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 powerful tools 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. We aim to clarify the transformative potential of pipeline-oriented scripting languages in data science and automation by in-depth examination of domain specifics, use cases, and best practices.

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-like 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.

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-like 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-like 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 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 different 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.

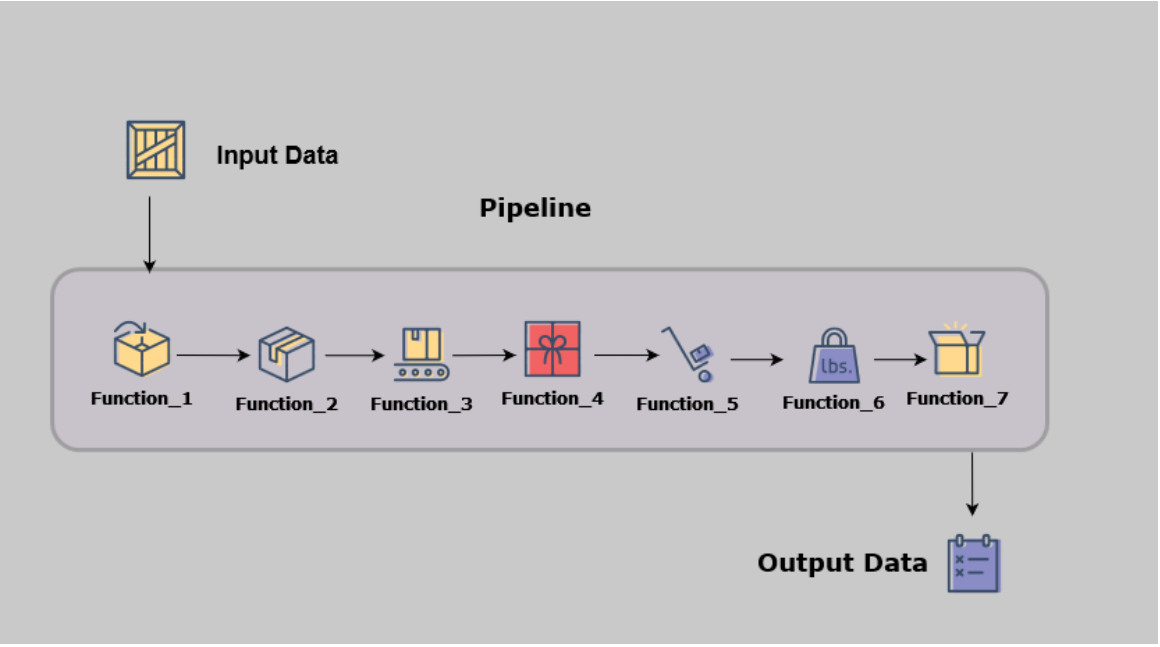

Figure 1.1.1 - Pipeline Pattern

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.

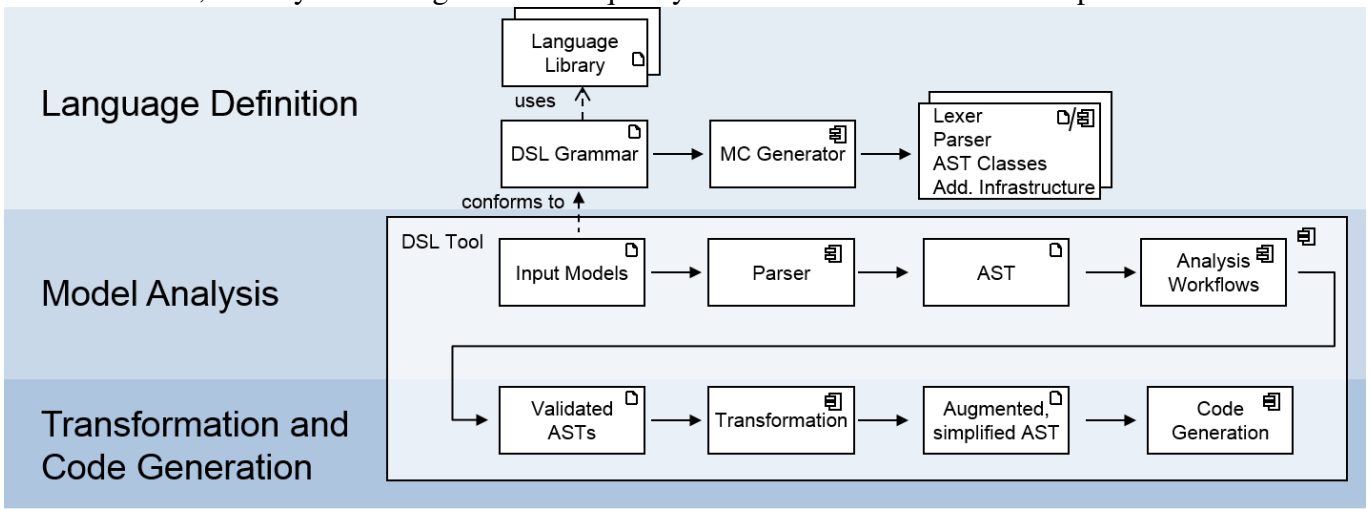

Figure 1.2.1 - Architecture of a typical DSL processing tool in a model-based software engineering process

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 re-

ducing 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.

Interoperability and Integration: DSL seamlessly integrates with existing tools, libraries, and systems, enabling interoperability and data exchange. Users can leverage built-in mechanisms for inter-

facing 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 and researchers 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.

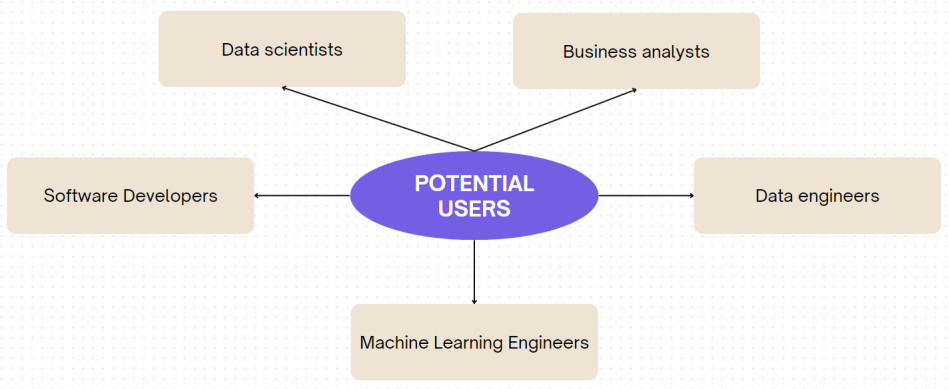

Figure 1.2.2 - Potential Users

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, analyz-

ing, and converting enormous amounts of data. 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

```
grammar Expr;

prog: imports (assignment single_pipe_statement*)+;

imports: import_statement*;

import_statement: 'from' NAME 'import' NAME;

single_pipe_statement: NAME (SPIPE function_call)+;

function_call: NAME '(' args ')' | NAME '(' ')';
```

```
assignment: NAME '=' (value | array);

array: '[' ']' | '[' (value | array) (',' (value | array))* ']';

value: INT | FLOAT | STRING | BOOL | CHAR | NAME;

args: (value | array) (',' (value | array))*;

INT: [0-9]+ | [-][0-9]+;
FLOAT: [0-9]+[.][0-9]+ | [-][0-9]+[.][0-9]+;
CHAR: ["][a-zA-Z0-9]["];
STRING: ["]~["]*["];
WS: [ \t\r\n]+ -> skip;
NAME: [a-zA-Z][a-zA-Z0-9_]*;
BOOL: 'true' | 'false';
SPIPE: [|][>];
```

Parsing example

```
from import_test_1 import split_period


x = "    MYSTriNG   "
x = [1, 0, 3, 0, -2, -3, 44.5, 5]
x |> abs() |> squareAll() |> increment(333) |> decrement(-33) |> multiply(3)
|> division(2) |> arr2Str() |> replaceAll("5", "a") |> split_period()


g = [1, 2, 3, 4]
g |> findMax()


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

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.

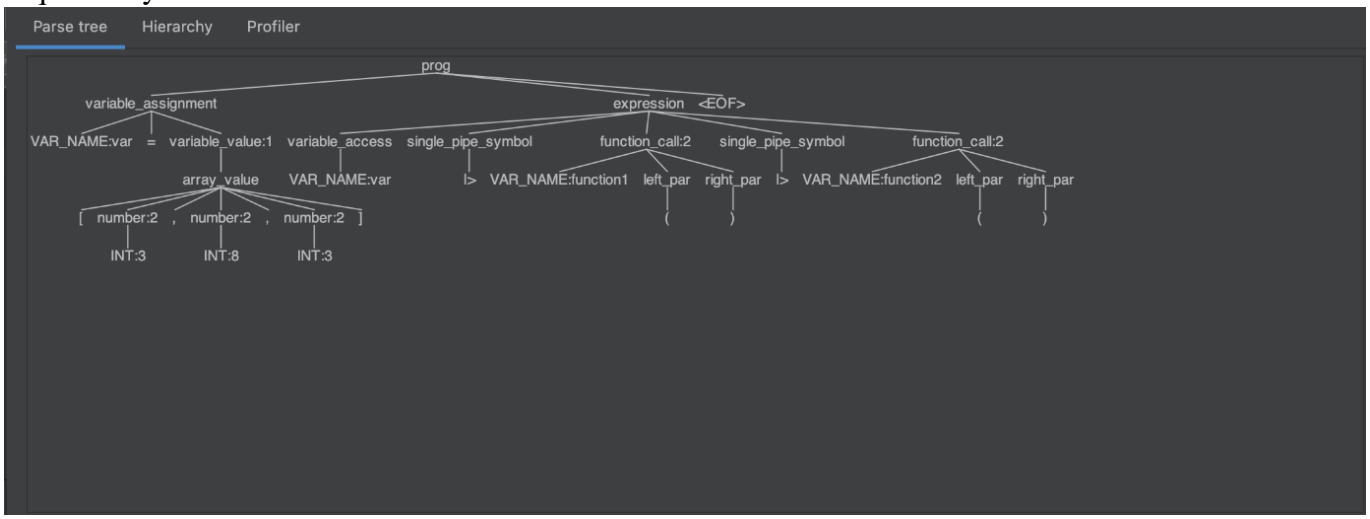

Figure 1.3.1 - Parse tree

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:** Several literal and symbolic tokens are defined, such as '=', '[', ']', '-', '| > ', '(', ')', 'VAR_NAME', 'FLOAT', 'STRING', 'INT', and 'WS' (whitespace).
- **Pipeline_flagContext class:** This class represents a context for pipeline flags. It inherits from ParserRuleContext. However, in the provided code, it appears empty, with no actual parsing logic defined inside it.

Rules

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

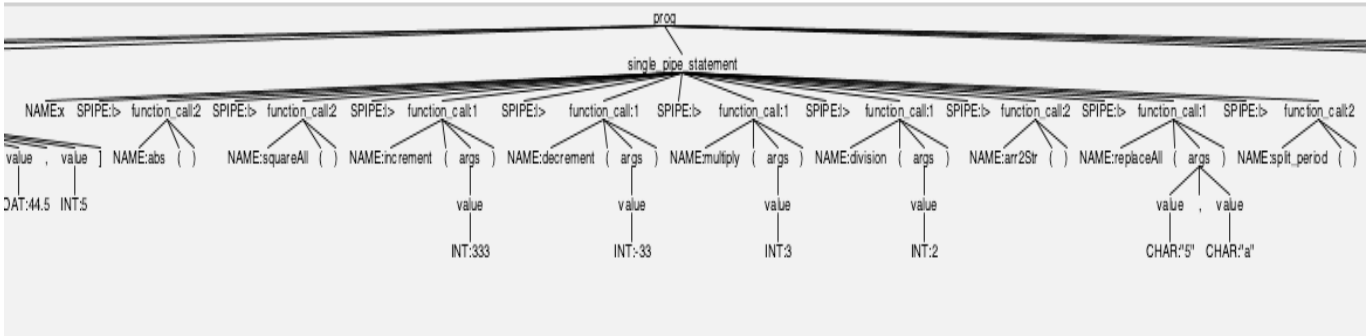

Figure 1.4.1 - Parse Tree Prog

variable_assignment: This rule defines how variables are assigned values.

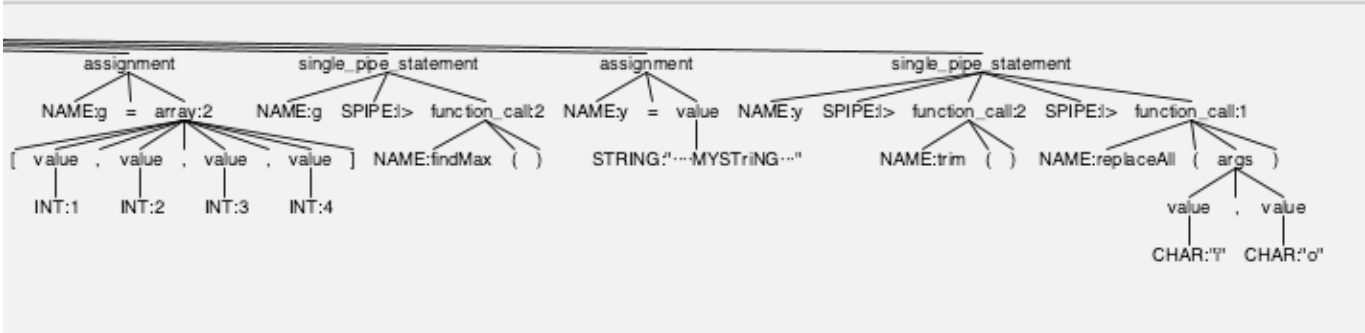

Figure 1.4.2 - Parse Tree Variable Assignment

Expression: This rule represents an expression in the language. It includes;

- `variable_access`,
- `single_pipe_symbol`,
- `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.

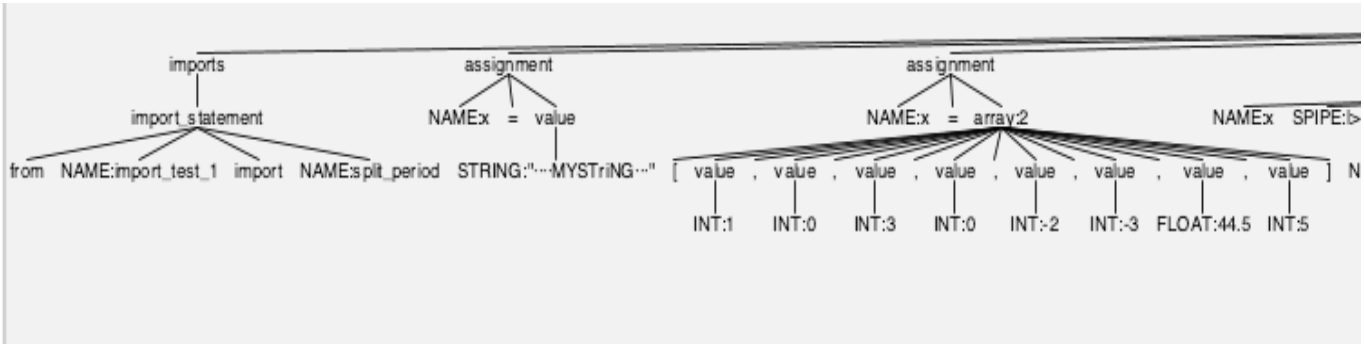

Figure 1.4.3 - Parse Tree Array Value

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:

ExprListener code defines a custom listener class named ExprListener, which has been partialy generated by ANTLR and extends the ParseTreeListener class provided by it.

```python
from antlr4 import *
if "." in __name__:
    from .ExprParser import ExprParser
else:
    from ExprParser import ExprParser

from FunctionHandler import FunctionHandler

# This class defines a complete listener for a parse tree produced by ExprParser.
2 usages
class ExprListener(ParseTreeListener):

    class variable:
        def __init__(self, name, value=None, type=None):
            self.name = name
            self.value = value
            self.type = type

        def __str__(self):
            return str(self.value)

    def __init__(self, debug=False):
        self.variables = []
        self.current_variable = None
        self.function_handler = FunctionHandler()
        self.debug = debug

    1 usage
    def add_variable(self, variable):
        for var in self.variables:
            if var.name == variable.name:
                var.value = variable.value
                var.type = variable.type
                return

        self.variables.append(variable)

    3 usages
    def get_var(self, name):
        for variable in self.variables:
            if variable.name == name:
                return variable
        return None
```

Figure 1.4.4 - Grammar Listener

Variable: __init__: Initializes a variable with a given name, optional value, and type. **__str__:** Returns the string representation of the variable's value.

__init__: Initializes an instance of `ExprListener`. Takes an optional `debug` parameter to enable debug mode. Initializes the attributes `variables` (a list to store variables), `current_variable` (to temporarily hold the current variable being processed), `function_handler` `FunctionHandler` to manage function calls), and `debug` (to store the debug mode status).

add_variable: Adds or updates a variable in the `variables` list. Takes a `variable` parameter representing the variable to be added or updated.

get_var: Retrieves a variable by its name. Takes a `name` parameter and returns the variable with the specified name, or `None` if not found.

get_var_value: Retrieves the value of a variable from the parse tree context. Takes a `var` parameter and returns the value of the variable, converted to the appropriate type. Raises an exception if the variable type is invalid or if the variable is not found.

get_type: Determines the type of a variable from the parse tree context. Takes a `var` parameter and returns the type of the variable as a string (`'int'`, `'float'`, `'char'`, `'str'`, `'bool'`).

parse_arrays: Parses an array from the parse tree context. Takes a `ctx` parameter and returns a list containing the parsed values of the array.

enterImport_statement: Handles the entry of an import statement in the parse tree. Takes a `ctx` parameter representing the context of the import statement. Imports a module and retrieves a function from it, then adds the function to the `function_handler`.

enterSingle_pipe_statement: Handles the entry of a single pipe statement in the parse tree. Takes a `ctx` parameter representing the context of the single pipe statement. Retrieves the variable associated with the statement, executes functions on the variable, updates its value, and prints debug information if debug mode is enabled.

enterAssignment: Handles the entry of an assignment statement in the parse tree. Takes a `ctx` parameter representing the context of the assignment statement. Creates or updates the current variable based on the assignment context, parses the value or array being assigned, and sets the variable's value and type.

exitAssignment: Handles the exit of an assignment statement in the parse tree. Takes a `ctx` parameter representing the context of the assignment statement.

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

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

Figure 1.4.5 - 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.

```python
class FunctionHandler:
    def __init__(self):
        self.function_map = {
            "removeAllZero": self.removeAllZero,
            "filterPositive": self.filterPositive,
            "filterNegative": self.filterNegative,
            "filterEven": self.filterEven,
            "filterOdd": self.filterOdd,
            "squareAll": self.square,
            "doubleAll": self.double,
            "toString": self.toString,
            "sum": self.sum,
            "average": self.average,
            "findMin": self.findMin,
            "findMax": self.findMax,
            "arr2Str": self.arr2Str,
            "abs": self.abs,
            "increment": self.increment,
            "decrement": self.decrement,
            "multiply": self.multiply,
            "division": self.division,
            "sortAscending": self.sortAscending,
            "sortDescending": self.sortDescending,
            # ------------------------------------ String ----------------------------------------
            "split": self.split,
            "substring": self.substring,
            "replaceAll": self.replaceAll,
            "replaceFirst": self.replaceFirst,
            "toLowerCase": self.toLowerCase,
            "toUpperCase": self.toUpperCase,
            "capitalize": self.capitalize,
            "trim": self.trim,
            "trimStart": self.trimStart,
            "trimEnd": self.trimEnd
        }

        self.added_functions_map = {}

    2 usages
    def add_function(self, function_name, function):
        self.added_functions_map[function_name] = function
```

Figure 1.4.6 - FunctionHandler Object

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

The FunctionHandler class is designed to map and execute a variety of predefined functions on arrays and strings. It includes mechanisms for adding custom functions and validating input types and argument counts. The __init__ method initializes the class with a ‘function_map‘ containing predefined functions and an ‘added_functions_map‘ for user-defined functions. The ‘add_function‘ method allows users to add custom functions to the handler, stored in ‘added_functions_map‘.

```
def execute(self, var, function_name, args):
    if function_name in self.function_map:
        function = self.function_map[function_name]
        expected_args = function.__code__.co_argcount - 2
        if expected_args != len(args):
            raise Exception(f"Expected {expected_args} arguments, but got {len(args)}")
        if args:
            return function(var, *args)
        return function(var)

    if function_name in self.added_functions_map:
        function = self.added_functions_map[function_name]
        expected_args = function.__code__.co_argcount - 1
        if expected_args != len(args):
            raise Exception(f"Expected {expected_args} arguments, but got {len(args)}")
        if args:
            return function(var, *args)
        return function(var)

    raise Exception(f"Function {function_name} not found")
```

Figure 1.4.7 - 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 1.4.8 - 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 1.4.9 - Example Result

Conclusions

DSLs play an integral role in modern software development by providing specialized languages that address the unique needs and challenges of specific domains. In this report, we have explored the development and implementation of a domain specific pipeline-oriented workflow in Python. 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 language enable users to develop flexible, scalable, and maintainable solutions tailored to their specific requirements.

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. The implementation involves a custom listener class, MyListener, and error handling through MyErrorListener, both based on ANTLR. Additionally, the parser tree provides a visual representation of how the DSL interprets and processes the defined pipelines, offering insights into its internal workings.

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.

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

driver.py

```
from antlr4 import *
from ExprLexer import ExprLexer
from ExprParser import ExprParser
from ExprListener import ExprListener


def main():
    input_stream = FileStream("input.txt")
    lexer = ExprLexer(input_stream)
    token_stream = CommonTokenStream(lexer)
    parser = ExprParser(token_stream)
    tree = parser.prog()
    listener = ExprListener(debug=True)
    walker = ParseTreeWalker()
    walker.walk(listener, tree)


if __name__ == '__main__':
    main()
```

CustomListener.py

```python
# Generated from Expr.g4 by ANTLR 4.13.1
from antlr4 import *
if "." in __name__:
    from .ExprParser import ExprParser
else:
    from ExprParser import ExprParser

from FunctionHandler import FunctionHandler

# This class defines a complete listener for a parse tree produced by ExprParser.
class ExprListener(ParseTreeListener):

    class variable:
        def __init__(self, name, value=None, type=None):
            self.name = name
            self.value = value
            self.type = type

        def __str__(self):
            return str(self.value)

    def __init__(self):
        self.variables = []
        self.current_variable = None
        self.function_handler = FunctionHandler()

    def add_variable(self, variable):
        for var in self.variables:
            if var.name == variable.name:
                var.value = variable.value
                var.type = variable.type
                return

        self.variables.append(variable)

    def get_var(self, name):
        for variable in self.variables:
            if variable.name == name:
                return variable
        return None

    def get_var_value(self, var):
        if var.INT():
            return int(var.INT().getText())
        elif var.FLOAT():
            return float(var.FLOAT().getText())
        elif var.CHAR():
            return var.CHAR().getText().replace("'", '')
        elif var.STRING():
            return var.STRING().getText().replace('"', '')
        elif var.BOOL():
            return var.BOOL().getText() == 'true'
        elif var.NAME():
            var = self.get_var(var.NAME().getText())
            if var:
                return var.value
            else:
                raise Exception("Variable not found")
```

```
59              else:
60                  raise Exception("Invalid type")
61
62          def get_type(self, var):
63              if var.INT():
64                  return 'int'
65              elif var.FLOAT():
66                  return 'float'
67              elif var.CHAR():
68                  return 'char'
69              elif var.STRING():
70                  return 'str'
71              elif var.BOOL():
72                  return 'bool'
73              else:
74                  pass
75
76          def parse_arrays(self, ctx):
77              arr = []
78              for child in ctx.children:
79                  if child in ctx.value():
80                      # print("value -> ", child.getText())
81                      arr.append(self.get_var_value(child))
82                  elif child in ctx.array():
83                      # print("array -> ", child.getText())
84                      arr.append(self.parse_arrays(child))
85              return arr
86
87          # Enter a parse tree produced by ExprParser#prog.
88          def enterProg(self, ctx:ExprParser.ProgContext):
89              pass
90
91          # Exit a parse tree produced by ExprParser#prog.
92          def exitProg(self, ctx:ExprParser.ProgContext):
93              for var in self.variables: print(f'{var.name} => {var.value} ({var.type})')
94              pass
95
96
97          # Enter a parse tree produced by ExprParser#imports.
98          def enterImports(self, ctx:ExprParser.ImportsContext):
99              pass
100
101
102         # Exit a parse tree produced by ExprParser#imports.
103         def exitImports(self, ctx:ExprParser.ImportsContext):
104             pass
105
106
107         # Enter a parse tree produced by ExprParser#import_statement.
108         def enterImport_statement(self, ctx:ExprParser.Import_statementContext):
109             import importlib
110
111
112             file_name = ctx.NAME()[0].getText()
113             function_name = ctx.NAME()[1].getText()
114
115             module = importlib.import_module(file_name)
116             function = getattr(module, function_name)
117
118
119
```

```
            self.function_handler.add_function(function_name, function)




    # Exit a parse tree produced by ExprParser#import_statement.
    def exitImport_statement(self, ctx:ExprParser.Import_statementContext):
        pass


    # Enter a parse tree produced by ExprParser#single_pipe_statement.
    def enterSingle_pipe_statement(self, ctx:ExprParser.Single_pipe_statementContext):
        var_name = ctx.NAME().getText()
        var = self.get_var(var_name)

        print("------------------------------")
        print(f'{var_name} = {var.value}')

        if not var:
            raise Exception("Variable not found")

        for function in ctx.function_call():
            if function.NAME().getText() == 'print':
                print(var.value)
            else:
                args = []
                if function.args():
                    for arg in function.args().value():
                        args.append(self.get_var_value(arg))
                print("->", var)
                result = self.function_handler.execute(var.value, function.NAME().getText(),
                ↪  args)
                print(f"----> ({function.NAME().getText()})", result)
                result_type = type(result).__name__
                var.value = result
                var.type = result_type


        print(f'{var_name} = {var.value}')
        print("------------------------------")

    # Exit a parse tree produced by ExprParser#single_pipe_statement.
    def exitSingle_pipe_statement(self, ctx:ExprParser.Single_pipe_statementContext):
        pass


    # Enter a parse tree produced by ExprParser#function_call.
    def enterFunction_call(self, ctx:ExprParser.Function_callContext):
        pass

    # Exit a parse tree produced by ExprParser#function_call.
    def exitFunction_call(self, ctx:ExprParser.Function_callContext):
        pass


    # Enter a parse tree produced by ExprParser#assignment.
    def enterAssignment(self, ctx:ExprParser.AssignmentContext):
        arr = []
        var_name = ctx.NAME().getText()
        var = self.get_var(var_name)
```

```
180
181             if var:
182                 self.current_variable = var
183             else:
184                 self.current_variable = self.variable(var_name)
185
186             if ctx.value():
187                 if ctx.value().INT():
188                     self.current_variable.value = int(ctx.value().INT().getText())
189                     self.current_variable.type = 'int'
190                 elif ctx.value().FLOAT():
191                     self.current_variable.value = float(ctx.value().FLOAT().getText())
192                     self.current_variable.type = 'float'
193                 elif ctx.value().CHAR():
194                     self.current_variable.value = ctx.value().CHAR().getText().replace("'", '')
195                     self.current_variable.type = 'char'
196                 elif ctx.value().STRING():
197                     self.current_variable.value = ctx.value().STRING().getText().replace('"', '')
198                     self.current_variable.type = 'str'
199                 elif ctx.value().BOOL():
200                     self.current_variable.value = ctx.value().BOOL().getText() == 'true'
201                     self.current_variable.type = 'bool'
202                 else:
203                     raise Exception("Invalid type")
204             elif ctx.array():
205                 self.current_variable.value = self.parse_arrays(ctx.array())
206                 self.current_variable.type = 'list'
207             else:
208                 raise Exception("Invalid type")
209
210         # Exit a parse tree produced by ExprParser#assignment.
211         def exitAssignment(self, ctx:ExprParser.AssignmentContext):
212             self.add_variable(self.current_variable)
213             self.current_variable = None
214
215
216         # Enter a parse tree produced by ExprParser#array.
217         def enterArray(self, ctx:ExprParser.ArrayContext):
218             pass
219
220         # Exit a parse tree produced by ExprParser#array.
221         def exitArray(self, ctx:ExprParser.ArrayContext):
222             pass
223
224
225         # Enter a parse tree produced by ExprParser#value.
226         def enterValue(self, ctx:ExprParser.ValueContext):
227             pass
228
229         # Exit a parse tree produced by ExprParser#value.
230         def exitValue(self, ctx:ExprParser.ValueContext):
231             pass
232
233
234         # Enter a parse tree produced by ExprParser#args.
235         def enterArgs(self, ctx:ExprParser.ArgsContext):
236             pass
237
238         # Exit a parse tree produced by ExprParser#args.
239         def exitArgs(self, ctx:ExprParser.ArgsContext):
240             pass
```

```
241
242
243
244     del ExprParser
```

ExprLexer.py

```
# Generated from Expr.g4 by ANTLR 4.13.1
from antlr4 import *
from io import StringIO
import sys
if sys.version_info[1] > 5:
    from typing import TextIO
else:
    from typing.io import TextIO


def serializedATN():
    return [
        4,0,16,136,6,-1,2,0,7,0,2,1,7,1,2,2,7,2,2,3,7,3,2,4,7,4,2,5,7,5,
        2,6,7,6,2,7,7,7,2,8,7,8,2,9,7,9,2,10,7,10,2,11,7,11,2,12,7,12,2,
        13,7,13,2,14,7,14,2,15,7,15,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,2,1,2,1,3,1,3,1,4,1,4,1,5,1,5,1,6,1,6,1,7,1,7,1,8,
        4,8,59,8,8,11,8,12,8,60,1,8,1,8,4,8,65,8,8,11,8,12,8,66,3,8,69,8,
        8,1,9,4,9,72,8,9,11,9,12,9,73,1,9,1,9,4,9,78,8,9,11,9,12,9,79,1,
        9,1,9,4,9,84,8,9,11,9,12,9,85,1,9,1,9,4,9,90,8,9,11,9,12,9,91,3,
        9,94,8,9,1,10,1,10,1,10,1,10,1,11,1,11,5,11,102,8,11,10,11,12,11,
        105,9,11,1,11,1,11,1,12,4,12,110,8,12,11,12,12,12,111,1,12,1,12,
        1,13,1,13,5,13,118,8,13,10,13,12,13,121,9,13,1,14,1,14,1,14,1,14,
        1,14,1,14,1,14,1,14,1,14,3,14,132,8,14,1,15,1,15,1,15,0,0,16,1,1,
        3,2,5,3,7,4,9,5,11,6,13,7,15,8,17,9,19,10,21,11,23,12,25,13,27,14,
        29,15,31,16,1,0,10,1,0,48,57,1,0,45,45,1,0,46,46,1,0,34,34,3,0,48,
        57,65,90,97,122,3,0,9,10,13,13,32,32,2,0,65,90,97,122,4,0,48,57,
        65,90,95,95,97,122,1,0,124,124,1,0,62,62,147,0,1,1,0,0,0,0,3,1,0,
        0,0,0,5,1,0,0,0,0,7,1,0,0,0,0,9,1,0,0,0,0,11,1,0,0,0,0,13,1,0,0,
        0,0,15,1,0,0,0,0,17,1,0,0,0,0,19,1,0,0,0,0,21,1,0,0,0,0,23,1,0,0,
        0,0,25,1,0,0,0,0,27,1,0,0,0,0,29,1,0,0,0,0,31,1,0,0,0,1,33,1,0,0,
        0,3,38,1,0,0,0,5,45,1,0,0,0,7,47,1,0,0,0,9,49,1,0,0,0,11,51,1,0,
        0,0,13,53,1,0,0,0,15,55,1,0,0,0,17,68,1,0,0,0,19,93,1,0,0,0,21,95,
        1,0,0,0,23,99,1,0,0,0,25,109,1,0,0,0,27,115,1,0,0,0,29,131,1,0,0,
        0,31,133,1,0,0,0,33,34,5,102,0,0,34,35,5,114,0,0,35,36,5,111,0,0,
        36,37,5,109,0,0,37,2,1,0,0,0,38,39,5,105,0,0,39,40,5,109,0,0,40,
        41,5,112,0,0,41,42,5,111,0,0,42,43,5,114,0,0,43,44,5,116,0,0,44,
        4,1,0,0,0,45,46,5,40,0,0,46,6,1,0,0,0,47,48,5,41,0,0,48,8,1,0,0,
        0,49,50,5,61,0,0,50,10,1,0,0,0,51,52,5,91,0,0,52,12,1,0,0,0,53,54,
        5,93,0,0,54,14,1,0,0,0,55,56,5,44,0,0,56,16,1,0,0,0,57,59,7,0,0,
        0,58,57,1,0,0,0,59,60,1,0,0,0,60,58,1,0,0,0,60,61,1,0,0,0,61,69,
        1,0,0,0,62,64,7,1,0,0,63,65,7,0,0,0,64,63,1,0,0,0,65,66,1,0,0,0,
        66,64,1,0,0,0,66,67,1,0,0,0,67,69,1,0,0,0,68,58,1,0,0,0,68,62,1,
        0,0,0,69,18,1,0,0,0,70,72,7,0,0,0,71,70,1,0,0,0,72,73,1,0,0,0,73,
        71,1,0,0,0,73,74,1,0,0,0,74,75,1,0,0,0,75,77,7,2,0,0,76,78,7,0,0,
        0,77,76,1,0,0,0,78,79,1,0,0,0,79,77,1,0,0,0,79,80,1,0,0,0,80,94,
        1,0,0,0,81,83,7,1,0,0,82,84,7,0,0,0,83,82,1,0,0,0,84,85,1,0,0,0,
        85,83,1,0,0,0,85,86,1,0,0,0,86,87,1,0,0,0,87,89,7,2,0,0,88,90,7,
        0,0,0,89,88,1,0,0,0,90,91,1,0,0,0,91,89,1,0,0,0,91,92,1,0,0,0,92,
        94,1,0,0,0,93,71,1,0,0,0,93,81,1,0,0,0,94,20,1,0,0,0,95,96,7,3,0,
        0,96,97,7,4,0,0,97,98,7,3,0,0,98,22,1,0,0,0,99,103,7,3,0,0,100,102,
        8,3,0,0,101,100,1,0,0,0,102,105,1,0,0,0,103,101,1,0,0,0,103,104,
        1,0,0,0,104,106,1,0,0,0,105,103,1,0,0,0,106,107,7,3,0,0,107,24,1,
        0,0,0,108,110,7,5,0,0,109,108,1,0,0,0,110,111,1,0,0,0,111,109,1,
        0,0,0,111,112,1,0,0,0,112,113,1,0,0,0,113,114,6,12,0,0,114,26,1,
        0,0,0,115,119,7,6,0,0,116,118,7,7,0,0,117,116,1,0,0,0,118,121,1,
        0,0,0,119,117,1,0,0,0,119,120,1,0,0,0,120,28,1,0,0,0,121,119,1,0,
        0,0,122,123,5,116,0,0,123,124,5,114,0,0,124,125,5,117,0,0,125,132,
        5,101,0,0,126,127,5,102,0,0,127,128,5,97,0,0,128,129,5,108,0,0,129,
```

```
            130,5,115,0,0,130,132,5,101,0,0,131,122,1,0,0,0,131,126,1,0,0,0,
            132,30,1,0,0,0,133,134,7,8,0,0,134,135,7,9,0,0,135,32,1,0,0,0,13,
            0,60,66,68,73,79,85,91,93,103,111,119,131,1,6,0,0
        ]

    class ExprLexer(Lexer):

        atn = ATNDeserializer().deserialize(serializedATN())

        decisionsToDFA = [ DFA(ds, i) for i, ds in enumerate(atn.decisionToState) ]

        T__0 = 1
        T__1 = 2
        T__2 = 3
        T__3 = 4
        T__4 = 5
        T__5 = 6
        T__6 = 7
        T__7 = 8
        INT = 9
        FLOAT = 10
        CHAR = 11
        STRING = 12
        WS = 13
        NAME = 14
        BOOL = 15
        SPIPE = 16

        channelNames = [ u"DEFAULT_TOKEN_CHANNEL", u"HIDDEN" ]

        modeNames = [ "DEFAULT_MODE" ]

        literalNames = [ "<INVALID>",
                "'from'", "'import'", "'('", "')'", "'='", "'['", "']'", "','" ]

        symbolicNames = [ "<INVALID>",
                "INT", "FLOAT", "CHAR", "STRING", "WS", "NAME", "BOOL", "SPIPE" ]

        ruleNames = [ "T__0", "T__1", "T__2", "T__3", "T__4", "T__5", "T__6",
                      "T__7", "INT", "FLOAT", "CHAR", "STRING", "WS", "NAME",
                      "BOOL", "SPIPE" ]

        grammarFileName = "Expr.g4"

        def __init__(self, input=None, output:TextIO = sys.stdout):
            super().__init__(input, output)
            self.checkVersion("4.13.1")
            self._interp = LexerATNSimulator(self, self.atn, self.decisionsToDFA,
            ↪  PredictionContextCache())
            self._actions = None
            self._predicates = None
```

ExprListener.py

```python
# Generated from Expr.g4 by ANTLR 4.13.1
from antlr4 import *
if "." in __name__:
    from .ExprParser import ExprParser
else:
    from ExprParser import ExprParser

from FunctionHandler import FunctionHandler

# This class defines a complete listener for a parse tree produced by ExprParser.
class ExprListener(ParseTreeListener):

    class variable:
        def __init__(self, name, value=None, type=None):
            self.name = name
            self.value = value
            self.type = type

        def __str__(self):
            return str(self.value)

    def __init__(self, debug=False):
        self.variables = []
        self.current_variable = None
        self.function_handler = FunctionHandler()
        self.debug = debug

    def add_variable(self, variable):
        for var in self.variables:
            if var.name == variable.name:
                var.value = variable.value
                var.type = variable.type
                return

        self.variables.append(variable)

    def get_var(self, name):
        for variable in self.variables:
            if variable.name == name:
                return variable
        return None

    def get_var_value(self, var):
        if var.INT():
            return int(var.INT().getText())
        elif var.FLOAT():
            return float(var.FLOAT().getText())
        elif var.CHAR():
            return var.CHAR().getText().replace("'", '')
        elif var.STRING():
            return var.STRING().getText().replace('"', '')
        elif var.BOOL():
            return var.BOOL().getText() == 'true'
        elif var.NAME():
            var = self.get_var(var.NAME().getText())
            if var:
                return var.value
            else:
```

```
                raise Exception("Variable not found")
            else:
                raise Exception("Invalid type")

    def get_type(self, var):
        if var.INT():
            return 'int'
        elif var.FLOAT():
            return 'float'
        elif var.CHAR():
            return 'char'
        elif var.STRING():
            return 'str'
        elif var.BOOL():
            return 'bool'
        else:
            pass

    def parse_arrays(self, ctx):
        arr = []
        for child in ctx.children:
            if child in ctx.value():
                # print("value -> ", child.getText())
                arr.append(self.get_var_value(child))
            elif child in ctx.array():
                # print("array -> ", child.getText())
                arr.append(self.parse_arrays(child))
        return arr

    # Enter a parse tree produced by ExprParser#prog.
    def enterProg(self, ctx:ExprParser.ProgContext):
        pass

    # Exit a parse tree produced by ExprParser#prog.
    def exitProg(self, ctx:ExprParser.ProgContext):
        if self.debug:
            for var in self.variables: print(f'{var.name} => {var.value} ({var.type})')
        pass


    # Enter a parse tree produced by ExprParser#imports.
    def enterImports(self, ctx:ExprParser.ImportsContext):
        pass


    # Exit a parse tree produced by ExprParser#imports.
    def exitImports(self, ctx:ExprParser.ImportsContext):
        pass


    # Enter a parse tree produced by ExprParser#import_statement.
    def enterImport_statement(self, ctx:ExprParser.Import_statementContext):
        import importlib

        file_name = ctx.NAME()[0].getText()
        function_name = ctx.NAME()[1].getText()

        module = importlib.import_module(file_name)
        function = getattr(module, function_name)

        self.function_handler.add_function(function_name, function)
```

```
120
121
122
123
124
125         # Exit a parse tree produced by ExprParser#import_statement.
126         def exitImport_statement(self, ctx:ExprParser.Import_statementContext):
127             pass
128
129
130         # Enter a parse tree produced by ExprParser#single_pipe_statement.
131         def enterSingle_pipe_statement(self, ctx:ExprParser.Single_pipe_statementContext):
132             var_name = ctx.NAME().getText()
133             var = self.get_var(var_name)
134
135             if self.debug:
136                 print("------------------------------")
137                 print(f'{var_name} = <<{var.value}>>')
138
139             if not var:
140                 raise Exception("Variable not found")
141
142             for function in ctx.function_call():
143                 if function.NAME().getText() == 'print':
144                     print(var.value)
145                 else:
146                     args = []
147                     if function.args():
148                         for arg in function.args().value():
149                             args.append(self.get_var_value(arg))
150                     result = self.function_handler.execute(var.value, function.NAME().getText(),
                         ↪  args)
151                     if self.debug:
152                         print(f"----> ({function.NAME().getText()}) <<{result}>>")
153                     result_type = type(result).__name__
154                     var.value = result
155                     var.type = result_type
156
157             if self.debug:
158                 print(f'{var_name} = {var.value}')
159                 print("------------------------------")
160
161         # Exit a parse tree produced by ExprParser#single_pipe_statement.
162         def exitSingle_pipe_statement(self, ctx:ExprParser.Single_pipe_statementContext):
163             pass
164
165
166         # Enter a parse tree produced by ExprParser#function_call.
167         def enterFunction_call(self, ctx:ExprParser.Function_callContext):
168             pass
169
170         # Exit a parse tree produced by ExprParser#function_call.
171         def exitFunction_call(self, ctx:ExprParser.Function_callContext):
172             pass
173
174
175         # Enter a parse tree produced by ExprParser#assignment.
176         def enterAssignment(self, ctx:ExprParser.AssignmentContext):
177             arr = []
178             var_name = ctx.NAME().getText()
179             var = self.get_var(var_name)
```

```
180
181             if var:
182                 self.current_variable = var
183             else:
184                 self.current_variable = self.variable(var_name)
185
186             if ctx.value():
187                 if ctx.value().INT():
188                     self.current_variable.value = int(ctx.value().INT().getText())
189                     self.current_variable.type = 'int'
190                 elif ctx.value().FLOAT():
191                     self.current_variable.value = float(ctx.value().FLOAT().getText())
192                     self.current_variable.type = 'float'
193                 elif ctx.value().CHAR():
194                     self.current_variable.value = ctx.value().CHAR().getText().replace("'", '')
195                     self.current_variable.type = 'char'
196                 elif ctx.value().STRING():
197                     self.current_variable.value = ctx.value().STRING().getText().replace('"', '')
198                     self.current_variable.type = 'str'
199                 elif ctx.value().BOOL():
200                     self.current_variable.value = ctx.value().BOOL().getText() == 'true'
201                     self.current_variable.type = 'bool'
202                 else:
203                     raise Exception("Invalid type")
204             elif ctx.array():
205                 self.current_variable.value = self.parse_arrays(ctx.array())
206                 self.current_variable.type = 'list'
207             else:
208                 raise Exception("Invalid type")
209
210         # Exit a parse tree produced by ExprParser#assignment.
211         def exitAssignment(self, ctx:ExprParser.AssignmentContext):
212             self.add_variable(self.current_variable)
213             self.current_variable = None
214
215
216         # Enter a parse tree produced by ExprParser#array.
217         def enterArray(self, ctx:ExprParser.ArrayContext):
218             pass
219
220         # Exit a parse tree produced by ExprParser#array.
221         def exitArray(self, ctx:ExprParser.ArrayContext):
222             pass
223
224
225         # Enter a parse tree produced by ExprParser#value.
226         def enterValue(self, ctx:ExprParser.ValueContext):
227             pass
228
229         # Exit a parse tree produced by ExprParser#value.
230         def exitValue(self, ctx:ExprParser.ValueContext):
231             pass
232
233
234         # Enter a parse tree produced by ExprParser#args.
235         def enterArgs(self, ctx:ExprParser.ArgsContext):
236             pass
237
238         # Exit a parse tree produced by ExprParser#args.
239         def exitArgs(self, ctx:ExprParser.ArgsContext):
240             pass
```

```
241
242
243
244     del ExprParser
```

ExprParser.py

```
# Generated from Expr.g4 by ANTLR 4.13.1
# encoding: utf-8
from antlr4 import *
from io import StringIO
import sys
if sys.version_info[1] > 5:
        from typing import TextIO
else:
        from typing.io import TextIO

def serializedATN():
    return [
        4,1,16,102,2,0,7,0,2,1,7,1,2,2,7,2,2,3,7,3,2,4,7,4,2,5,7,5,2,6,7,
        6,2,7,7,7,2,8,7,8,1,0,1,0,1,0,5,0,22,8,0,10,0,12,0,25,9,0,4,0,27,
        8,0,11,0,12,0,28,1,1,5,1,32,8,1,10,1,12,1,35,9,1,1,2,1,2,1,2,1,2,
        1,2,1,3,1,3,1,3,4,3,45,8,3,11,3,12,3,46,1,4,1,4,1,4,1,4,1,4,1,4,
        1,4,1,4,3,4,57,8,4,1,5,1,5,1,5,1,5,3,5,63,8,5,1,6,1,6,1,6,1,6,1,
        6,3,6,70,8,6,1,6,1,6,1,6,3,6,75,8,6,5,6,77,8,6,10,6,12,6,80,9,6,
        1,6,1,6,3,6,84,8,6,1,7,1,7,1,8,1,8,3,8,90,8,8,1,8,1,8,1,8,3,8,95,
        8,8,5,8,97,8,8,10,8,12,8,100,9,8,1,8,0,0,9,0,2,4,6,8,10,12,14,16,
        0,1,2,0,9,12,14,15,105,0,18,1,0,0,0,2,33,1,0,0,0,4,36,1,0,0,0,6,
        41,1,0,0,0,8,56,1,0,0,0,10,58,1,0,0,0,12,83,1,0,0,0,14,85,1,0,0,
        0,16,89,1,0,0,0,18,26,3,2,1,0,19,23,3,10,5,0,20,22,3,6,3,0,21,20,
        1,0,0,0,22,25,1,0,0,0,23,21,1,0,0,0,23,24,1,0,0,0,24,27,1,0,0,0,
        25,23,1,0,0,0,26,19,1,0,0,0,27,28,1,0,0,0,28,26,1,0,0,0,28,29,1,
        0,0,0,29,1,1,0,0,0,30,32,3,4,2,0,31,30,1,0,0,0,32,35,1,0,0,0,33,
        31,1,0,0,0,33,34,1,0,0,0,34,3,1,0,0,0,35,33,1,0,0,0,36,37,5,1,0,
        0,37,38,5,14,0,0,38,39,5,2,0,0,39,40,5,14,0,0,40,5,1,0,0,0,41,44,
        5,14,0,0,42,43,5,16,0,0,43,45,3,8,4,0,44,42,1,0,0,0,45,46,1,0,0,
        0,46,44,1,0,0,0,46,47,1,0,0,0,47,7,1,0,0,0,48,49,5,14,0,0,49,50,
        5,3,0,0,50,51,3,16,8,0,51,52,5,4,0,0,52,57,1,0,0,0,53,54,5,14,0,
        0,54,55,5,3,0,0,55,57,5,4,0,0,56,48,1,0,0,0,56,53,1,0,0,0,57,9,1,
        0,0,0,58,59,5,14,0,0,59,62,5,5,0,0,60,63,3,14,7,0,61,63,3,12,6,0,
        62,60,1,0,0,0,62,61,1,0,0,0,63,11,1,0,0,0,64,65,5,6,0,0,65,84,5,
        7,0,0,66,69,5,6,0,0,67,70,3,14,7,0,68,70,3,12,6,0,69,67,1,0,0,0,
        69,68,1,0,0,0,70,78,1,0,0,0,71,74,5,8,0,0,72,75,3,14,7,0,73,75,3,
        12,6,0,74,72,1,0,0,0,74,73,1,0,0,0,75,77,1,0,0,0,76,71,1,0,0,0,77,
        80,1,0,0,0,78,76,1,0,0,0,78,79,1,0,0,0,79,81,1,0,0,0,80,78,1,0,0,
        0,81,82,5,7,0,0,82,84,1,0,0,0,83,64,1,0,0,0,83,66,1,0,0,0,84,13,
        1,0,0,0,85,86,7,0,0,0,86,15,1,0,0,0,87,90,3,14,7,0,88,90,3,12,6,
        0,89,87,1,0,0,0,89,88,1,0,0,0,90,98,1,0,0,0,91,94,5,8,0,0,92,95,
        3,14,7,0,93,95,3,12,6,0,94,92,1,0,0,0,94,93,1,0,0,0,95,97,1,0,0,
        0,96,91,1,0,0,0,97,100,1,0,0,0,98,96,1,0,0,0,98,99,1,0,0,0,99,17,
        1,0,0,0,100,98,1,0,0,0,13,23,28,33,46,56,62,69,74,78,83,89,94,98
    ]

class ExprParser ( Parser ):

    grammarFileName = "Expr.g4"

    atn = ATNDeserializer().deserialize(serializedATN())

    decisionsToDFA = [ DFA(ds, i) for i, ds in enumerate(atn.decisionToState) ]

    sharedContextCache = PredictionContextCache()

    literalNames = [ "<INVALID>", "'from'", "'import'", "'('", "')'", "'='",
                     "'['", "']'", "','" ]
```

```
59
60          symbolicNames = [ "<INVALID>", "<INVALID>", "<INVALID>", "<INVALID>",
61                            "<INVALID>", "<INVALID>", "<INVALID>", "<INVALID>",
62                            "<INVALID>", "INT", "FLOAT", "CHAR", "STRING", "WS",
63                            "NAME", "BOOL", "SPIPE" ]
64
65          RULE_prog = 0
66          RULE_imports = 1
67          RULE_import_statement = 2
68          RULE_single_pipe_statement = 3
69          RULE_function_call = 4
70          RULE_assignment = 5
71          RULE_array = 6
72          RULE_value = 7
73          RULE_args = 8
74
75          ruleNames =  [ "prog", "imports", "import_statement", "single_pipe_statement",
76                         "function_call", "assignment", "array", "value", "args" ]
77
78          EOF = Token.EOF
79          T__0=1
80          T__1=2
81          T__2=3
82          T__3=4
83          T__4=5
84          T__5=6
85          T__6=7
86          T__7=8
87          INT=9
88          FLOAT=10
89          CHAR=11
90          STRING=12
91          WS=13
92          NAME=14
93          BOOL=15
94          SPIPE=16
95
96          def __init__(self, input:TokenStream, output:TextIO = sys.stdout):
97              super().__init__(input, output)
98              self.checkVersion("4.13.1")
99              self._interp = ParserATNSimulator(self, self.atn, self.decisionsToDFA,
                ↪  self.sharedContextCache)
100             self._predicates = None
101
102
103
104
105         class ProgContext(ParserRuleContext):
106             __slots__ = 'parser'
107
108             def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
109                 super().__init__(parent, invokingState)
110                 self.parser = parser
111
112             def imports(self):
113                 return self.getTypedRuleContext(ExprParser.ImportsContext,0)
114
115
116             def assignment(self, i:int=None):
117                 if i is None:
118                     return self.getTypedRuleContexts(ExprParser.AssignmentContext)
```

```
            else:
                return self.getTypedRuleContext(ExprParser.AssignmentContext,i)


        def single_pipe_statement(self, i:int=None):
            if i is None:
                return self.getTypedRuleContexts(ExprParser.Single_pipe_statementContext)
            else:
                return self.getTypedRuleContext(ExprParser.Single_pipe_statementContext,i)


        def getRuleIndex(self):
            return ExprParser.RULE_prog

        def enterRule(self, listener:ParseTreeListener):
            if hasattr( listener, "enterProg" ):
                listener.enterProg(self)

        def exitRule(self, listener:ParseTreeListener):
            if hasattr( listener, "exitProg" ):
                listener.exitProg(self)




    def prog(self):

        localctx = ExprParser.ProgContext(self, self._ctx, self.state)
        self.enterRule(localctx, 0, self.RULE_prog)
        self._la = 0 # Token type
        try:
            self.enterOuterAlt(localctx, 1)
            self.state = 18
            self.imports()
            self.state = 26
            self._errHandler.sync(self)
            _la = self._input.LA(1)
            while True:
                self.state = 19
                self.assignment()
                self.state = 23
                self._errHandler.sync(self)
                _alt = self._interp.adaptivePredict(self._input,0,self._ctx)
                while _alt!=2 and _alt!=ATN.INVALID_ALT_NUMBER:
                    if _alt==1:
                        self.state = 20
                        self.single_pipe_statement()
                    self.state = 25
                    self._errHandler.sync(self)
                    _alt = self._interp.adaptivePredict(self._input,0,self._ctx)

                self.state = 28
                self._errHandler.sync(self)
                _la = self._input.LA(1)
                if not (_la==14):
                    break

        except RecognitionException as re:
            localctx.exception = re
            self._errHandler.reportError(self, re)
            self._errHandler.recover(self, re)
```

```
            finally:
                self.exitRule()
            return localctx


        class ImportsContext(ParserRuleContext):
            __slots__ = 'parser'

            def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
                super().__init__(parent, invokingState)
                self.parser = parser

            def import_statement(self, i:int=None):
                if i is None:
                    return self.getTypedRuleContexts(ExprParser.Import_statementContext)
                else:
                    return self.getTypedRuleContext(ExprParser.Import_statementContext,i)


            def getRuleIndex(self):
                return ExprParser.RULE_imports

            def enterRule(self, listener:ParseTreeListener):
                if hasattr( listener, "enterImports" ):
                    listener.enterImports(self)

            def exitRule(self, listener:ParseTreeListener):
                if hasattr( listener, "exitImports" ):
                    listener.exitImports(self)




        def imports(self):

            localctx = ExprParser.ImportsContext(self, self._ctx, self.state)
            self.enterRule(localctx, 2, self.RULE_imports)
            self._la = 0 # Token type
            try:
                self.enterOuterAlt(localctx, 1)
                self.state = 33
                self._errHandler.sync(self)
                _la = self._input.LA(1)
                while _la==1:
                    self.state = 30
                    self.import_statement()
                    self.state = 35
                    self._errHandler.sync(self)
                    _la = self._input.LA(1)

            except RecognitionException as re:
                localctx.exception = re
                self._errHandler.reportError(self, re)
                self._errHandler.recover(self, re)
            finally:
                self.exitRule()
            return localctx


        class Import_statementContext(ParserRuleContext):
            __slots__ = 'parser'
```

```
241
242             def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
243                 super().__init__(parent, invokingState)
244                 self.parser = parser
245
246             def NAME(self, i:int=None):
247                 if i is None:
248                     return self.getTokens(ExprParser.NAME)
249                 else:
250                     return self.getToken(ExprParser.NAME, i)
251
252             def getRuleIndex(self):
253                 return ExprParser.RULE_import_statement
254
255             def enterRule(self, listener:ParseTreeListener):
256                 if hasattr( listener, "enterImport_statement" ):
257                     listener.enterImport_statement(self)
258
259             def exitRule(self, listener:ParseTreeListener):
260                 if hasattr( listener, "exitImport_statement" ):
261                     listener.exitImport_statement(self)
262
263
264
265
266         def import_statement(self):
267
268             localctx = ExprParser.Import_statementContext(self, self._ctx, self.state)
269             self.enterRule(localctx, 4, self.RULE_import_statement)
270             try:
271                 self.enterOuterAlt(localctx, 1)
272                 self.state = 36
273                 self.match(ExprParser.T__0)
274                 self.state = 37
275                 self.match(ExprParser.NAME)
276                 self.state = 38
277                 self.match(ExprParser.T__1)
278                 self.state = 39
279                 self.match(ExprParser.NAME)
280             except RecognitionException as re:
281                 localctx.exception = re
282                 self._errHandler.reportError(self, re)
283                 self._errHandler.recover(self, re)
284             finally:
285                 self.exitRule()
286             return localctx
287
288
289         class Single_pipe_statementContext(ParserRuleContext):
290             __slots__ = 'parser'
291
292             def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
293                 super().__init__(parent, invokingState)
294                 self.parser = parser
295
296             def NAME(self):
297                 return self.getToken(ExprParser.NAME, 0)
298
299             def SPIPE(self, i:int=None):
300                 if i is None:
301                     return self.getTokens(ExprParser.SPIPE)
```

```
            else:
                return self.getToken(ExprParser.SPIPE, i)

        def function_call(self, i:int=None):
            if i is None:
                return self.getTypedRuleContexts(ExprParser.Function_callContext)
            else:
                return self.getTypedRuleContext(ExprParser.Function_callContext,i)


        def getRuleIndex(self):
            return ExprParser.RULE_single_pipe_statement

        def enterRule(self, listener:ParseTreeListener):
            if hasattr( listener, "enterSingle_pipe_statement" ):
                listener.enterSingle_pipe_statement(self)

        def exitRule(self, listener:ParseTreeListener):
            if hasattr( listener, "exitSingle_pipe_statement" ):
                listener.exitSingle_pipe_statement(self)




    def single_pipe_statement(self):

        localctx = ExprParser.Single_pipe_statementContext(self, self._ctx, self.state)
        self.enterRule(localctx, 6, self.RULE_single_pipe_statement)
        self._la = 0 # Token type
        try:
            self.enterOuterAlt(localctx, 1)
            self.state = 41
            self.match(ExprParser.NAME)
            self.state = 44
            self._errHandler.sync(self)
            _la = self._input.LA(1)
            while True:
                self.state = 42
                self.match(ExprParser.SPIPE)
                self.state = 43
                self.function_call()
                self.state = 46
                self._errHandler.sync(self)
                _la = self._input.LA(1)
                if not (_la==16):
                    break

        except RecognitionException as re:
            localctx.exception = re
            self._errHandler.reportError(self, re)
            self._errHandler.recover(self, re)
        finally:
            self.exitRule()
        return localctx


    class Function_callContext(ParserRuleContext):
        __slots__ = 'parser'

        def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
            super().__init__(parent, invokingState)
```

```
            self.parser = parser

        def NAME(self):
            return self.getToken(ExprParser.NAME, 0)

        def args(self):
            return self.getTypedRuleContext(ExprParser.ArgsContext,0)


        def getRuleIndex(self):
            return ExprParser.RULE_function_call

        def enterRule(self, listener:ParseTreeListener):
            if hasattr( listener, "enterFunction_call" ):
                listener.enterFunction_call(self)

        def exitRule(self, listener:ParseTreeListener):
            if hasattr( listener, "exitFunction_call" ):
                listener.exitFunction_call(self)




    def function_call(self):

        localctx = ExprParser.Function_callContext(self, self._ctx, self.state)
        self.enterRule(localctx, 8, self.RULE_function_call)
        try:
            self.state = 56
            self._errHandler.sync(self)
            la_ = self._interp.adaptivePredict(self._input,4,self._ctx)
            if la_ == 1:
                self.enterOuterAlt(localctx, 1)
                self.state = 48
                self.match(ExprParser.NAME)
                self.state = 49
                self.match(ExprParser.T__2)
                self.state = 50
                self.args()
                self.state = 51
                self.match(ExprParser.T__3)
                pass

            elif la_ == 2:
                self.enterOuterAlt(localctx, 2)
                self.state = 53
                self.match(ExprParser.NAME)
                self.state = 54
                self.match(ExprParser.T__2)
                self.state = 55
                self.match(ExprParser.T__3)
                pass


        except RecognitionException as re:
            localctx.exception = re
            self._errHandler.reportError(self, re)
            self._errHandler.recover(self, re)
        finally:
            self.exitRule()
        return localctx
```

```
424
425
426         class AssignmentContext(ParserRuleContext):
427             __slots__ = 'parser'
428
429             def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
430                 super().__init__(parent, invokingState)
431                 self.parser = parser
432
433             def NAME(self):
434                 return self.getToken(ExprParser.NAME, 0)
435
436             def value(self):
437                 return self.getTypedRuleContext(ExprParser.ValueContext,0)
438
439
440             def array(self):
441                 return self.getTypedRuleContext(ExprParser.ArrayContext,0)
442
443
444             def getRuleIndex(self):
445                 return ExprParser.RULE_assignment
446
447             def enterRule(self, listener:ParseTreeListener):
448                 if hasattr( listener, "enterAssignment" ):
449                     listener.enterAssignment(self)
450
451             def exitRule(self, listener:ParseTreeListener):
452                 if hasattr( listener, "exitAssignment" ):
453                     listener.exitAssignment(self)
454
455
456
457
458         def assignment(self):
459
460             localctx = ExprParser.AssignmentContext(self, self._ctx, self.state)
461             self.enterRule(localctx, 10, self.RULE_assignment)
462             try:
463                 self.enterOuterAlt(localctx, 1)
464                 self.state = 58
465                 self.match(ExprParser.NAME)
466                 self.state = 59
467                 self.match(ExprParser.T__4)
468                 self.state = 62
469                 self._errHandler.sync(self)
470                 token = self._input.LA(1)
471                 if token in [9, 10, 11, 12, 14, 15]:
472                     self.state = 60
473                     self.value()
474                     pass
475                 elif token in [6]:
476                     self.state = 61
477                     self.array()
478                     pass
479                 else:
480                     raise NoViableAltException(self)
481
482             except RecognitionException as re:
483                 localctx.exception = re
484                 self._errHandler.reportError(self, re)
```

```
            self._errHandler.recover(self, re)
        finally:
            self.exitRule()
        return localctx


    class ArrayContext(ParserRuleContext):
        __slots__ = 'parser'

        def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
            super().__init__(parent, invokingState)
            self.parser = parser

        def value(self, i:int=None):
            if i is None:
                return self.getTypedRuleContexts(ExprParser.ValueContext)
            else:
                return self.getTypedRuleContext(ExprParser.ValueContext,i)


        def array(self, i:int=None):
            if i is None:
                return self.getTypedRuleContexts(ExprParser.ArrayContext)
            else:
                return self.getTypedRuleContext(ExprParser.ArrayContext,i)


        def getRuleIndex(self):
            return ExprParser.RULE_array

        def enterRule(self, listener:ParseTreeListener):
            if hasattr( listener, "enterArray" ):
                listener.enterArray(self)

        def exitRule(self, listener:ParseTreeListener):
            if hasattr( listener, "exitArray" ):
                listener.exitArray(self)




    def array(self):

        localctx = ExprParser.ArrayContext(self, self._ctx, self.state)
        self.enterRule(localctx, 12, self.RULE_array)
        self._la = 0 # Token type
        try:
            self.state = 83
            self._errHandler.sync(self)
            la_ = self._interp.adaptivePredict(self._input,9,self._ctx)
            if la_ == 1:
                self.enterOuterAlt(localctx, 1)
                self.state = 64
                self.match(ExprParser.T__5)
                self.state = 65
                self.match(ExprParser.T__6)
                pass

            elif la_ == 2:
                self.enterOuterAlt(localctx, 2)
                self.state = 66
```

```
                    self.match(ExprParser.T__5)
                    self.state = 69
                    self._errHandler.sync(self)
                    token = self._input.LA(1)
                    if token in [9, 10, 11, 12, 14, 15]:
                        self.state = 67
                        self.value()
                        pass
                    elif token in [6]:
                        self.state = 68
                        self.array()
                        pass
                    else:
                        raise NoViableAltException(self)

                    self.state = 78
                    self._errHandler.sync(self)
                    _la = self._input.LA(1)
                    while _la==8:
                        self.state = 71
                        self.match(ExprParser.T__7)
                        self.state = 74
                        self._errHandler.sync(self)
                        token = self._input.LA(1)
                        if token in [9, 10, 11, 12, 14, 15]:
                            self.state = 72
                            self.value()
                            pass
                        elif token in [6]:
                            self.state = 73
                            self.array()
                            pass
                        else:
                            raise NoViableAltException(self)

                        self.state = 80
                        self._errHandler.sync(self)
                        _la = self._input.LA(1)

                    self.state = 81
                    self.match(ExprParser.T__6)
                    pass


            except RecognitionException as re:
                localctx.exception = re
                self._errHandler.reportError(self, re)
                self._errHandler.recover(self, re)
            finally:
                self.exitRule()
            return localctx


    class ValueContext(ParserRuleContext):
        __slots__ = 'parser'

        def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
            super().__init__(parent, invokingState)
            self.parser = parser

        def INT(self):
```

```
            return self.getToken(ExprParser.INT, 0)

        def FLOAT(self):
            return self.getToken(ExprParser.FLOAT, 0)

        def STRING(self):
            return self.getToken(ExprParser.STRING, 0)

        def BOOL(self):
            return self.getToken(ExprParser.BOOL, 0)

        def CHAR(self):
            return self.getToken(ExprParser.CHAR, 0)

        def NAME(self):
            return self.getToken(ExprParser.NAME, 0)

        def getRuleIndex(self):
            return ExprParser.RULE_value

        def enterRule(self, listener:ParseTreeListener):
            if hasattr( listener, "enterValue" ):
                listener.enterValue(self)

        def exitRule(self, listener:ParseTreeListener):
            if hasattr( listener, "exitValue" ):
                listener.exitValue(self)



    def value(self):

        localctx = ExprParser.ValueContext(self, self._ctx, self.state)
        self.enterRule(localctx, 14, self.RULE_value)
        self._la = 0 # Token type
        try:
            self.enterOuterAlt(localctx, 1)
            self.state = 85
            _la = self._input.LA(1)
            if not((((_la) & ~0x3f) == 0 and ((1 << _la) & 56832) != 0)):
                self._errHandler.recoverInline(self)
            else:
                self._errHandler.reportMatch(self)
                self.consume()
        except RecognitionException as re:
            localctx.exception = re
            self._errHandler.reportError(self, re)
            self._errHandler.recover(self, re)
        finally:
            self.exitRule()
        return localctx


    class ArgsContext(ParserRuleContext):
        __slots__ = 'parser'

        def __init__(self, parser, parent:ParserRuleContext=None, invokingState:int=-1):
            super().__init__(parent, invokingState)
            self.parser = parser

```

```
668             def value(self, i:int=None):
669                 if i is None:
670                     return self.getTypedRuleContexts(ExprParser.ValueContext)
671                 else:
672                     return self.getTypedRuleContext(ExprParser.ValueContext,i)
673 
674 
675             def array(self, i:int=None):
676                 if i is None:
677                     return self.getTypedRuleContexts(ExprParser.ArrayContext)
678                 else:
679                     return self.getTypedRuleContext(ExprParser.ArrayContext,i)
680 
681 
682             def getRuleIndex(self):
683                 return ExprParser.RULE_args
684 
685             def enterRule(self, listener:ParseTreeListener):
686                 if hasattr( listener, "enterArgs" ):
687                     listener.enterArgs(self)
688 
689             def exitRule(self, listener:ParseTreeListener):
690                 if hasattr( listener, "exitArgs" ):
691                     listener.exitArgs(self)
692 
693 
694 
695 
696         def args(self):
697 
698             localctx = ExprParser.ArgsContext(self, self._ctx, self.state)
699             self.enterRule(localctx, 16, self.RULE_args)
700             self._la = 0 # Token type
701             try:
702                 self.enterOuterAlt(localctx, 1)
703                 self.state = 89
704                 self._errHandler.sync(self)
705                 token = self._input.LA(1)
706                 if token in [9, 10, 11, 12, 14, 15]:
707                     self.state = 87
708                     self.value()
709                     pass
710                 elif token in [6]:
711                     self.state = 88
712                     self.array()
713                     pass
714                 else:
715                     raise NoViableAltException(self)
716 
717                 self.state = 98
718                 self._errHandler.sync(self)
719                 _la = self._input.LA(1)
720                 while _la==8:
721                     self.state = 91
722                     self.match(ExprParser.T__7)
723                     self.state = 94
724                     self._errHandler.sync(self)
725                     token = self._input.LA(1)
726                     if token in [9, 10, 11, 12, 14, 15]:
727                         self.state = 92
728                         self.value()
```

```
                    pass
                elif token in [6]:
                    self.state = 93
                    self.array()
                    pass
                else:
                    raise NoViableAltException(self)

                self.state = 100
                self._errHandler.sync(self)
                _la = self._input.LA(1)

        except RecognitionException as re:
            localctx.exception = re
            self._errHandler.reportError(self, re)
            self._errHandler.recover(self, re)
        finally:
            self.exitRule()
        return localctx

```

Expr.g4

```
grammar Expr;

prog: imports (assignment single_pipe_statement*)+;

imports: import_statement*;

import_statement: 'from' NAME 'import' NAME;

single_pipe_statement: NAME (SPIPE function_call)+;

function_call: NAME '(' args ')' | NAME '(' ')';

assignment: NAME '=' (value | array);

array: '[' ']' | '[' (value | array) (',' (value | array))* ']';

value: INT | FLOAT | STRING | BOOL | CHAR | NAME;

args: (value | array) (',' (value | array))*;

INT: [0-9]+ | [-][0-9]+;
FLOAT: [0-9]+[.][0-9]+ | [-][0-9]+[.][0-9]+;
CHAR: ["][a-zA-Z0-9]["];
STRING: ["]~["]*["];
WS: [ \t\r\n]+ -> skip;
NAME: [a-zA-Z][a-zA-Z0-9_]*;
BOOL: 'true' | 'false';
SPIPE: [|][>];
```

Expr.interp

```
        token literal names:
    null
    'from'
    'import'
    '('
    ')'
    '='
    '['
    ']'
    ','
    null
    null
    null
    null
    null
    null
    null
    null

    token symbolic names:
    null
    null
    null
    null
    null
    null
    null
    null
    null
    INT
    FLOAT
    CHAR
    STRING
    WS
    NAME
    BOOL
    SPIPE

    rule names:
    prog
    imports
    import_statement
    single_pipe_statement
    function_call
    assignment
    array
    value
    args


    atn:
```

```
52  [4, 1, 16, 102, 2, 0, 7, 0, 2, 1, 7, 1, 2, 2, 7, 2, 2, 3, 7, 3, 2, 4, 7, 4, 2, 5, 7, 5, 2, 6,
    ↪ 7, 6, 2, 7, 7, 7, 2, 8, 7, 8, 1, 0, 1, 0, 1, 0, 5, 0, 22, 8, 0, 10, 0, 12, 0, 25, 9, 0,
    ↪ 4, 0, 27, 8, 0, 11, 0, 12, 0, 28, 1, 1, 5, 1, 32, 8, 1, 10, 1, 12, 1, 35, 9, 1, 1, 2, 1,
    ↪ 2, 1, 2, 1, 2, 1, 2, 1, 3, 1, 3, 1, 3, 4, 3, 45, 8, 3, 11, 3, 12, 3, 46, 1, 4, 1, 4, 1,
    ↪ 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 3, 4, 57, 8, 4, 1, 5, 1, 5, 1, 5, 1, 5, 3, 5, 63, 8, 5,
    ↪ 1, 6, 1, 6, 1, 6, 1, 6, 1, 6, 3, 6, 70, 8, 6, 1, 6, 1, 6, 1, 6, 3, 6, 75, 8, 6, 5, 6, 77,
    ↪ 8, 6, 10, 6, 12, 6, 80, 9, 6, 1, 6, 1, 6, 3, 6, 84, 8, 6, 1, 7, 1, 7, 1, 8, 1, 8, 3, 8,
    ↪ 90, 8, 8, 1, 8, 1, 8, 1, 8, 3, 8, 95, 8, 8, 5, 8, 97, 8, 8, 10, 8, 12, 8, 100, 9, 8, 1,
    ↪ 8, 0, 0, 9, 0, 2, 4, 6, 8, 10, 12, 14, 16, 0, 1, 2, 0, 9, 12, 14, 15, 105, 0, 18, 1, 0,
    ↪ 0, 0, 2, 33, 1, 0, 0, 0, 4, 36, 1, 0, 0, 0, 6, 41, 1, 0, 0, 0, 8, 56, 1, 0, 0, 0, 10, 58,
    ↪ 1, 0, 0, 0, 12, 83, 1, 0, 0, 0, 14, 85, 1, 0, 0, 0, 16, 89, 1, 0, 0, 0, 18, 26, 3, 2, 1,
    ↪ 0, 19, 23, 3, 10, 5, 0, 20, 22, 3, 6, 3, 0, 21, 20, 1, 0, 0, 0, 22, 25, 1, 0, 0, 0, 23,
    ↪ 21, 1, 0, 0, 0, 23, 24, 1, 0, 0, 0, 24, 27, 1, 0, 0, 0, 25, 23, 1, 0, 0, 0, 26, 19, 1, 0,
    ↪ 0, 0, 27, 28, 1, 0, 0, 0, 28, 26, 1, 0, 0, 0, 28, 29, 1, 0, 0, 0, 29, 1, 1, 0, 0, 0, 30,
    ↪ 32, 3, 4, 2, 0, 31, 30, 1, 0, 0, 0, 32, 35, 1, 0, 0, 0, 33, 31, 1, 0, 0, 0, 33, 34, 1, 0,
    ↪ 0, 0, 34, 3, 1, 0, 0, 0, 35, 33, 1, 0, 0, 0, 36, 37, 5, 1, 0, 0, 37, 38, 5, 14, 0, 0, 38,
    ↪ 39, 5, 2, 0, 0, 39, 40, 5, 14, 0, 0, 40, 5, 1, 0, 0, 0, 41, 44, 5, 14, 0, 0, 42, 43, 5,
    ↪ 16, 0, 0, 43, 45, 3, 8, 4, 0, 44, 42, 1, 0, 0, 0, 45, 46, 1, 0, 0, 0, 46, 44, 1, 0, 0, 0,
    ↪ 46, 47, 1, 0, 0, 0, 47, 7, 1, 0, 0, 0, 48, 49, 5, 14, 0, 0, 49, 50, 5, 3, 0, 0, 50, 51,
    ↪ 3, 16, 8, 0, 51, 52, 5, 4, 0, 0, 52, 57, 1, 0, 0, 0, 53, 54, 5, 14, 0, 0, 54, 55, 5, 3,
    ↪ 0, 0, 55, 57, 5, 4, 0, 0, 56, 48, 1, 0, 0, 0, 56, 53, 1, 0, 0, 0, 57, 9, 1, 0, 0, 0, 58,
    ↪ 59, 5, 14, 0, 0, 59, 62, 5, 5, 0, 0, 60, 63, 3, 14, 7, 0, 61, 63, 3, 12, 6, 0, 62, 60, 1,
    ↪ 0, 0, 0, 62, 61, 1, 0, 0, 0, 63, 11, 1, 0, 0, 0, 64, 65, 5, 6, 0, 0, 65, 84, 5, 7, 0, 0,
    ↪ 66, 69, 5, 6, 0, 0, 67, 70, 3, 14, 7, 0, 68, 70, 3, 12, 6, 0, 69, 67, 1, 0, 0, 0, 69, 68,
    ↪ 1, 0, 0, 0, 70, 78, 1, 0, 0, 0, 71, 74, 5, 8, 0, 0, 72, 75, 3, 14, 7, 0, 73, 75, 3, 12,
    ↪ 6, 0, 74, 72, 1, 0, 0, 0, 74, 73, 1, 0, 0, 0, 75, 77, 1, 0, 0, 0, 76, 71, 1, 0, 0, 0, 77,
    ↪ 80, 1, 0, 0, 0, 78, 76, 1, 0, 0, 0, 78, 79, 1, 0, 0, 0, 79, 81, 1, 0, 0, 0, 80, 78, 1, 0,
    ↪ 0, 0, 81, 82, 5, 7, 0, 0, 82, 84, 1, 0, 0, 0, 83, 64, 1, 0, 0, 0, 83, 66, 1, 0, 0, 0, 84,
    ↪ 13, 1, 0, 0, 0, 85, 86, 7, 0, 0, 0, 86, 15, 1, 0, 0, 0, 87, 90, 3, 14, 7, 0, 88, 90, 3,
    ↪ 12, 6, 0, 89, 87, 1, 0, 0, 0, 89, 88, 1, 0, 0, 0, 90, 98, 1, 0, 0, 0, 91, 94, 5, 8, 0, 0,
    ↪ 92, 95, 3, 14, 7, 0, 93, 95, 3, 12, 6, 0, 94, 92, 1, 0, 0, 0, 94, 93, 1, 0, 0, 0, 95, 97,
    ↪ 1, 0, 0, 0, 96, 91, 1, 0, 0, 0, 97, 100, 1, 0, 0, 0, 98, 96, 1, 0, 0, 0, 98, 99, 1, 0, 0,
    ↪ 0, 99, 17, 1, 0, 0, 0, 100, 98, 1, 0, 0, 0, 13, 23, 28, 33, 46, 56, 62, 69, 74, 78, 83,
    ↪ 89, 94, 98]
```

Expr.tokens

```
T__0=1
T__1=2
T__2=3
T__3=4
T__4=5
T__5=6
T__6=7
T__7=8
INT=9
FLOAT=10
CHAR=11
STRING=12
WS=13
NAME=14
BOOL=15
SPIPE=16
'from'=1
'import'=2
'('=3
')'=4
'='=5
'['=6
']'=7
','=8

```

ExprLexer.interp

```
token literal names:
null
'from'
'import'
'('
')'
'='
'['
']'
','
null
null
null
null
null
null
null
null

token symbolic names:
null
null
null
null
null
null
null
null
null
INT
FLOAT
CHAR
STRING
WS
NAME
BOOL
SPIPE

rule names:
T__0
T__1
T__2
T__3
T__4
T__5
T__6
T__7
INT
FLOAT
CHAR
STRING
WS
NAME
BOOL
SPIPE

channel names:
DEFAULT_TOKEN_CHANNEL
```

```
    HIDDEN

    mode names:
    DEFAULT_MODE

    atn:
    [4, 0, 16, 136, 6, -1, 2, 0, 7, 0, 2, 1, 7, 1, 2, 2, 7, 2, 2, 3, 7, 3, 2, 4, 7, 4, 2, 5, 7,
     ↪ 5, 2, 6, 7, 6, 2, 7, 7, 7, 2, 8, 7, 8, 2, 9, 7, 9, 2, 10, 7, 10, 2, 11, 7, 11, 2, 12, 7,
     ↪ 12, 2, 13, 7, 13, 2, 14, 7, 14, 2, 15, 7, 15, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1,
     ↪ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 3, 1, 3, 1, 4, 1, 4, 1, 5, 1, 5, 1, 6, 1, 6,
     ↪ 1, 7, 1, 7, 1, 8, 4, 8, 59, 8, 8, 11, 8, 12, 8, 60, 1, 8, 1, 8, 4, 8, 65, 8, 8, 11, 8,
     ↪ 12, 8, 66, 3, 8, 69, 8, 8, 1, 9, 4, 9, 72, 8, 9, 11, 9, 12, 9, 73, 1, 9, 1, 9, 4, 9, 78,
     ↪ 8, 9, 11, 9, 12, 9, 79, 1, 9, 1, 9, 4, 9, 84, 8, 9, 11, 9, 12, 9, 85, 1, 9, 1, 9, 4, 9,
     ↪ 90, 8, 9, 11, 9, 12, 9, 91, 3, 9, 94, 8, 9, 1, 10, 1, 10, 1, 10, 1, 10, 1, 11, 1, 11, 5,
     ↪ 11, 102, 8, 11, 10, 11, 12, 11, 105, 9, 11, 1, 11, 1, 11, 1, 12, 4, 12, 110, 8, 12, 11,
     ↪ 12, 12, 12, 111, 1, 12, 1, 12, 1, 13, 1, 13, 5, 13, 118, 8, 13, 10, 13, 12, 13, 121, 9,
     ↪ 13, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 1, 14, 3, 14, 132, 8, 14, 1,
     ↪ 15, 1, 15, 1, 15, 0, 0, 16, 1, 1, 3, 2, 5, 3, 7, 4, 9, 5, 11, 6, 13, 7, 15, 8, 17, 9, 19,
     ↪ 10, 21, 11, 23, 12, 25, 13, 27, 14, 29, 15, 31, 16, 1, 0, 10, 1, 0, 48, 57, 1, 0, 45, 45,
     ↪ 1, 0, 46, 46, 1, 0, 34, 34, 3, 0, 48, 57, 65, 90, 97, 122, 3, 0, 9, 10, 13, 13, 32, 32,
     ↪ 2, 0, 65, 90, 97, 122, 4, 0, 48, 57, 65, 90, 95, 95, 97, 122, 1, 0, 124, 124, 1, 0, 62,
     ↪ 62, 147, 0, 1, 1, 0, 0, 0, 0, 3, 1, 0, 0, 0, 0, 5, 1, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 9, 1,
     ↪ 0, 0, 0, 0, 11, 1, 0, 0, 0, 0, 13, 1, 0, 0, 0, 0, 15, 1, 0, 0, 0, 0, 17, 1, 0, 0, 0, 0,
     ↪ 19, 1, 0, 0, 0, 0, 21, 1, 0, 0, 0, 0, 23, 1, 0, 0, 0, 0, 25, 1, 0, 0, 0, 0, 27, 1, 0, 0,
     ↪ 0, 0, 29, 1, 0, 0, 0, 0, 31, 1, 0, 0, 0, 1, 33, 1, 0, 0, 0, 3, 38, 1, 0, 0, 0, 5, 45, 1,
     ↪ 0, 0, 0, 7, 47, 1, 0, 0, 0, 9, 49, 1, 0, 0, 0, 11, 51, 1, 0, 0, 0, 13, 53, 1, 0, 0, 0,
     ↪ 15, 55, 1, 0, 0, 0, 17, 68, 1, 0, 0, 0, 19, 93, 1, 0, 0, 0, 21, 95, 1, 0, 0, 0, 23, 99,
     ↪ 1, 0, 0, 0, 25, 109, 1, 0, 0, 0, 27, 115, 1, 0, 0, 0, 29, 131, 1, 0, 0, 0, 31, 133, 1, 0,
     ↪ 0, 0, 33, 34, 5, 102, 0, 0, 34, 35, 5, 114, 0, 0, 35, 36, 5, 111, 0, 0, 36, 37, 5, 109,
     ↪ 0, 0, 37, 2, 1, 0, 0, 0, 38, 39, 5, 105, 0, 0, 39, 40, 5, 109, 0, 0, 40, 41, 5, 112, 0,
     ↪ 0, 41, 42, 5, 111, 0, 0, 42, 43, 5, 114, 0, 0, 43, 44, 5, 116, 0, 0, 44, 4, 1, 0, 0, 0,
     ↪ 45, 46, 5, 40, 0, 0, 46, 6, 1, 0, 0, 0, 47, 48, 5, 41, 0, 0, 48, 8, 1, 0, 0, 0, 49, 50,
     ↪ 5, 61, 0, 0, 50, 10, 1, 0, 0, 0, 51, 52, 5, 91, 0, 0, 52, 12, 1, 0, 0, 0, 53, 54, 5, 93,
     ↪ 0, 0, 54, 14, 1, 0, 0, 0, 55, 56, 5, 44, 0, 0, 56, 16, 1, 0, 0, 0, 57, 59, 7, 0, 0, 0,
     ↪ 58, 57, 1, 0, 0, 0, 59, 60, 1, 0, 0, 0, 60, 58, 1, 0, 0, 0, 60, 61, 1, 0, 0, 0, 61, 69,
     ↪ 1, 0, 0, 0, 62, 64, 7, 1, 0, 0, 63, 65, 7, 0, 0, 0, 64, 63, 1, 0, 0, 0, 65, 66, 1, 0, 0,
     ↪ 0, 66, 64, 1, 0, 0, 0, 66, 67, 1, 0, 0, 0, 67, 69, 1, 0, 0, 0, 68, 58, 1, 0, 0, 0, 68,
     ↪ 62, 1, 0, 0, 0, 69, 18, 1, 0, 0, 0, 70, 72, 7, 0, 0, 0, 71, 70, 1, 0, 0, 0, 72, 73, 1, 0,
     ↪ 0, 0, 73, 71, 1, 0, 0, 0, 73, 74, 1, 0, 0, 0, 74, 75, 1, 0, 0, 0, 75, 77, 7, 2, 0, 0, 76,
     ↪ 78, 7, 0, 0, 0, 77, 76, 1, 0, 0, 0, 78, 79, 1, 0, 0, 0, 79, 77, 1, 0, 0, 0, 79, 80, 1, 0,
     ↪ 0, 0, 80, 94, 1, 0, 0, 0, 81, 83, 7, 1, 0, 0, 82, 84, 7, 0, 0, 0, 83, 82, 1, 0, 0, 0, 84,
     ↪ 85, 1, 0, 0, 0, 85, 83, 1, 0, 0, 0, 85, 86, 1, 0, 0, 0, 86, 87, 1, 0, 0, 0, 87, 89, 7, 2,
     ↪ 0, 0, 88, 90, 7, 0, 0, 0, 89, 88, 1, 0, 0, 0, 90, 91, 1, 0, 0, 0, 91, 89, 1, 0, 0, 0, 91,
     ↪ 92, 1, 0, 0, 0, 92, 94, 1, 0, 0, 0, 93, 71, 1, 0, 0, 0, 93, 81, 1, 0, 0, 0, 94, 20, 1, 0,
     ↪ 0, 0, 95, 96, 7, 3, 0, 0, 96, 97, 7, 4, 0, 0, 97, 98, 7, 3, 0, 0, 98, 22, 1, 0, 0, 0, 99,
     ↪ 103, 7, 3, 0, 0, 100, 102, 8, 3, 0, 0, 101, 100, 1, 0, 0, 0, 102, 105, 1, 0, 0, 0, 103,
     ↪ 101, 1, 0, 0, 0, 103, 104, 1, 0, 0, 0, 104, 106, 1, 0, 0, 0, 105, 103, 1, 0, 0, 0, 106,
     ↪ 107, 7, 3, 0, 0, 107, 24, 1, 0, 0, 0, 108, 110, 7, 5, 0, 0, 109, 108, 1, 0, 0, 0, 110,
     ↪ 111, 1, 0, 0, 0, 111, 109, 1, 0, 0, 0, 111, 112, 1, 0, 0, 0, 112, 113, 1, 0, 0, 0, 113,
     ↪ 114, 6, 12, 0, 0, 114, 26, 1, 0, 0, 0, 115, 119, 7, 6, 0, 0, 116, 118, 7, 7, 0, 0, 117,
     ↪ 116, 1, 0, 0, 0, 118, 121, 1, 0, 0, 0, 119, 117, 1, 0, 0, 0, 119, 120, 1, 0, 0, 0, 120,
     ↪ 28, 1, 0, 0, 0, 121, 119, 1, 0, 0, 0, 122, 123, 5, 116, 0, 0, 123, 124, 5, 114, 0, 0,
     ↪ 124, 125, 5, 117, 0, 0, 125, 132, 5, 101, 0, 0, 126, 127, 5, 102, 0, 0, 127, 128, 5, 97,
     ↪ 0, 0, 128, 129, 5, 108, 0, 0, 129, 130, 5, 115, 0, 0, 130, 132, 5, 101, 0, 0, 131, 122,
     ↪ 1, 0, 0, 0, 131, 126, 1, 0, 0, 0, 132, 30, 1, 0, 0, 0, 133, 134, 7, 8, 0, 0, 134, 135, 7,
     ↪ 9, 0, 0, 135, 32, 1, 0, 0, 0, 13, 0, 60, 66, 68, 73, 79, 85, 91, 93, 103, 111, 119, 131,
     ↪ 1, 6, 0, 0]
```

ExprLexer.tokens

```
T__0=1
T__1=2
T__2=3
T__3=4
T__4=5
T__5=6
T__6=7
T__7=8
INT=9
FLOAT=10
CHAR=11
STRING=12
WS=13
NAME=14
BOOL=15
SPIPE=16
'from'=1
'import'=2
'('=3
')'=4
'='=5
'['=6
']'=7
','=8

```