PIPELINE – ORIENTED SCRIPTING LANGUAGE FOR DATA PROCESSING

Adrian PASCAL*, Emre-Batuhan SUNGUR, Rodion CLEPA, Tudor GAVRILIUC

Department of Software Engineering and Automation, group FAF-221, Faculty of Computers, Informatics, and Microelectronics, Technical University of Moldova, Chisinau, Republic of Moldova (10pt., Italic, centered)

*Corresponding author: Adrian Pascal, adrian.pascal@isa.utm.md

Tutor/coordinator: **Leon BRÎNZAN**, university assistant, Technical University of Moldova

Abstract. *This article explores the field of pipeline-oriented scripting languages, focusing on the development and use of a Python-based 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 pipeline language include domain-specific abstractions, support for pipeline composition, declarative syntax, integration with an existing ecosystem. The proposal is to develop a special Python pipeline language to improve data processing and analysis.*

Keywords: *ANTLR 4, automation, machine learning, Python programming language.*

Introduction

In the rapidly changing programming industry, the need 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 useful insights from massive amounts of data. In the search for optimization, pipeline implementations have emerged as a powerful paradigm that offers a universal approach to organizing complex data workflows. The goal of this article is to provide a comprehensive analysis of pipeline scenarios. The growth of data in recent years has created both opportunities and challenges for organizations around the world. Huge volume, speed and variety of data generated from different sources has required the development of innovative approaches to manage, process and extract value from a variety of information.

A pipeline-oriented scripting language will offer new methods in data workflows conceptualization, design, and execution. Based on the concept of pipelines, these languages will provide a streamlined framework for automating tasks and data transformations. By breaking complex operations into modular building blocks, pipeline-oriented scripting languages enable users to develop flexible, scalable, and efficient data [1].

Domain Analysis

As was said earlier pipeline-oriented scripting languages are important tools for modern data processing, as they successfully streamline complex operations. These languages excel at defining and composing data processing pipelines, breaking down tasks into modular phases to improve flexibility and scalability.

The technique of pipelining is a valuable tool for streamlining program engineering, particularly in the realm of extensive software development. This involves the process of dividing a complex problem into smaller, more manageable stages that can be tackled by multiple design teams

simultaneously, resulting in more efficient and faster development. By designing interfaces between these stages, development teams can work independently with minimal cooperation, ultimately leading to higher-quality software while also reducing development costs by simplifying the problem and allowing developers to specialize in specific areas. For large-scale software projects, the pipeline architecture is the perfect approach to take [2].

Description of Domain Specific Language

A pipeline-oriented scripting language is a programming language which groups functions into interconnected operations to make it easier to create automated processes. The foundation of Domain Specific Language is the idea of pipelines, where data passes through several processing steps, each of which modifies the incoming data in a particular way. To build these pipelines and configure complex data processing operations, the language provides a short and clear syntax for defining these pipelines and organizing complex data processing tasks.

One of the key features of the Pipeline-oriented scripting language is its simplicity. The has an easy-to-read and clean syntax that is designed to be intuitive for users. The language offers a wide set of data formats:

- integers;
- floating-point numbers;
- strings;
- arrays.

Pipeline Composition allows users to create pipelines by combining individual operations or functions. Each stage represents a discrete operation or transformation applied to the input data.

Modularity and reusability divide the problem into smaller groups of reusable components. Encapsulated operations users can transform into individual stages, making it easier to reuse code across projects.

Grammar

For Domain Specific Language, it is crafted a grammar table to provide a clear understanding of its syntax and structure.

Table 1

Reference Grammar

Notation	Meaning
<foo>	foo is a non-terminal
foo	foo is 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

To investigate a programming language mathematically a mechanism is required to characterize it. English is insufficient due to the informal description, imprecision and ambiguity, because of it a grammar, with a set of notation is introduced [3].

Notation:

```
<prog> := <pipeline def> <var def> <pipe block> EOF;
<pipeline flag> := 'pipe';
<arg> := INT | CHAR | STRING | FLOAT | VAR NAME;
20
<args> := <arg> ( <comma> <space> * <arg> )*;
<function args> := <left par> <args> <right par> ;
<function name> := VAR NAME;
<pipeline def> := <pipeline flag> <space> <function name> <left par> <right par> <two points>
```

```
      | <pipeline flag> <space> <function name> <function args> <two
points> ;
      <single pipe symbol> := '|>' ;
      <double pipe symbol> := '||>' ;
      <triple pipe symbol> := '|||>' ;
      <pipe> := <tab> <single pipe symbol> <space> <function name> <left
par> <right par>
      | <tab> <double pipe symbol> <space> <function name> <left par>
<right par>
      | <tab> <tripple pipe symbol> <space> <function name> <left par>
<right par>
      | <tab> <single pipe symbol> <space> <function name> <function
args>
      | <tab> <double pipe symbol> <space> <function name> <function
args>
      | <tab> <tripple pipe symbol> <space> <function name> <function
args> ;
      <pipe block> := <pipe> ( <pipe> )*;
      <tab> := ' ';
      <space> := ' ';
      <left par> := '(';
      <right par> := ')';
      <comma> := ',';
      <two points> := ':';
      <var name> := VAR NAME | CHAR;
      <var def> := <tab> <var name> ;
      NEWLINE : [ \r \n]+ -> skip;
      INT : [0-9]+ ;
      CHAR : [a-zA-Z] ;
      STRING : ['][a-zA-Z]+['] ;
      FLOAT : [0-9]+[.][0-9]+ ;
      VAR NAME : [a-zA-Z][a-zA-Z0-9]* ;
      COMMENT : '#' ( ' \r' | ' \n' )* -> skip;
```

Parsing Example

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.

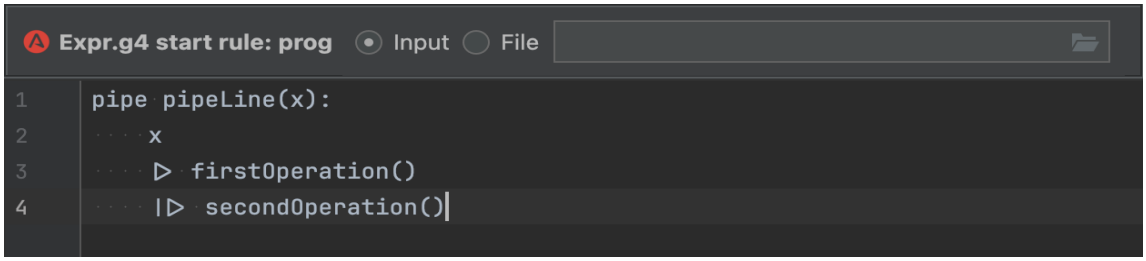

Figure 1. Code example

The code in this image will be parsed by ANTLR built parser, which was built based on the grammar we created. It shows a valid code block in our DSL, featuring pipelines. The pipeline

is declared with a variable as an argument. Each operation is added on a new line with proper tabulation. Pipeline operations, symbolized, declare and chain operations with compatible variable types.

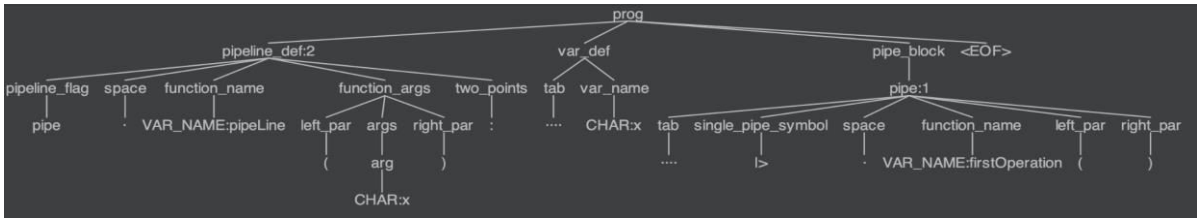

Figure 2. Parsing tree example

In this parsing example, it is demonstrated how a pipeline is parsed. Components of this script:

1. Pipe Declaration: The script begins with the declaration of a pipe function named pipeLine. This pipe takes a single parameter x.
2. Pipeline Operator (|>): This operator is used to interconnect operations within the pipeline, where the output of one operation serves as the input to the next.
3. Operation: `firstOperation()`, is applied to the input parameter x. This operation represents a transformation or action that is performed on the input data.

Conclusions

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.

References

[1] R. Molowny-Horas, “Intro R piping” [Online]. Available https://emf.creaf.cat/training/r_basics/day_2_03_pipes.pdf

[2] C. Bienia, Li K. “Characteristics of Workloads Using the Pipeline Programming Model” [Online]. Available https://doi.org/10.1007/978-3-642-24322-6_14

[3] P. Linz, “An introduction to formal languages and automata” [Online]. Available https://fall14cs.wordpress.com/wp-content/uploads/2017/04/an-introduction-to-formal-languages-and-automata-5th-edition-2011.pdf