processed and the logs are updated accordingly, maintaining a chronological order of events.

- **Correction Transaction Evaluation**

The Correction rule is designed to handle transactions that correct previous entries:

- **Initial Evaluation:** The previous input stream Δ is evaluated to produce intermediate logs $(L'', B'', T'', F'', V'')$ that the input stream would have produced in the absence of corrections. Δ is well-formed, since it evaluates to the intermediate logs.
- **Pre-Dated Correction:** The correction transaction $(\text{correction}\ tf\ DT, DT')$ is processed by evaluating the previous input stream with the correction tf inserted as if it was recorded at the original date DT. We also check for the DateTime stamp DT' to be more than or equal to all the previous DateTime stamps in the Δ. This is the date and time of the time of the user input. DT is the date and time for when the correction is to be made. We check to make sure that the correction can only be made for a past time and date (less than current), not the future.
- **Final State:** The final evaluation produces logs (L', B', T', F', V') that reflect the corrected state of the transactions.

This process ensures that corrections are applied accurately and retroactively, maintaining the integrity of the financial records.

9.3.3 Log Updates

Create Account

$$\frac{\neg(\exists c, T'. B@act_1 \rightsquigarrow (c, Ty)) \quad Ty \in T}{L; B; T; F; V; (\text{create\_acc}\ Ty\ act_1, DT) \Downarrow L; B, act_1 \mapsto (0, Ty); T; F; V}$$

Create Type

$$\frac{t_1 \notin T}{L; B; T; F; V; (\text{create\_ty}\ t_1, DT) \Downarrow L; B; T, t_1; F; V}$$

Create Flag

$$\frac{f_1 \notin F}{L; B; T; F; V; (\text{create\_flag } f_1, DT) \Downarrow L; B; T; F, f_1; V}$$

Delete Flag

$$\frac{f_1 \in F}{L; B; T; F; V; (\text{delete\_flag } f_1, DT) \Downarrow L; B; T; F - f_1; V}$$

Override Flag

$$\frac{L; B; T; F - (f_0, f_1, \ldots, f_n); V; (tf, DT) \Downarrow L'; B'; T'; F'; V'}{\begin{array}{l} L; B; T; F; V; ((tf;\ \text{override\_flag } (f_0, f_1, \ldots, f_n)\ \text{note}), DT) \Downarrow \\ L'; B'; T'; F', (f_0, f_1, \ldots, f_n); V', (tf, (f_0, f_1, \ldots, f_n), \text{note}, DT) \end{array}}$$

Accounting Transfer Entry

$$\frac{\begin{array}{r} B; (\text{transfer } act_1\ act_2\ amt, DT) \Downarrow B' \\ \forall f \in F, f(L, B, T, F, V, (\text{transfer } act_1\ act_2\ amt, DT)) = \text{True} \end{array}}{L; B; T; F; V; (\text{transfer } act_1\ act_2\ amt, DT) \Downarrow L, (\text{transfer } act_1\ act_2\ amt, DT); B'; T; F; V}$$

Current Time

$$\frac{\forall f \in F, f(L, B, T, F, V, (\text{current\_time}, DT)) = \text{True}}{L; B; T; F; V; (\text{current\_time}, DT) \Downarrow L; B; T; F; V}$$

- **Create Account**

  The Create Account rule ensures that new accounts can be added to the system only if they do not already exist. The new account starts with a balance of 0 and the specified type, ensuring it integrates correctly into the existing logs.

- **Create Type**

  The Create Type rule allows for the addition of new account types, ensuring the system can evolve and accommodate new types as needed. This is primarily for the flag functionality and is crucial for maintaining flexibility and adaptability in accounting practices.

- **Create Flag**

  The Create Flag rule introduces new flags into the system, enhancing the ability to monitor and enforce specific conditions on transactions. Flags are essential for maintaining data integrity and ensuring

transactions meet predefined criteria.

- **Delete Flag**

  The Delete Flag rule removes existing flags from the system when they are no longer needed. This keeps the flag system clean and relevant, ensuring that only necessary checks are performed on transactions.

- **Override Flag**

  The Override Flag rule is a mechanism to bypass certain flag conditions temporarily. This rule requires an accompanying transfer transaction and a note explaining the reason for the override. It ensures that exceptions are documented and justified, maintaining accountability.

  The Override Flag rule works as follows:

  - **Initial Evaluation:** The current state of the logs (L, B, T, F, V) is evaluated with the transfer transaction tf. The logs are then updated to reflect the results of the transfer.
  - **Flag Override:** The specified flags $(f_0, f_1, \ldots, f_n)$ are overridden. This involves temporarily bypassing the conditions enforced by these flags for the current transaction.
  - **Documentation:** An explanatory note is added to the override log V, documenting the reason for the override. This note ensures that the exception is justified and traceable.
  - **Final State:** The logs (L, B, T, F, V) are updated to include the results of the transfer transaction and the overridden flags, along with the explanatory note.

  Explanation:

  - **Ensuring Accountability:** By requiring a note to explain the override, the rule ensures that every exception is well-documented. This maintains transparency and accountability within the system.
  - **Conditional Bypass:** The ability to override flags allows for flexibility in handling unique or exceptional circumstances while ensuring that the core rules and conditions are generally upheld.
  - **Accompanying Transfer:** This rule can only be applied alongside a transfer transaction (tf), ensuring that the override is tied to a specific financial action, thereby maintaining the context and relevance of the override.

- **Accounting Transfer Entry**

  The Accounting Transfer Entry rule is the core transaction process in ASL. It executes transfers between accounts, updating the logs if all flag conditions are satisfied. This rule ensures that all transfers are valid and comply with the system's checks and balances.

  Steps Involved in Accounting Transfer Entry:

  1. **Initial Balance Log Update**: The balance log B is updated by processing the transfer transaction transfer $act_1\ act_2\ amt$. This results in a new balance log B', which reflects the updated balances of the involved accounts. The balance log update also ensures that cash accounts do not become negative and supports parallel processing of multiple transactions.
  2. **Flag Conditions Check**: All flags F must be evaluated against the current state of the logs (L, B, T, F, V) and the transfer transaction $(\text{transfer}\ act_1\ act_2\ amt, DT)$. The transaction is considered valid only if all flag conditions return True.
  3. **Final Logs Update**: If all flag conditions are met, the transfer transaction is executed, and the logs are updated as follows:
     - The True Accounting Log L is updated to include the new transfer transaction $(\text{transfer}\ act_1\ act_2\ amt, DT)$.
     - The balance log B is replaced with the updated balance log B'.
     - The other logs (T, F, V) remain unchanged.

- **Current Time**

  The Current Time rule is used to validate the current state of the logs against all active flags without making any changes. It serves as a checkpoint to ensure the system's integrity at any given moment.

These rules collectively ensure that ASL maintains accurate, consistent, and comprehensive logs, facilitating reliable financial record-keeping and analysis.

9.3.4 Balance Updates (for simultaneous evaluation)

Parallel Entry:

$$\frac{B; tf_1 \Downarrow B' \quad B'; tf_2 \Downarrow B''}{B; (tf_1, tf_2) \Downarrow B''}$$

Accounting Transfer Entry:

$$\frac{Ty_1 \neq \text{Cash} \quad B@act_1 \rightsquigarrow (c_1, Ty_1) \quad B@act_2 \rightsquigarrow (c_2, Ty_2) \quad c_1, c_2, amt \geq 0}{B; \text{transfer } act_1\ act_2\ amt \Downarrow B, act_1 \mapsto ((c_1 - amt), Ty_1), act_2 \mapsto ((c_2 + amt), Ty_2)}$$

Accounting Transfer Entry involving Cash:

$$\frac{Ty_1 = \text{Cash} \quad B@act_1 \rightsquigarrow (c_1, Ty_1) \quad B@act_2 \rightsquigarrow (c_2, Ty_2) \quad c_1, c_2, amt \geq 0 \quad c_1 - amt \geq 0}{B; \text{transfer } act_1\ act_2\ amt \Downarrow B, act_1 \mapsto ((c_1 - amt), Ty_1), act_2 \mapsto ((c_2 + amt), Ty_2)}$$

Parallel Entry rule allows for simultaneous evaluation of transfer transactions. If the log B can be updated by processing tf_1 to get B' and then processing tf_2 to get B'', it implies that processing the combined transfer (tf_1, tf_2) on B will result in B''.

Accounting Transfer Entry rule handles a standard transfer transaction between two accounts that do not involve cash. If the source account (act_1) and the destination account (act_2) are both found in the log B with balances c_1 and c_2 respectively, and the amount to be transferred (amt) is non-negative, the transfer is executed by deducting the amount from act_1 and adding it to act_2.

Accounting Transfer Entry involving Cash rule specifically handles transfer transactions involving cash accounts. When the source account (act_1) is a cash account, an additional constraint is applied: the cash account (act_1) must have a balance that remains non-negative after the transfer ($c_1 - amt \geq 0$). If this condition is met, the transfer proceeds by deducting the amount from act_1 and adding it to act_2.

The rules for updating the balance log B ensure that transfer transactions are processed correctly, maintaining data integrity and consistency. The **Parallel Entry** rule facilitates efficient parallel processing of transactions. The **Accounting Transfer Entry** rule handles standard transfers, ensuring non-cash accounts are updated correctly. The **Accounting Transfer Entry involving Cash** rule adds an extra check to ensure cash balances do not become negative, maintaining the accuracy of cash transactions. These rules collectively ensure that the accounting system remains robust and reliable.

9.4 Final Thoughts on ASL's Syntax and Semantics

The syntax and semantics of ASL have been designed to uphold the fundamental principles of accounting, ensuring accurate and reliable financial records. This framework allows for precise validation and correction of accounting data, making it a robust tool for accountants. ASL provides a robust framework designed to ensure the well-formedness of accounting data. While ASL is effective in modeling fundamental accounting principles and transactions, it has limitations in handling tasks outside the core accounting domain, such as Inventory Valuation and Management Reporting.

9.4.1 Limitations

ASL is currently weak in modeling tasks closely related to accounting but outside its direct scope, like Inventory Valuation (FIFO and LIFO) and Management Reporting.

- **Inventory Valuation (FIFO and LIFO):**

  However, FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) inventory valuation methods cannot be accurately modeled within ASL. Costing is a distinct discipline that requires a substantial amount of detailed information, such as the specific costs and dates of inventory purchases and sales. This level of detail is beyond what ASL is currently capable of. The complexity arises because these inventory valuation methods involve tracking individual inventory items and their costs over time, which is not within the scope of ASL's current capabilities.

- **Management Reporting:**

  Management reporting is another area where ASL shows limitations. Management reports often require aggregation, comparison, and interpretation of data across different periods and categories, which involves complex queries and data manipulation. While ASL can ensure the well-formedness of basic accounting data, generating meaningful management reports typically involves higher-level operations and insights that go beyond the transaction-level focus of ASL. This includes tasks like budget variance analysis, performance metrics, and strategic decision-making tools, which require a broader set of data and more sophisticated analytical capabilities than ASL currently provides.

- **Costing Complexity:**

Originally, there was an attempt to make ASL Object-Oriented, but the complexity of the effort expanded significantly due to the extensive additional information needed for accurate costing. Object-Oriented design would have required ASL to manage a much larger set of attributes and relationships, significantly increasing the complexity of the language and the overhead for users. For example, accurately modeling FIFO or LIFO would require ASL to track each inventory item as a distinct object with properties such as purchase date, cost, and quantity. This would complicate the language and its usage without necessarily providing proportional benefits for the core accounting functions ASL is designed to support.

There was also an attempt to make ASL Object-Oriented to handle various other accounting-adjacent tasks. However, due to the extensive additional information needed for accurate costing and other functionalities, the language was becoming too complex. Here are some examples:

- **Accounts Receivable and Payable:** Managing detailed attributes like invoice dates, amounts, and statuses for each account was too complex due to the intricate relationships and history tracking required.
- **Employee and Payroll Management:** Tracking employee details, salaries, benefits, and tax withholdings involves extensive data and relationships, adding significant overhead and complexity to the language.
- **Project Cost Tracking:** Tracking project budgets, actual costs, and resource allocations would require dynamic data management and complex validations, which complicate the language significantly.
- **Multi-Currency Transactions:** Handling fluctuating exchange rates and ensuring accurate conversion across multiple currencies involves detailed tracking and validation that adds considerable complexity.
- **Tax Management:** Modeling various tax rules, deductions, and compliance requirements would necessitate detailed, jurisdiction-specific data, making the system overly complex and less flexible.

These examples illustrate the challenges and complexities encountered when attempting to incorporate these functionalities via an Object-Oriented design. The additional data structures, relationships, and val-

idation rules required would significantly increase the language's complexity, making it cumbersome and harder to use. This led to the decision to maintain a simpler, more focused design, ensuring ASL remains manageable and effective for its core accounting purposes.

9.4.2 Challenges

The challenges in expanding ASL to cover these additional areas highlight a broader issue: the complexity and scope of accounting-related tasks are vast. Each new feature or capability introduces additional requirements for data capture, storage, processing, and validation. The goal of ASL has been to strike a balance between comprehensive validation and usability. Expanding the scope to include tasks like detailed inventory costing or complex management reporting could compromise this balance, making the language too cumbersome for its primary users. ASL can handle the accounting and ledger-keeping for all these tasks, ensuring that transactions are recorded accurately and consistently. However, it does not perform the additional computations and data management required for the extra functionalities like costing and detailed management reporting. ASL is a purely accounting-focused data formatting language, designed to ensure the integrity and well-formedness of accounting records.

Developing ASL presented several challenges, primarily around deciding the scope of the language and determining what to include and exclude. Balancing flexibility with rigorous validation while keeping the language usable for accountants required significant effort. The dynamic and context-dependent nature of transactions in accounting posed additional challenges, especially in ensuring that human errors could be corrected without compromising data integrity.

Despite these limitations, ASL remains a powerful tool for its intended purpose: ensuring the well-formedness of accounting data through robust validation mechanisms. The focus on core accounting principles allows ASL to maintain a manageable level of complexity while providing significant utility to accountants and financial professionals. Future enhancements to ASL could explore integrations with specialized systems for inventory management and reporting, leveraging ASL's strengths in data validation and integrity while addressing its current limitations in these extended areas.

9.4.3 Practical Implementation

In practice, ASL would maintain an immutable log of transactions. Users would submit new transactions, and if validated against the existing log, they would become part of the permanent record. Otherwise, they would be rejected with sufficiently detailed reasons provided.

Example of Usage:

1. **Submitting a Transaction:**
    - User submits a transaction.
    - System checks flags and validation rules.
    - If valid, the transaction is logged permanently.
    - If invalid, the system provides a reason and asks if the user wants to override the flag.
2. **Making Corrections:**
    - User submits a correction transaction.
    - The system evaluates the correction as if it occurred at the original time.
    - If valid, the correction is logged.
    - If it causes issues downstream, appropriate feedback is provided.

Future work will explore the possibility of corrections carrying transfer transactions and overrides for any resultant flags. Additionally, mechanisms to handle scenarios where multiple corrections might trigger flags and require coordinated resolution will be developed.

Overall, ASL is a powerful tool for ensuring the well-formedness of accounting data. It is less a language of computation and more a data format language with strong guarantees about its integrity. By maintaining an immutable log of transactions and providing mechanisms for validation, correction, and flag overrides, ASL offers a robust framework for accurate financial record-keeping. This approach ensures that the financial data remains reliable, transparent, and traceable, providing significant utility for accountants and financial professionals.

Chapter 10

Conclusion and Future Work

10.1 Summary of Findings

This study's main motive was to develop a Domain-Specific Language for Accounting, namely ASL, with two clear goals:

1. Reducing the gap between accounting professionals and programming capabilities
2. Providing a centralized tool to ensure standardized and uniform data integrity guarantees

We identified the limitations of existing accounting software solutions, which are often fragmented and lack standardized validity checks, leading to undetected errors and fraud. ASL was designed to standardize accounting practices, enhance the accuracy of financial reporting, and facilitate the detection of inconsistencies and fraudulent activities. By incorporating accounting-specific syntax and semantics, ASL bridges the gap between accountants and programmers, making programming more accessible to accounting professionals. The study explored the design principles, integration strategies, and potential benefits of ASL, demonstrating its potential to revolutionize accounting practices.

10.2 Contributions to the Field

The contributions of this thesis are multifaceted:

- **Development of ASL**: We introduced a Domain-Specific Language specifically tailored for accounting, providing a unified approach to implementing accounting principles programmatically.

- **Standardization**: ASL offers a standardized framework for accounting practices, reducing the fragmentation seen in current software solutions.
- **Enhanced Validity Checks**: By embedding robust validity checks into ASL, we enhance the accuracy and reliability of financial reporting.
- **Bridging the Gap**: ASL makes programming more accessible to accounting professionals, facilitating better collaboration between accountants and programmers.
- **Real-Time Detection**: The language supports real-time detection of inconsistencies and potential fraud, promoting more timely interventions.

10.3 Implications for Accounting Practice

The implementation of ASL in accounting practices has several significant implications:

- **Improved Accuracy**: Standardized syntax and semantics reduce the likelihood of errors in financial reporting.
- **Fraud Detection**: Enhanced validity checks and real-time monitoring facilitate the early detection of fraudulent activities.
- **Accessibility**: By making programming more accessible to accounting professionals, ASL encourages more accurate and consistent application of accounting principles.
- **Efficiency**: A unified approach reduces the time and effort required to develop and maintain accounting software, increasing overall efficiency.

10.4 Recommendations for Future Research

While this thesis establishes a strong foundation, several areas warrant further exploration:

10.4.1 Language for Flags

Future research should focus on developing a detailed syntax and comprehensive framework for the implementation of flags within ASL. This includes:

- **Defining a Standard Syntax**: Establish a clear and standardized syntax for writing and implementing flags. This should be intuitive and easily understandable by accounting professionals.
- **Customizability**: Allow users to create customizable flags to fit specific organizational needs and compliance requirements.
- **Efficiency**: Ensure that the flag evaluation process is efficient and does not significantly impact system performance.
- **Scalability**: Develop methods to scale the flag system for large organizations with complex accounting structures.
- **User Interface Integration**: Integrate flag definitions and management into user-friendly interfaces to facilitate ease of use by non-technical users.
- **Error Messages**: Define robust error handling to help ensure timely resolution of flag-related errors.

10.4.2 Different Flags: Overrideable and Non-Overrideable

Investigate the distinction between overrideable and non-overrideable flags, determining the appropriate contexts and mechanisms for each type. This will ensure that the system balances flexibility with the need for stringent controls. This can be achieved by:

- **Contextual Analysis**: Determine scenarios where each type of flag is appropriate, considering the risk and impact of allowing overrides.
- **Policy Integration**: Align flag definitions with organizational policies and regulatory requirements, ensuring non-overrideable flags are used for critical controls.

10.4.3 Contradictory Flags

Research methods to handle contradictory flags, ensuring that the system can effectively resolve conflicts and maintain the integrity of financial data by:

- **Conflict Resolution Algorithms**: Develop algorithms to identify and resolve conflicts between flags, prioritizing based on predefined criteria.

- **Flag Dependencies**: Establish dependency relationships between flags to understand and manage their interactions.
- **User Notifications**: Implement systems to notify users of contradictory flags, providing guidance on resolution steps.
- **Testing and Validation**: Conduct extensive testing to identify potential contradictions and validate the effectiveness of resolution strategies.

10.4.4 Inventory Valuation

Explore the integration of inventory valuation methods within ASL by:

- **Valuation Techniques**: Incorporate various inventory valuation techniques such as FIFO, LIFO, and weighted average.
- **Dynamic Valuation**: Allow for dynamic updates to inventory valuation as new transactions occur.
- **Compliance**: Ensure compliance with relevant accounting standards and regulations for inventory valuation.
- **Reporting**: Develop reporting capabilities to reflect inventory valuation changes and their impact on financial statements.

10.4.5 Accrual Basis of Accounting

Examine the implementation of accrual basis accounting in ASL by:

- **Accrual Handling**: Develop methods to accurately record and manage accrued revenues and expenses.
- **Period End Adjustments**: Implement automated period-end adjustments to reflect accrued transactions.

10.4.6 Tax Calculation

Investigate the integration of tax calculation functionalities by:

- **Tax Rules**: Incorporate comprehensive tax rules and regulations, ensuring compliance with local and international tax laws.
- **Dynamic Updates**: Develop systems to update tax rules dynamically as regulations change.
- **Accuracy and Validation**: Implement robust validation mechanisms to ensure accurate tax calculations.
- **Reporting**: Enhance reporting capabilities to include detailed tax breakdowns and compliance reports.

10.4.7 Integration with Current Accounting Systems

Research strategies for integrating ASL with existing accounting systems, ensuring a smooth transition and interoperability between different platforms. One idea worth exploring is to conceptualize ASL as a data format layer that sits between the database layer and the application layer. This positioning would maintain standardized, uniform, and centralized validity guarantees about the data across all accounting software.

Integrating ASL in this manner involves developing robust protocols for seamless data exchange between ASL and existing accounting systems. This ensures that data flows smoothly and accurately between different platforms without loss of integrity. Additionally, creating efficient data migration tools and methodologies will help organizations transition their data from legacy systems to ASL with minimal disruption, preserving the accuracy and completeness of the data.

If ASL can be effectively integrated with current accounting systems, it can help in providing a unified and reliable framework for financial data management and ensuring long-term benefits in terms of accuracy, transparency, and compliance.

10.5 Challenges and Limitations

While we have made significant advancements towards achieving our goal, we also encountered several challenges and limitations. One of the primary challenges was distinguishing between “wrong” entries due to human error and those breaking fundamental accounting rules. While human errors are common and can often be corrected through adjustment entries, entries that violate fundamental accounting principles pose a significant challenge. ASL guarantees the adherence to two fundamental accounting rules: every debit

must have a corresponding credit, and cash accounts must never be negative. However, implementing the three "Golden Rules" of accounting proved more complex due to the dynamic nature of transactions and the context-dependent classification of accounts.

The complexity arises from the necessity to dynamically assign debit or credit status based on the specific circumstances of each transaction. Static type systems were too rigid for this domain, as accounts can change roles based on context. Therefore, ASL allows for correction transactions to rectify mistakes while ensuring that the overall books of accounts remain balanced.

10.6 Conclusion

The development of ASL marks a significant step towards standardizing and enhancing accounting practices through a unified programming approach. By embedding robust validity checks and real-time auditing capabilities, ASL can effectively detect inconsistencies and potential fraud. The language's accessibility for accounting professionals bridges the gap between accountants and programmers, promoting more accurate and efficient application of accounting principles. Future research and development efforts will further refine and expand ASL, ensuring it meets the evolving needs of the accounting profession. Through continued innovation and collaboration, ASL has the potential to revolutionize accounting practices, making them more transparent, reliable, and resilient against fraud and errors.

Bibliography

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk — a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279, 1979.

[2] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone. The relational model is dead, sql is dead, and i don't feel so good myself. *ACM SIGMOD Record*, 42(1):64–68, 2013.

[3] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 249–264, 1974.

[4] J. Doshi. Technical perspective: Catala: A programming language for law. Technical report, Rochester Institute of Technology, 2022.

[5] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, pages 1–10, 2013.

[6] J. Melton and A. R. Simon. *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993.

[7] D. Merigoux, N. Chataing, and J. Protzenko. Catala: A programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021.

[8] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The sql++ unifying semi-structured query language, and an expressiveness benchmark of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[9] R. J. Schmitz, M. D. Schultz, M. A. Urich, J. R. Nery, M. Pelizzola, O. Libiger, A. Alix, R. B. McCosh, H. Chen, N. J. Schork, and J. R. Ecker. Patterns of population epigenomic diversity. *Nature*, 495(7440):193–198, 2013.

[10] C. J. Van Wyk. Literate programming: An assessment. *Communications of the ACM*, 29(7):592–595, 1986.

Appendices

.1 Survey : Programming for Accountants

Programming for Accountants Survey

Purpose:
To understand the current level of knowledge, interest, and needs of accounting professionals in programming for enhancing efficiency and productivity.

Objective:
To design a domain-specific programming language for accounting thus making programming more accessible to accountants.

Instructions:
Please take a few minutes to complete the following survey. Your responses will be kept confidential.

* Indicates required question

1. Email *

2. Name *

3. Years of Experience in Accounting *

Check all that apply.

- [ ] Less than 1 year
- [ ] 1-5 years
- [ ] 6-10 years
- [ ] 11-20 years
- [ ] More than 20 years

4. Industry *

Check all that apply.

- [ ] Finance
- [ ] Healthcare
- [ ] Retail
- [ ] Manufacturing
- [ ] Consulting
- [ ] Other: _____

Programming Experience

5. Do you have any programming experience? *

Check all that apply.

- [ ] Yes
- [ ] No

6. If yes, which programming languages are you familiar with? (Select all that apply) *

Check all that apply.

- [ ] Python
- [ ] R
- [ ] SQL
- [ ] Java
- [ ] C/C++
- [ ] None
- [ ] Other: _____

7. What software tools do you currently use in your accounting tasks?

_____

8. What challenges do you face with the current technology or programming languages in use? *

Check all that apply.

- [ ] Complexity
- [ ] Technical Jargon
- [ ] Lack of autonimity
- [ ] Lack of specific features (specify features in Other)
- [ ] Integration issues with other tools
- [ ] Security concerns
- [ ] User Errors
- [ ] Other: _____

New Domain-Specific Language Preferences

9. What features would you like to see in a new programming language designed specifically for accounting? (Select all that apply) *

Check all that apply.

- [ ] Simplified syntax
- [ ] Enhanced security features
- [ ] Integrated data analysis tools
- [ ] Automation of repetitive tasks
- [ ] Customizable to specific accounting tasks
- [ ] Other: _____

10. What types of accounting tasks or processes do you believe should be simplified or automated through this new programming language? (select all that apply) *

Check all that apply.

- [ ] Data entry
- [ ] Financial reporting
- [ ] Tax preparation
- [ ] Auditing
- [ ] Forecasting and budgeting
- [ ] Other: _____

11. How would you prefer to learn this new programming language? (Select all that apply) *

Check all that apply.

- [ ] Online tutorials
- [ ] Workshops
- [ ] Classroom training
- [ ] Webinars
- [ ] Self-taught (documentation, examples, etc.)
- [ ] Other: _____

12. What are some repetitive manual tasks that you wish could be automated? (examples can be management reports, repetitive similar data entries, please be as specific as possible) *

_____

_____

_____

_____

_____

13. What are some common errors that you wish could be eliminated? (examples can be specific errors in tools or recurring human errors, please be as specific as possible) *

14. What are some of the biggest reasons that are keeping you from learning programming?

15. Please provide any additional comments or insights that you believe would be beneficial in the development of a domain-specific programming language for accounting: