MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Pipeline-oriented scripting language for Data processing

Project report

Mentor: prof., Leon Brînzan
Students: Clepa Rodion, FAF-221
Gavriliuc Tudor, FAF-221
Pascal Adrian, FAF-221
Sungur Emre-Batuhan, FAF-221

Chișinău, 2024

Introduction

In the fast-changing world of modern computing, the demand for efficient data processing and automation tools continues to grow. Organizations across industries are constantly looking for innovative solutions to streamline workflows, speed up processes, and extract actionable insights from massive amounts of data. In the search for optimization, pipeline-based scripting languages have emerged as a powerful paradigm that offers a universal approach to organizing complex data workflows.

The purpose of this report is to provide a comprehensive analysis of pipeline-oriented scripting languages, with a focus on their use in revolutionizing data workflows. We delve into the fundamental concepts, design principles, and practical implications of these languages, exploring their implications in modern computing environments. Through detailed exploration of domain specifics, use cases, and best practices, we aim to elucidate the transformative potential of pipeline-oriented scripting languages in the fields of data science and automation.

The exponential growth of data in recent years has created both opportunities and challenges for organizations around the world. The sheer volume, velocity and variety of data generated from different sources has required the development of innovative approaches to manage, process and extract value from this wealth of information. Traditional programming paradigms often struggle to cope with the complexities of modern data workflows, leading to inefficiencies, bottlenecks, and scalability issues.

Pipeline-oriented scripting languages offer a paradigm shift in how data workflows are conceptualized, designed, and executed. Based on the concept of pipelines—a series of interconnected processes in which the output of one stage serves as input to the next—these languages provide a streamlined framework for automating tasks, coordinating data transformations, and facilitating seamless integration with existing systems and tools. By breaking complex operations into modular, building blocks, pipeline-oriented scripting languages enable users to develop flexible, scalable, and efficient data workflows tailored to their specific requirements.

Abstract

This article explores the field of pipeline-oriented scripting languages, focusing on the development and use of a Python-esque pipeline language. Pipeline-oriented scripting languages offer an approach that will optimize workflows through interconnections between processes. Benefits include optimized workflow, modularity—which refers to the ease of breaking down a system into interconnected modules, clean and simple syntax, efficient use of resources, and full compatibility. The study outlines the key requirements for building large-scale data pipelines and describes existing solutions that meet them.

The article addresses common issues in data science, automation, integration, maintainability, and scalability, and highlights the benefits of Python's pipeline language. The designed tool is used to simplify complex data processing tasks. Design considerations for a Python-esque pipeline oriented language include domain-specific abstractions, support for pipeline composition, declarative syntax, integration with an existing ecosystem. The proposal is to develop a special Python-esque pipeline language to improve data processing and analysis.

Keywords: Pipeline, data, processing, language

Content

1 Midterm 1

1.1 Domain Analysis

In the field of software engineering, domain-specific languages (DSLs) have emerged as powerful tools to address specific problems within a particular domain, enhancing productivity and reducing errors. This report details the creation of a pipeline-oriented DSL with a syntax similar to pyhton, aiming to streamline the development and management of data processing pipelines. The motivation for this project stems from the need to simplify complex processes in domains such as data analysis, machine learning, and ETL (Extract, Transform, Load) operations. Traditional methods often involve intricate configurations and significant boilerplate code, leading to increased development time and potential errors.

The pipeline-oriented DSL developed in this project seeks to provide a more intuitive and declarative approach to defining and managing data pipelines. By leveraging the strengths of DSLs, our goal is to offer a tool that not only enhances developer productivity but also ensures robust and maintainable pipeline configurations.

The primary advantage of DSLs lies in their ability to simplify complex tasks within their domain. By providing domain-specific abstractions and constructs, DSLs enable users to write programs that are more readable, maintainable, and less prone to errors.

The Case for a Dedicated Pipeline Oriented Domain Specific Language

The absence of native pipeline support in Python presents several challenges and opportunities for improvement. Firstly, the current landscape forces developers to cobble together solutions using disparate tools and libraries, leading to code that is harder to understand, maintain, and debug. Secondly, a dedicated pipeline library could provide a unified interface with built-in support for common pipeline operations such as data transformation, filtering, aggregation, and parallel execution. Such a library would simplify code and accelerate development, particularly in domains like data science and automation where pipelines are prevalent

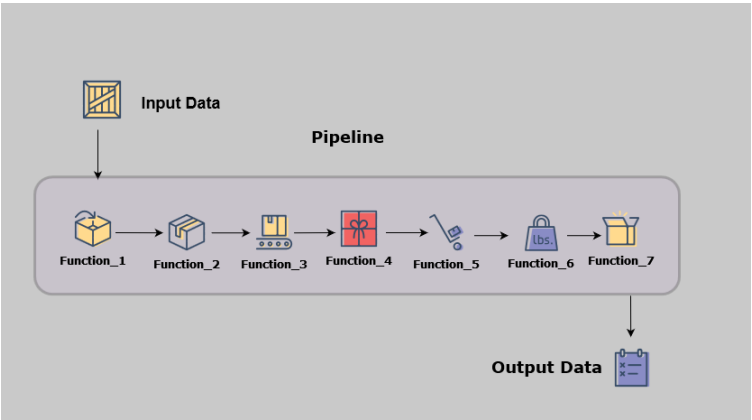

Figure 1.1.1 - Pipeline Pattern

How a Python-like DSL Benefits Machine Learning AI Workflows

In our project, the Python-like DSL demonstrates significant benefits for machine learning AI workflows. By employing a structured and standardized approach, this library enhances efficiency and productivity throughout the development life-cycle.

Firstly, the library streamlines the development process by providing a cohesive framework for defining pipelines and variables. This standardized approach ensures consistency and reproducibility across projects, reducing the risk of errors and accelerating time-to-deployment.

Furthermore, the Python-like DSL helps with automation of repetitive tasks such as data preprocessing, model training, and evaluation. By helping automating these processes, it not only saves time but also improves the reliability and robustness of machine learning models.

Additionally, the language facilitates collaboration among team members by providing a common platform and language for communication. With standardized workflows and documentation, team members can easily understand and contribute to each other's projects, fostering a culture of knowledge sharing and innovation.

Moreover, the library offers scalability and flexibility, allowing users to adapt to changing requirements and environments. Whether working with small datasets or large-scale deployments, the Python-like DSL provides the tools and resources needed to scale workflows efficiently.

Overall, the Python-like DSL is a valuable asset for machine learning AI workflows, empowering teams to focus on innovation and problem-solving. By embracing this library, organizations can unlock the full potential of their machine learning initiatives and drive impactful results.

Design Considerations for a Python-like DSL

Designing a Python-like DSL tailored for data science warrants careful deliberation:

Pipeline Blocks: Implement a syntax for defining blocks of pipeline operations using the $|>$pipe block production rule. This involves specifying individual operations within the pipeline block, along with any associated arguments.

Operation Syntax: Define a syntax for specifying individual pipeline operations using the $|>$pipe production rule. This should include the use of pipe symbol $|>$ to denote the type of operation, the function name, and any function arguments.

Whitespace Handling: Implement rules for handling whitespace within pipeline definitions and operations to ensure readability and consistency.

Error Handling: Include mechanisms for handling syntax errors and other issues that may arise during parsing to provide informative error messages to users.

Integration with Python: Design the language to seamlessly integrate with Python code, allowing users to define and execute pipelines within their Python programs.

1.2 Description of the DSL

The role of Domain-Specific Languages (DSLs) is pivotal in modern software development, offering a focused and efficient way to address the unique challenges of various domains. By design, DSLs offer a high level of abstraction tailored to a specific field, whether it be web development, data science, machine learning, or another specialized area. This targeted approach allows developers and domain experts to communicate more effectively, leveraging a language that embodies the concepts, operations, and workflows intrinsic to their domain.

DSLs can be broadly categorized into two types: external DSLs, which are standalone languages with their own syntax and compiler or interpreter, and internal DSLs, which are embedded within a host general-purpose language, leveraging its syntax while providing domain-specific functionalities. Both types aim to simplify complex tasks, but they do so in ways that best suit their intended use cases and environments.

A DSL designed with deep domain insights can inherently guide users toward adopting best practices. In the context of machine learning, this might mean integrating data validation checks directly into the language, or offering simplified abstractions for complex model evaluation metrics. By encoding such practices into the language's syntax and libraries, a DSL not only educates its users but also helps prevent common errors, thereby enhancing the overall quality and effectiveness of the work produced.

The specificity of a DSL allows it to address challenges unique to its domain with unmatched precision. For instance, in the realm of real-time systems, timing constraints, and concurrency are critical issues. A DSL tailored for this domain can provide first-class language constructs for expressing time-bound operations or concurrent execution flows, making it significantly easier for developers to build robust real-time applications. This level of specificity in tackling domain challenges is difficult, if not impossible, to achieve with general-purpose languages without extensive boilerplate code or external libraries

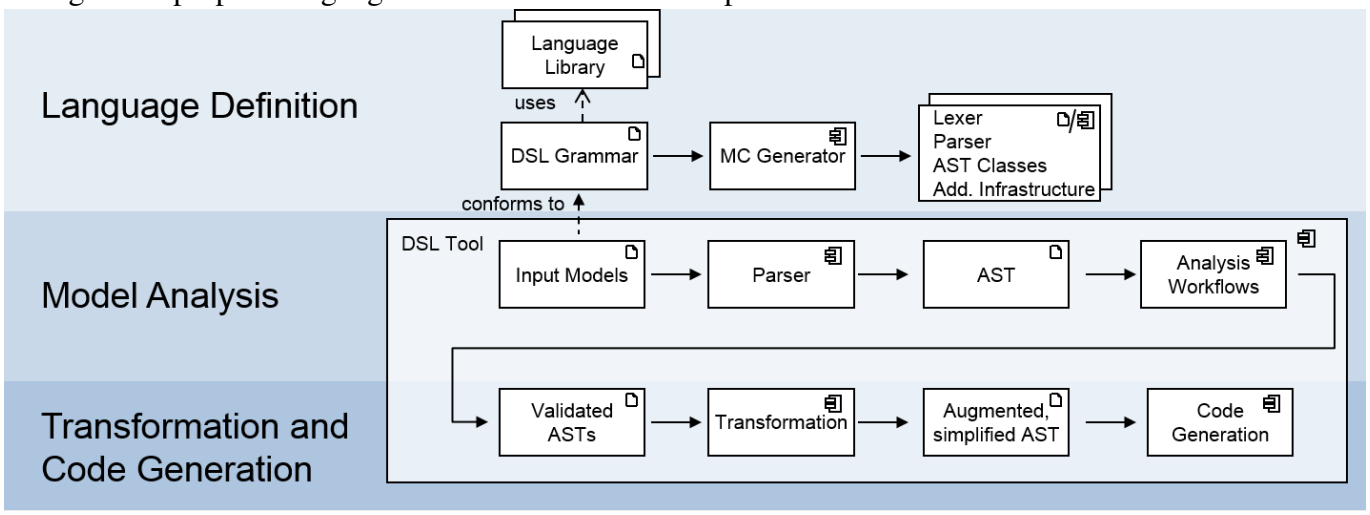

Figure 1.2.1 - Architecture of a typical DSL processing tool in a model-based software engineering process

Key Features:

Intuitive Syntax: Our DSL offers a simple and clear syntax that allows users to define complex data processing pipelines with minimal effort.

Benefits:

- Ease of Use: Users can quickly write and understand pipeline definitions without extensive training.
- Readability: Code written in the DSL is easy to read and maintain, enhancing collaboration and reducing the likelihood of errors.

Example:

```
data := [3, 8, 3]
data |> process1() |> process2()
```

Modularity and Reusability: DSL promotes modularity by breaking down complex tasks into smaller, reusable components. Users can encapsulate common processing logic into standalone stages, making it easier to maintain and reuse code across different pipelines and projects.

```
data |> clean_data() |> transform_data()
```

Streamlined Workflow: By organizing tasks into pipelines, DSL streamlines the workflow of data processing and automation tasks. Pipelines enable sequential execution of processing stages, eliminating the need for manual intervention between each step and accelerating the execution of tasks.

Error Handling and Validation: The DSL includes built-in mechanisms for syntax checking and error handling, ensuring robust and reliable pipelines.

Benefits:

- Reliability: Reduces the risk of runtime errors by catching issues early.
- User Feedback: Provides immediate feedback on syntax errors, helping users correct mistakes quickly.

Example:

```
if data is invalid
    raise Error("Invalid data format")
```

Scalability: The DSL is designed to handle large-scale data processing tasks efficiently, making it suitable for both small and enterprise-level applications.

Benefits:

- Performance: Optimized for performance to handle large datasets and complex transformations.
- Scalability: Easily scales to accommodate growing data processing needs without significant changes to the pipeline definitions.

Example:

```
big_data := [/* large dataset */]
big_data |> preprocess() |> analyze() |> report()
```

Extensibility: The DSL allows for custom extensions and user-defined functions, providing flexibility to adapt to specific requirements.

Benefits:

- Customization: Users can extend the DSL's capabilities to meet their unique needs.
- Adaptability: Easily adapts to various domains and applications by incorporating custom logic and functionality.

Interoperability and Integration: DSL seamlessly integrates with existing tools, libraries, and systems, enabling interoperability and data exchange. Users can leverage built-in mechanisms for interfacing with external programs, APIs, and data sources, facilitating seamless integration with third-party services and platforms. Our DSL is designed to integrate seamlessly with Python, allowing users to leverage Python's extensive libraries and ecosystem.

Benefits:

- Extended Functionality: Users can enhance their pipelines with additional functionality from Python libraries.
- Flexibility: Combines the simplicity of the DSL with the power of Python, offering a versatile solution for data processing.

Potential users

The potential users of our pipeline-oriented DSL span a wide range of professions and industries, each with unique needs and challenges. Data scientists, data engineers, software developers, business analysts, researchers, operations teams, and educational institutions can all benefit from the streamlined, efficient, and user-friendly approach to defining and managing data pipelines that our DSL offers. By addressing the specific requirements of these diverse user groups, our DSL stands as a powerful tool for enhancing productivity, collaboration, and innovation in data processing tasks.

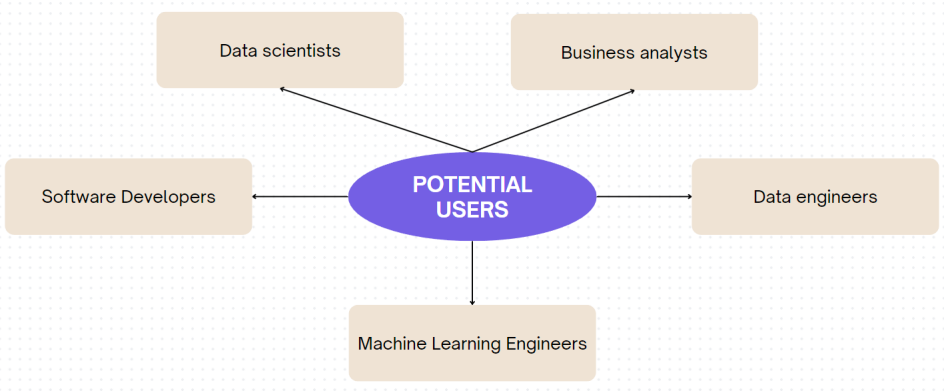

Figure 1.2.2 - Potential Users

1. Data scientists:

Data scientists will profit from the initiative by using the library to optimize their workflow, automate tedious operations, and increase the reproducibility of their results. By offering a standardized framework

for developing and running data pipelines, the project allows data scientists to focus on data analysis and interpretation rather than pipeline creation mechanics.

2. Machine Learning Engineers: Machine learning engineers can use the library to build and deploy large-scale machine learning models. By incorporating the library into their workflow, businesses can automate model training, evaluation, and deployment, minimizing manual labor and increasing productivity.

3. Data engineers:

Data engineers can use the library to create reliable and scalable data pipelines for ingesting, analyzing, and converting enormous amounts of data. The library's capabilities for parallel processing and distributed computing makes it ideal for managing large data sets. The initiative will help data engineers by making it easier to create and maintain data pipelines, lowering time-to-delivery, and enhancing overall system reliability.

4. Software Developers:

Software developers can use the library to automate data-related operations and workflows in their applications. By offering a Python-native interface for designing and executing pipelines, the library allows developers to quickly include data processing capabilities into their applications. This allows developers to focus on developing core application logic while utilizing the library's capability for data manipulation and analysis.

5. Business analysts:

Business analysts can utilize the library to automate data preparation and analysis operations, allowing them to generate insights faster and more correctly. The library's support for data transformation and visualization enables exploratory data analysis and reporting, allowing analysts to confidently make data-driven decisions. By optimizing the data pipeline from raw data to actionable insights, the project enables business analysts to discover useful insights and drive corporate growth.

1.3 Grammar

ANTLR, short for ANother Tool for Language Recognition, is a robust parser generator used to construct parsers, interpreters, compilers, and translators for various programming languages and domain-specific languages. It operates by taking a formal grammar of the language as input and producing a parser for that language in target languages such as Java, C, Python, and others. Its advantages include its language-agnostic nature, support for LL parsing allowing for more expressive grammars, automatic generation of Abstract Syntax Trees for parsed input, detailed error reporting facilitating easier debugging, seamless integration with IDEs and development tools, and a large and active community providing support and resources for users.

Lexical Considerations

All Decaf keywords are lowercase. Keywords and identifiers are case-sensitive. For example, 'if' is a keyword, but 'IF' is a variable name; 'name' and 'Name' are two different names referring to two distinct variables.

Comments are started by '#' and are terminated by the end of the line.

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, page and line-breaking characters, and comments.

Keywords and identifiers must be separated by white space or a token that is neither a keyword nor an identifier. For example, 'thisfortrue' is a single identifier, not three distinct keywords.

If a sequence begins with an alphabetic character or an underscore, then it, and the longest sequence of characters following it forms a token.

String literals are composed of <char>s enclosed in double quotes.

A character literal consists of a <char>enclosed in single quotes.

Numbers in Decaf are 32-bit signed. That is, decimal values between -2147483648 and 2147483647.

If a sequence begins with '0x', then these first two characters and the longest sequence of characters drawn from [0-9a-fA-F] form a hexadecimal integer literal.

If a sequence begins with a decimal digit (but not '0x'), then the longest prefix of decimal digits forms a decimal integer literal.

A 'char' is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) other than quote ("), single quote ('), or backslash (\), plus the 2-character sequences '\"' to denote quote, '\' to denote single quote, '\\' to denote backslash, '\t' to denote a literal tab, or '\n' to denote newline.

Reference Grammar

Notation	Meaning
`<foo>`	(in bold font) foo is a non-terminal
foo	foo is a terminal
`[x]`	zero or one occurrence of x
`x*`	zero or more occurrences of x
x^+	a comma-separated list of one or more x's
{}	large braces for grouping
|	separates alternatives

Table 1.3.1 - Context-Free Grammar Notations

```
[language=ANTLR]
grammar Expr;
```

```
prog: (variable_assignment | expression)* EOF ;
expression: variable_access single_pipe_symbol function_call (single_pipe_symbol functio
variable_assignment: VAR_NAME '=' (variable_value | STRING) ;
variable_value: array_value | STRING ;
array_value: '[' (number (',' number)*)? ']' ;
number: FLOAT | INT | '-' INT ;
variable_access: VAR_NAME ('[' index=INT ']')? ;
function_call: VAR_NAME left_par args right_par | VAR_NAME left_par right_par ;
arg: number | STRING | array_value ;
args: arg (',' arg)* ;
single_pipe_symbol: '|>';
VAR_NAME: [a-zA-Z][a-zA-Z0-9]* ;
FLOAT   : [0-9]+[.][0-9]+ ;
left_par: '(' ;
right_par: ')' ;
STRING: '"' ~["]* '"' ;
INT: [0-9]+ ;
WS: [ \t\r\n]+ -> skip ;
```

Parsing example

```
[language=Python]
    var = [3, 8, 3]
    var |> function1() |> function2()
```

The given Python code snippet illustrates the application of function chaining using the —¿ operator, also known as the pipe operator, to pass the result of one function as an argument to another function sequentially.

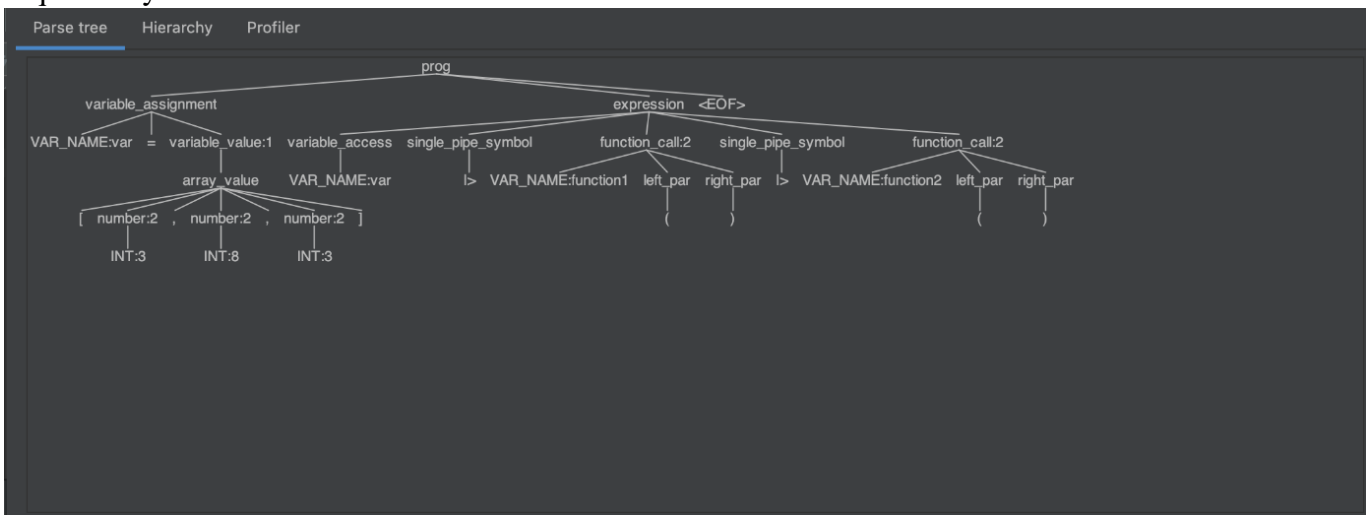

Figure 1.3.1 - Parse tree

ANTLR Grammar definiton

ANTLR Grammar definiton: Parsing example Some code + tree screen

Semantics

The pipe operation is a mechanism for sequentially applying a series of functions to a given value. In the provided example, the pipe function facilitates this operation. The functions to be applied are specified as arguments to the pipe function, and they are executed in the order they are provided. The result of each function becomes the input for the next one, creating a sequential transformation of the original value.

Scope Rules

Declaration before Use: All identifiers, such as variables and functions, must be declared before they are used.

Method Invocation: A method can only be invoked or called by code that appears after its header. This rule ensures that methods are defined and their functionality is known before any attempts to use them in the program.

No Redefinition in the Same Scope: No identifier may be defined more than once in the same scope. This rule ensures that there is clarity and uniqueness in the naming of fields, methods, local variables, and formal parameters.

Locations

In terms of data types, can include both scalar and array types. Scalar locations might hold individual quotes or author names, typically represented as strings. On the other hand, array locations could accommodate collections of quotes or authors, potentially organized by theme, category, or other criteria. This flexibility allows for the effective management of diverse data structures, enhancing the capabilities of the DSL.

1.4 Lexer and Parser

Parser:

Grammar Structure: The grammar consists of various rules, each representing a part of the syntax of the custom language.

Token Definitions: It defines several literal and symbolic tokens like '=', '[', ']', '-', '—¿', '(', ')', 'VAR_NAME', 'FLOAT', 'STRING', 'INT', and 'WS' (whitespace).

Pipeline_flagContext class: This class represents a context for pipeline flags. It inherits from ParserRuleContext as well. However, in the provided code, it seems empty. There's no actual parsing logic defined inside it.

Rules

prog: This rule represents the entire program. It consists of a series of variable_assignmentand èxpression.

variable_assignment: This rule defines how variables are assigned values.

Expression: This rule represents an expression in the language. It includes `variable_access`, `single_pipe_sy` and `function_call`.

variable_value: This rule represents the value of a variable, which could be an array or a string.

array_value: This rule defines the syntax for defining arrays.

number: This rule defines how numbers are represented in the language, either floats or integers.

variable_access: This rule represents how variables are accessed, optionally with an index.

function_call: This rule represents a function call, which includes the function name and its arguments.

arg: This rule represents an argument passed to a function.

args: This rule represents the list of arguments passed to a function.

single_pipe_symbol: This rule represents the pipe symbol used in expressions.

left_par and right_par: These rules represent left and right parentheses used in expressions.

Lexer:

Tokens: Tokens represent the smallest units of language syntax recognized by the lexer. Each token is assigned a numeric ID and may have a literal name and symbolic name.

Literal Names: These are literal symbols in the language, such as keywords and punctuation marks. 'pipe', ' |> ', '(', ')', ',', '[', ']', '='

Symbolic Names: These are abstract names assigned to token types. INT, STRING, FLOAT, VAR_NAME

Rule Names: These define the lexer rules for recognizing patterns in the input text. Each rule corresponds to a regular expression or set of conditions for identifying tokens. Rules like STRING, FLOAT, VAR_NAME define patterns for matching specific types of tokens.

Channel Names: ANTLR allows tokens to be assigned to different channels for processing. This can be useful for separating tokens used for different purposes, such as separating comments or whitespace from significant language constructs.

Mode Names: Lexer modes allow for different sets of rules to be applied based on the lexer's current state. However, this lexer only has the default mode.

Constructor: The __init__ method initializes the lexer, setting up its input and output streams, checking the ANTLR version, and initializing the lexer's internal components.

Midterm 2

Implementation

Listener - is most important part of the implementation it implements custom logic to traverse the AST and execute operations specified in the input script.

Pipeline Execution - Executes the specified pipeline commands on the provided data structures. This part also check if function respect parameters requirements for function and which functions can work with specific data types.

Implementation Description: **MyListener** code defines a custom listener class named MyListener, which extends the ParseTreeListener class provided by ANTLR.

```
class MyListener(ParseTreeListener):
    def __init__(self):
        self.listPipe = {}

    def enterFunction_call(self, ctx):
        function_name = ctx.VAR_NAME().getText()

        varName = ctx.parentCtx.getText()
        varName = varName[0:varName.find("|")]

        args_ctx = ctx.args()
        args = []
        if args_ctx is not None:
            for arg_ctx in args_ctx.arg():
                arg_text = arg_ctx.getText()

                token_re = arg_ctx.getToken(ExprParser.STRING, 0)
                if (token_re is not None):
```

Figure 1.4.1 - Grammar Listener

__init__: Initializes an empty dictionary listPipe to store information about pipelines. After the all data is collected the functions will be executed. Class myPipe will be explain below.

enterFunction_call: This method is triggered when the parser enters a function call node in the parse tree. It extracts the function name and the variable name from the context. It parses the arguments passed to the function call, converting string arguments to their appropriate types (int or float). It then adds the function and its arguments to the corresponding pipeline in the listPipe dictionary.

enterArray_value: This method is called when the parser enters an array value node in the parse tree. It extracts the elements of the array and their types. It determines the variable name associated with the array. It creates a new pipeline object (myPipe) with the array data and adds it to the listPipe dictionary.

enterVariable_assignment method: This method is invoked when the parser enters a variable assignment node in the parse tree. It retrieves the value assigned to the variable and determines its type (string or array). It then creates a new pipeline object (myPipe) with the variable data and adds it to the listPipe dictionary.

```
from antlr4.error.ErrorListener import ConsoleErrorListener
class MyErrorListener(ConsoleErrorListener):
    def __init__(self):
        super().__init__()
        self.has_errors = False

    def syntaxError(self, recognizer, offendingSymbol, line, column, msg, e):
        self.has_errors = True
        print("Syntax error at line {0}:{1} - {2}".format(line, column, msg))
```

Figure 1.4.2 - Error Listener

The code from Figure 1.4.2 defines a custom error listener class named MyErrorListener, which inherits from ConsoleErrorListener provided by ANTLR.

__init__: Initializes the ‘has_errors‘ attribute to ‘False‘ to track whether any syntax errors occur during parsing.

syntaxError: This method is invoked when a syntax error is encountered during parsing. It sets the ‘has_errors‘ flag to True to indicate that an error has occurred. It prints a message indicating the location (line and column) and the nature of the syntax error.

```
class myPipe():
    def __init__(self, varName, var, typeVar):
        self.debugWork = False
        self.varName = varName
        self.var = var
        self.type = typeVar
        self.pipeFunctions = []
        self.function_map = {
            "someFunctionName1": self.someFunctionName1,
            "someFunctionName2": self.someFunctionName2,
            "removeAllZero": self.removeAllZero,
            "filterPositive": self.filterPositive,
            "filterNegative": self.filterNegative,
            "filterEven": self.filterEven,
            "filterOdd": self.filterOdd,
            "mapSquare": self.mapSquare,
            "mapDouble": self.mapDouble,
            "mapToString": self.mapToString,
            "sum": self.sum,
```

Figure 1.4.3 - Pipe object

The code from Figure 1.4.3 defines pipeline of functions to be executed on some data.

__init__: Initializes several attributes. ‘debugWork‘: A boolean flag indicating whether debug messages should be printed. ‘varName‘: Name of the variable associated with this pipeline. ‘var‘: The variable or data on which the pipeline operations will be performed. ‘type‘: Type of the variable (array or string). ‘pipeFunctions‘: A list to store the functions to be executed in the pipeline. ‘function_map‘: A dictionary mapping function names to their corresponding methods within the class. This serves as a lookup table for dynamically selecting functions based on their names.

addPipeFunctions method: Appends a dictionary representing a function call to the ‘pipeFunctions list‘. Each dictionary contains two key-value pairs: ‘functionName‘: The name of the function to be called. ‘params‘: Parameters to be passed to the function when called.

```python
def executePipes(self):
    for pipe in self.pipeFunctions:
        funName = pipe["functionName"]
        params = pipe["params"]

        # Check if the function exists in the function map
        if funName not in self.function_map:
            raise Exception("No such function: " + funName)

        # Retrieve the function from the function map
        function = self.function_map[funName]

        # Check if the number of parameters matches the expected number
        expected_params = function.__code__.co_argcount - 1  # Subtract 1 for self
        if len(params) != expected_params:
            raise Exception(f"Function {funName} requires {expected_params} parameters")

        # Call the function with parameters
        if params:
            function(*params)
        else:
            function()

    print(self.var)
```

Figure 1.4.4 - Pipes execution

executePipes method is responsible for executing the pipeline of functions stored in the ‘pipeFunctions‘ attribute of the ‘myPipe‘ class.

Iterating through pipeFunctions: It iterates over each dictionary (representing a function call) in the ‘pipeFunctions list‘. For each function call, it extracts the function name (funName) and its parameters (params). It checks if the extracted function name exists in the function_map attribute of the class. If not, it raises an exception indicating that the function does not exist.

It also validates the number of parameters required by the function. If the number of parameters provided does not match the expected number, it raises an exception indicating a parameter mismatch.

Once validated, it retrieves the function reference from the function_map. If the function requires parameters (params is not empty), it calls the function with the provided parameters using the params syntax, which unpacks the parameters from the list. If the function does not require parameters, it calls the function without any parameters.

```
x = "    MYSTriNG   "
x = [1, 0, 3, 0, -2, -3, 44.5, 5]
x |> allAbs()|> mapSquare() |> increment(333) |> decrement(-33) |> multiply(3) |> division(2)

y = "    MYSTriNG   "
y |> trim()|> replaceAll("i", "o")
```

Figure 1.4.5 - Example Usage

In example it demonstrated how variable redefinition works. Elements from variable ‘x‘ will be transformed in positive after rise to power and basic math operations. For ‘y‘ will be removed whitespaces and replace letter ’i’ with ’o’.

```
[550.5, 549.0, 562.5, 549.0, 555.0, 562.5, 3519.375, 586.5]
MYSTroNG
```

Figure 1.4.6 - Example Result

Conclusions

In this report, we have explored the development and implementation of a Domain-Specific Language (DSL) tailored for pipeline-oriented workflows in Python. DSLs play an integral role in modern software development by providing specialized languages that address the unique needs and challenges of specific domains. Through our DSL, we aim to simplify and streamline the creation, manipulation, and management of data pipelines, thereby enhancing productivity and reducing the potential for errors.

The proposed Python pipeline language offers a clean and intuitive syntax, facilitating the construction of intricate data processing pipelines. The key benefit is its ability to streamline complex data workflows by organizing tasks into interconnected pipelines. By breaking operations down into modular building blocks, these languages enable users to develop flexible, scalable, and maintainable solutions tailored to their specific requirements. It is just one of many instruments which is developed to simplify complex tasks. It makes complex tasks more simple for a wider range of users.

Through examples, we have demonstrated how our DSL can be utilized to define and manage pipelines effectively. The inclusion of ANTLR for parsing the DSL showcases the technical foundation supporting its syntax and grammar. Additionally, the parser tree provides a visual representation of how the DSL interprets and processes the defined pipelines, offering insights into its internal workings.

However, it is important to recognize that implementing pipeline-oriented scripting languages is not without its challenges. Users may encounter issues related to performance optimization, resource utilization. Moreover, as data volumes continue to grow and new technologies emerge, pipeline-oriented scripting languages must evolve to meet the changing needs and requirements of the industry.

In conclusion, the development of a pipeline-oriented DSL in Python represents a significant step towards addressing the specialized needs of data pipeline workflows. By offering a high level of abstraction, improved readability, and integrated best practices, our DSL empowers users to create, manage, and optimize data pipelines with greater efficiency and confidence. This project underscores the importance of DSLs in modern software development, highlighting their potential to transform how domain-specific challenges are approached and solved.

Bibliography

[1] What is a data pipeline — IBM https://www.ibm.com/topics/data-pipeline#:~:text=A%20data%20pipeline%20is%20a,usually%20undergoes%20some%20data%20processing.

[2] Hapke, H., & Nelson, C. (2020). Building machine learning pipelines. O'Reilly Media.

https://books.google.ie/books?id=H6_wDwAAQBAJ&printsec=frontcover&dq=O%27REILLY+Building+Machine+Learning+Pipelines+Automating+Model+Life+Cycles+with+TensorFlow+Hannes+Hapke+%26+Catherine+Nelson+Foreword+By+Aur%C3%A9lien+G%C3%A9ron&hl=&cd=1&source=gbs_api#v=onepage&q=O'REILLY%20Building%20Machine%20Learning%20Pipelines%20Automating%20Model%20Life%20Cycles%20with%20TensorFlow%20Hannes%20Hapke%20%26%20Catherine%20Nelson%20Foreword%20By%20Aur%C3%A9lien%20G%C3%A9ron&f=false

[3] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 2639–2652. https://doi.org/10.1145/3448016.3457566

[4] Leo Goodstadt, Ruffus: a lightweight Python library for computational pipelines, Bioinformatics, Volume 26, Issue 21, November 2010, Pages 2778–2779, https://doi.org/10.1093/bioinformatics/btq524

[5] K. Hölldobler, J. Michael, J. O. Ringert, B. Rumpe, A. Wortmann: Innovations in Model-based Software and Systems Engineering. In: Journal of Object Technology (JOT), A. Pierantonio, M. van den Brand, B. Combemale (Eds.), Volume 18(1), pp. 1-60, AITO - Association Internationale pour les Technologies Objets, Jul. 2019. https://www.se-rwth.de/publications/Innovations-in-Model-based-Software-And-Systems-Engineering.pdf

[6] Tomaž Kosar, Sudev Bohra, Marjan Mernik, Domain-Specific Languages: A Systematic Mapping Study, Information and Software Technology, Volume 71, 2016, Pages 77-91, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2015.11.001.

[7] Github of the team https://github.com/RodionClepa/PBL_TEAM5_Pipe