# Question 2

Note: I did not setup the GitHub for version control till after I wrote the updated version of the code. So, I am unable to provide snippets of the original code. However, I was able to get a close enough version to the original from a workshop I had done. Any references to “the original” are in references to that code.

Contents

[Question 2 1](#_Toc196327317)

[**Grammar** 2](#_Toc196327318)

[***Parser Rules*** 2](#_Toc196327319)

[***Lexer Rules*** 3](#_Toc196327320)

[***Changes Applied*** 4](#_Toc196327321)

[**Python Code** 6](#_Toc196327322)

[***File Collection Changes*** 6](#_Toc196327323)

[***FSObject Changes*** 6](#_Toc196327324)

[***Selector Changes*** 6](#_Toc196327325)

[***Main and FspowVisitor*** 10](#_Toc196327326)

[Appendix 14](#_Toc196327327)

## **Grammar**

The fspow.g4 file followed a closed example of the workshop fspow.g4 file. The grammar had strict rules for the parse rules and lexer rules.

A screenshot of a computer program

AI-generated content may be incorrect.

### ***Parser Rules***

The parser rules describe the structure of valid programs in the fspow language. The entry point is the prog rule, which represents a program consisting of one or more statements (stats) followed by the end of the file (EOF). Each statement can take one of four forms: an assignment, applying a selector to a file collection, listing files in a collection, or displaying a message.

Here are the different rules used:

* Assignments (assignment): These allow associating an identifier (ID) with an expression. The expression can represent a file collection, a selector, or another identifier.
* Expressions (expression): Expressions define the creation of file collections (fcCreation), selectors (selCreation), or references to existing identifiers.
* File Collection Creation (fcCreation): This rule specifies how to create a FileCollection by providing a root directory (rootSpecifier), which is a string.
* Selector Creation (selCreation): This rule defines how to create a Selector using a filter (selfilter), which specifies criteria for selecting files.
* Filters (selfilter): Filters allow users to define selection criteria based on file attributes like name, size, or date. Filters can also be combined using logical operations like intersect (AND) or negated using not. Parentheses can group filters for clarity and precedence.
* File Collection Operations: The fcApplySelector rule applies a selector to a file collection, while the fcList rule lists the files in a collection.
* Messages (message): This rule enables displaying a message, which is a string enclosed in quotes.

### ***Lexer Rules***

The lexer rules define the tokens that make up the language. These tokens are the smallest building blocks used by the parser rules.

* Identifiers (ID): These are alphanumeric strings that start with a letter and can include underscores. They represent variable names or identifiers in the language.
* Strings (STRING): Strings are enclosed in double quotes and can contain any characters except double quotes themselves.
* Whitespace (WS): Whitespace characters (spaces, tabs, newlines) are ignored by the lexer, as indicated by the -> skip action.
* Comments (COMMENT): Single-line comments start with  and continue until the end of the line. These are also ignored by the lexer.

### ***Changes Applied***

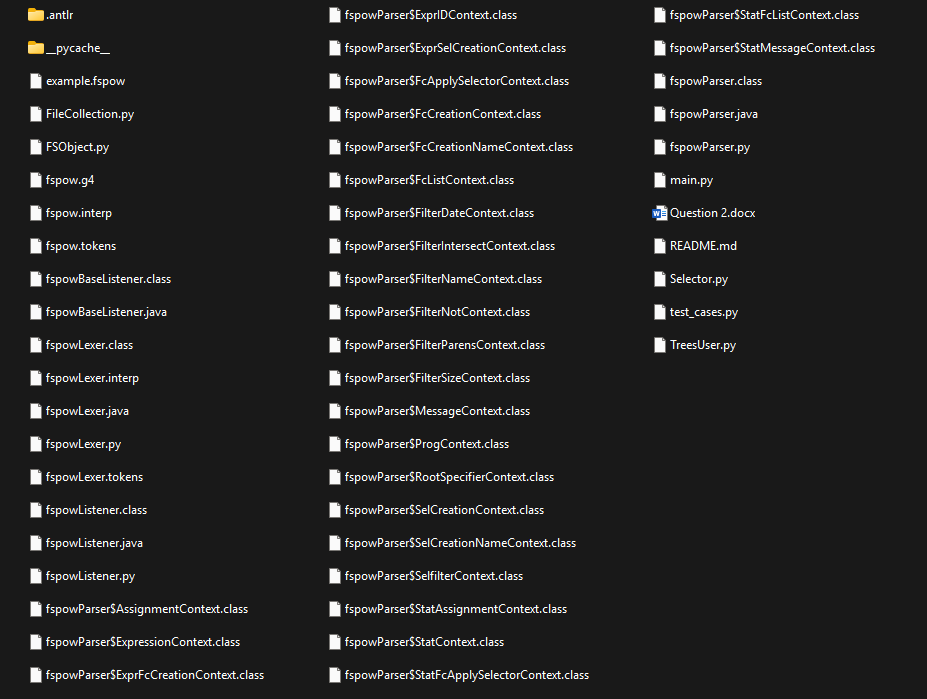
The changes applied to the grammar file were implemented to better focus the grammar on the aim of the question. For example, the filter was expanded to account for name, size, date, interaction, not, and parents.

The selector was updated to a one liner which calls the literal string Selector along with brackets around the parse rule for the filter. This simplified the structure of the grammar file overall as I did not need to manage different values for defining values.

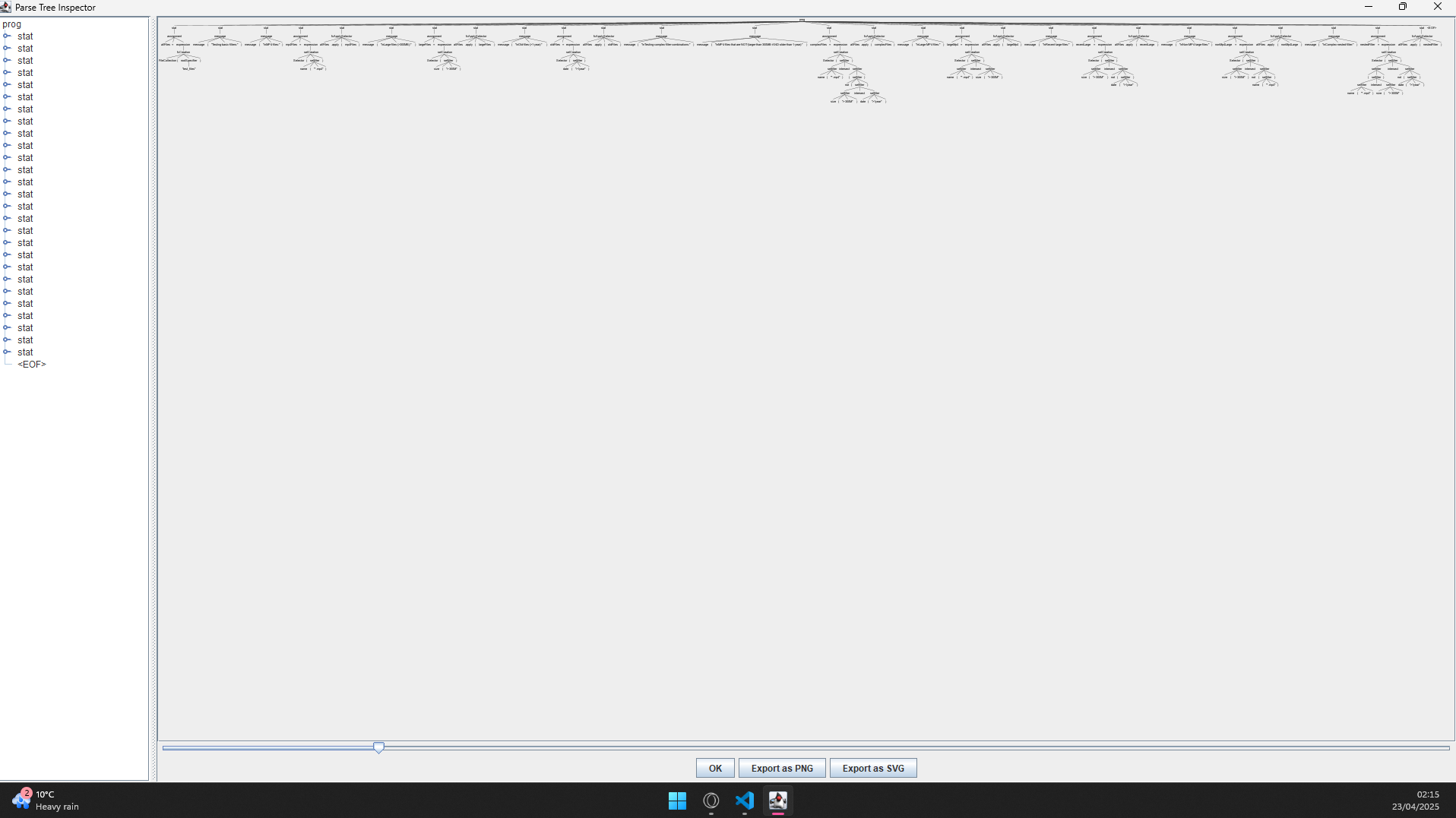
The regex was updated to adapt to different situations.

* ID: it accounts for underscores
* STRING: it accounts for if a single quote is included, and if so, it would prematurely terminate it.
* COMMENT: it skips over the line instead of channelling.

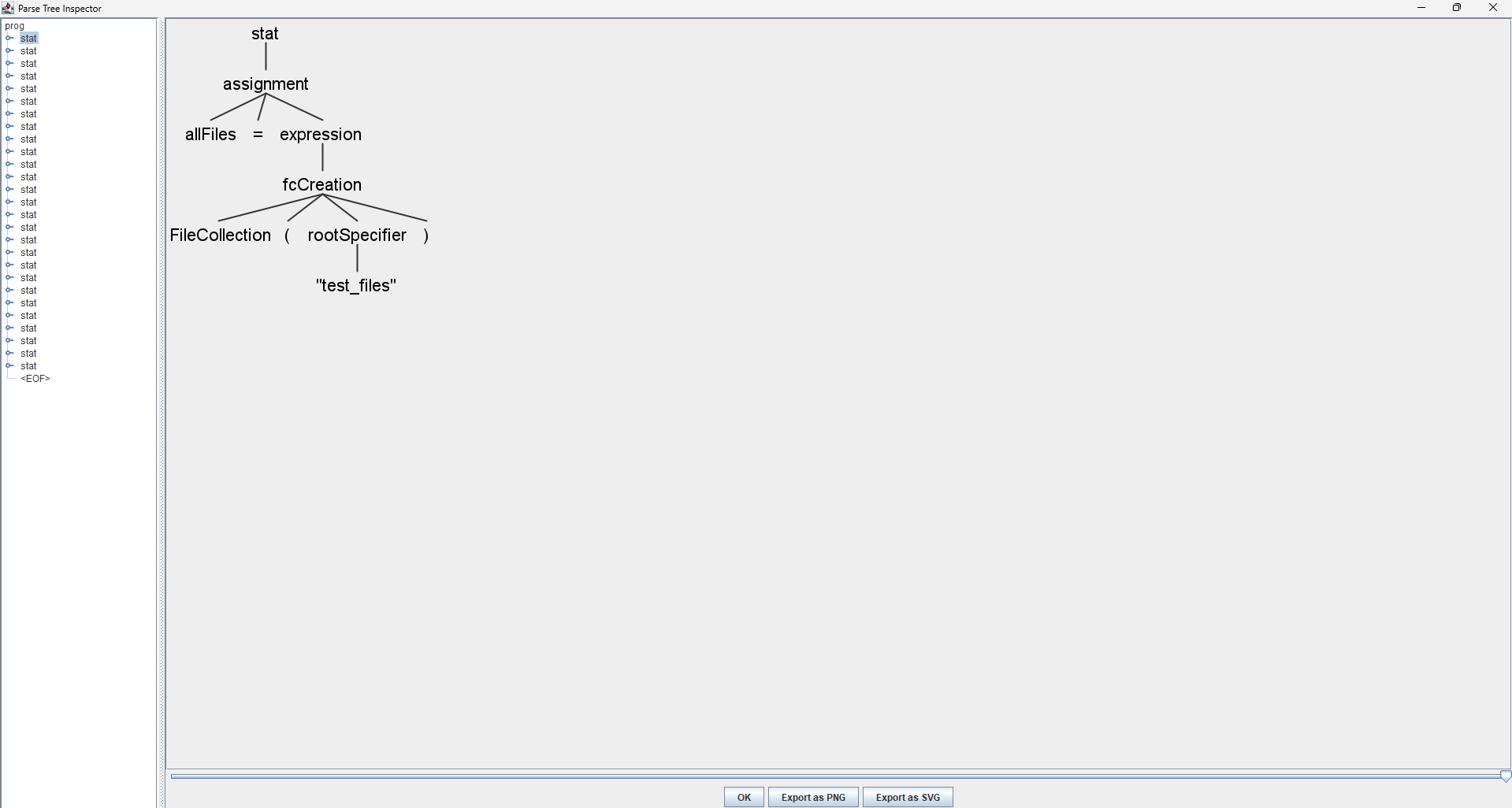
Some other changes include the modification to list and apply selectors. Like how the selector was turned into a single line, the list and apply selectors were adjusted so they work as a one liner for easier maintenance and better readability of the code.

The codebase structure when the antlr4 and javac commands were run: 

The tree view when the grun command was run showing the trees:



Isolated Tree:



## **Python Code**

When beginning the question, I originally chose to follow the examples shown during workshops and lectures to get files constructed. I reused files like FileCollection and FSObject and modified the code to better suit my use case. I removed the need of a file iterator as I was able to condense the iterator into a “\_scan\_directory” function which uses a for loop to traverse through the file system.

### ***File Collection Changes***

A screen shot of a computer program

AI-generated content may be incorrect.

For each entry returned by os.scandir, the code creates an FSObject instance (fs\_object = FSObject(entry)) to encapsulate the metadata of the file or directory. These FSObject instances are then appended to the files list of the FileCollection object.

The apply function was also modified from the original for several different reasons:

1. Original had mutable behaviour
   1. The original method used during workshops modified the original FileCollection place. It directly alters the content list by removing items that do not pass the filter.
   2. The updated method uses immutable behaviours and does not modify the original FileCollection. It instead creates a new FileCollection object (results) and populates it with the filtered files.
2. Original While Loop
   1. The original method used a while loop to iterate through the content list. If an item does not pass the filter (filter.testFSObject), it is removed from the list using pop.
   2. The update method uses list comprehension to filter the files list. The selector function is applied to each FSObject in self.files, and only those that pass the filter are included in the new collection
3. Original had no return value
   1. The original method never returns a value. Instead, it modifies the content attribute of the current FileCollection instance.
   2. The updated method on the other hand returns a new FileCollection object that contains the filtered files.

The list function was also updated compared to the previous in order to be more detailed. Here are the changes:

1. Output
   1. The original returned a string containing the list of entries in the file collection, with each entry on a new line.
   2. The updated version instead prints out the details of each file in the collection directly to the console, including the file name, size, and modification time.
      1. The issue with this change is not reusable as the output is printed directly to the console and cannot be easily reused or processed further.
      2. It’s also less flexible as the format is fixed
2. Return Value
   1. Original returns a string which can be stored or processed further later
   2. While the updated version does not as it prints directly to the console
3. Details Provided
   1. Original only provides the file name
   2. Updated version includes several details about the file in question
4. Flexibility
   1. Original includes an options parameter for potential customisation
   2. While the updated version does not and instead contain a fixed format
5. Usage Context
   1. The original is suitable for programmatic uses such as logging or further processing
   2. While the updated version is more suitable for direct user interaction such as CLI output.

### ***FSObject Changes***

The FSObject was modified slightly for a similar reason to the FileCollection file, for a better use case situation.

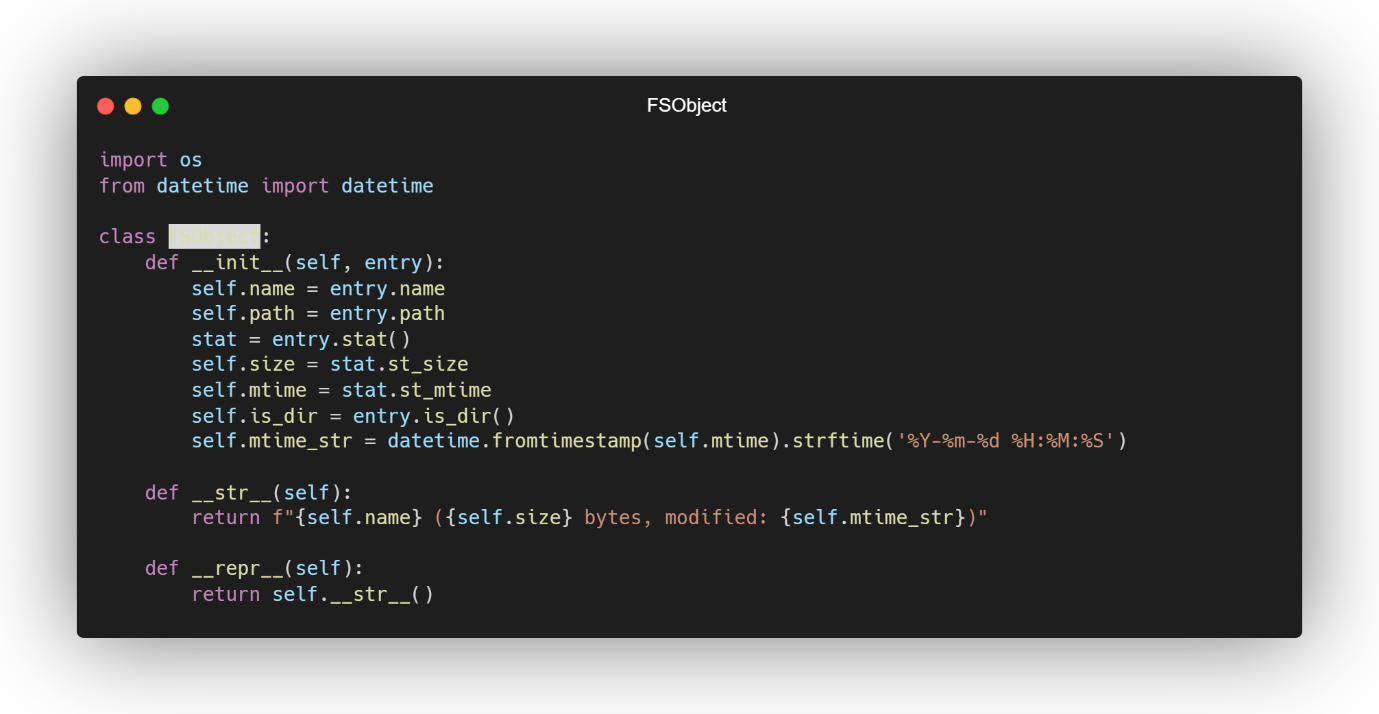
Main differences between the original code and the now updated version relates to how best to optimize the operations and performance. Such as how the meta data was being handled.

The metadata in the updated version explicitly stores the metadata as attributes, while the original relies on dynamic access via Path. Switching from a Pathlib-based method to a Scandir-Based method. The original had a more minimal take while the updated version lists out all the attributes then storing them.

The performance in the original was slower due to repeating file system calls for properties, this was cleaned up and optimized as the metadata was stored during the initialization process.

The flexibility in the updated version was more concrete and specific as it was designed to handle a Scandir-based method of operations, while the original was more abstract and generic which works best for Pathlib-based method of operations.

The string representation for the original was simple as it was just a file path, while the updated version is more detailed to show more details about the file object.

The directory handling for the updated version was explicit using a is\_dir attribute instead of an implicit handle which can infer from Path which is done in the original. 

### ***Selector Changes***

The updated version of the Selector class introduces significant changes in terms of design philosophy, flexibility, functionality, and implementation details compared to the original version. Below is a detailed explanation of the differences and changes made:

#### Design Philosophy

Original:

* The original Selector class is attribute-based. It uses instance attributes (nameContains, olderThan, biggerThan) to define filtering criteria.
* Each criterion is tested individually using dedicated methods (testNameContains, testOlderThan, testBiggerThan).
* The filtering logic is tightly coupled to the class, and the criteria are limited to the predefined attributes.

Updated:

* The updated Selector class is function-based. It uses a filter\_func attribute, which is a callable function that encapsulates the filtering logic.
* The class provides static factory methods (name, size, date) to create Selector objects with specific filtering logic.
* This approach is more flexible and extensible, as it allows for custom filtering logic beyond the predefined criteria.

#### Filtering Logic

Original:

* The filtering logic is hardcoded into the class. Each criterion (e.g., name, size, date) has a dedicated method (testNameContains, testOlderThan, testBiggerThan) that performs the corresponding test.
* The testFSObject method combines these tests, returning True only if all criteria are satisfied.
* The class supports only one criterion at a time, as noted in the comments.

Updated:

* The filtering logic is encapsulated in the filter\_func attribute, which can be any callable function.
* The static methods (name, size, date) generate specific filtering functions and return Selector objects that use these functions.
* The updated version supports combining filters using the intersect method and negating filters using the not\_ method, enabling more complex filtering logic.

#### Criteria Handling

Original:

* The class supports three specific criteria:
  + nameContains: Checks if the file name contains a specific substring.
  + olderThan: Checks if the file is older than a specified number of days.
  + biggerThan: Checks if the file size is larger than a specified threshold (in kilobytes).
* These criteria are limited in scope and cannot be easily extended.

Updated:

* The updated version supports more flexible and expressive criteria:
  + Name: Uses glob-like patterns (e.g., \*.txt) converted to regular expressions for matching.
  + Size: Supports conditions like "> 10K" or "< 1M", with units (B, K, M, G) and comparison operators (> or <).
  + Date: Supports conditions like "> 30 days" or "< 1 year", with units (days, months, years) and comparison operators.
* The updated version is more extensible, as additional static methods can be added to handle new criteria.

#### Implementation Details

Original:

* The filtering logic is implemented using simple comparisons and the pathlib.Path object.
* The testNameContains method uses Path.match to check if the file name contains the specified substring.
* The testOlderThan and testBiggerThan methods perform straightforward comparisons using the file's modification time and size.

Updated:

* The filtering logic is implemented using more advanced techniques:
  + Regular Expressions: The name method converts glob patterns to regular expressions for more flexible matching.
  + Parsing Conditions: The size and date methods parse human-readable conditions (e.g., "> 10K", "< 1 year") into thresholds for comparison.
  + Datetime Handling: The date method uses datetime and timedelta to calculate thresholds for date comparisons.
* The updated implementation is more robust and user-friendly, with error handling for invalid patterns or conditions.

#### Combining and Negating Filters

Original:

* The original version does not support combining or negating filters. Each Selector object operates independently, and there is no way to combine multiple criteria into a single filter.

Updated:

* The updated version introduces two new methods for combining and negating filters:
  + intersect: Combines two Selector objects using a logical AND operation. The resulting filter matches objects that satisfy both filters.
  + not\_: Negates a Selector object, creating a filter that matches objects that do not satisfy the original filter.
* These methods enable more complex filtering logic, such as combining multiple criteria or excluding specific files.

#### String Representation

Original:

* The \_\_str\_\_ method provides a basic string representation of the Selector object, listing the criteria that are set.

Updated:

* The updated version does not include a \_\_str\_\_ method. Instead, the filtering logic is encapsulated in the filter\_func attribute, and the class focuses on functionality rather than representation.

#### Error Handling

Original:

* The original version does not include explicit error handling. It assumes that the input criteria are valid and does not handle exceptions.

Updated:

* The updated version includes error handling for invalid patterns or conditions:
  + The name method catches re.error exceptions for invalid regular expressions.
  + The size and date methods raise ValueError for invalid conditions and print warnings when exceptions occur.
* This makes the updated version more robust and user-friendly.

The updated version of the Selector class is a significant improvement over the original. It is more flexible, extensible, and robust, with support for advanced filtering logic, error handling, and combining filters. While the original version is simpler and easier to understand, the updated version is better suited for real-world applications where complex filtering criteria are required.

### ***Main and FspowVisitor***

The main.py script file was designed to process and execute a cusomt scripting language for file system operations. This meant that anltr related imports were used to parse the input script (fspowLexer, fspowParser, CommonTokenStream, etc) and custom classes such as:

* FileCollection: Represents a collection of files in a directory.
* Selector: Provides filtering functionality (e.g., by name, size, or date).
* TreesUser: Likely used for debugging or pretty-printing the parse tree.

A screen shot of a computer

AI-generated content may be incorrect.

#### **FspowVisitor**

Along with these imports, the FspowVistor class was put inside the main.py file as there were issues with importing the class into the main.py file. The Vistor class interprets the parsed script by visiting nodes in the parse tree. It implements the logic for each rule in the grammar.

For example, variable dictionary, which stores variables created during the script execution and the visit method, which recursiverly visits nodes in the parse tree and executes the corresponding logic based on the grammar rule.

The grammar rules and their implementation are done through several different rules defined in the grammar language.

A screen shot of a computer screen

AI-generated content may be incorrect.

#### prog Rule

* Represents the program's entry point.
* Iterates through all statements in the script and visits them.

#### assignment Rule

* Handles variable assignments.
* Stores the result of the right-hand side expression in the variables dictionary.

#### fcCreation Rule

* Creates a FileCollection object using a specified root path.

#### selCreation Rule

* Creates a Selector object based on a filter (e.g., name, size, or date).

#### selfilter Rule

* Implements filtering logic:
  + Filters by name, size, or date.
  + Supports negation (not) and combining filters (intersect).
  + Handles parentheses for grouping filters.

#### fcApplySelector Rule

* Applies a Selector to a FileCollection and lists the filtered results.

#### fcList Rule

* Lists all files in a FileCollection.

#### message Rule

* Prints a message to the console.

#### Other Rules

* expressions and rootSpecifier handle specific parts of the grammar, such as evaluating expressions or specifying root paths.

#### **Main**

The main function operates by reading the input script file from the command line. Once it reads it, it uses Antlr to tokenise and parse the script. It proceeds to print out the parse tree using the TreeUsers function. Finally ending by executing the script by visiting the parse tree with the FspowVistor.

The script supports creating file collections, applying filters, and performing operations like listing files or printing messages.

A screen shot of a computer

AI-generated content may be incorrect.

# Appendix

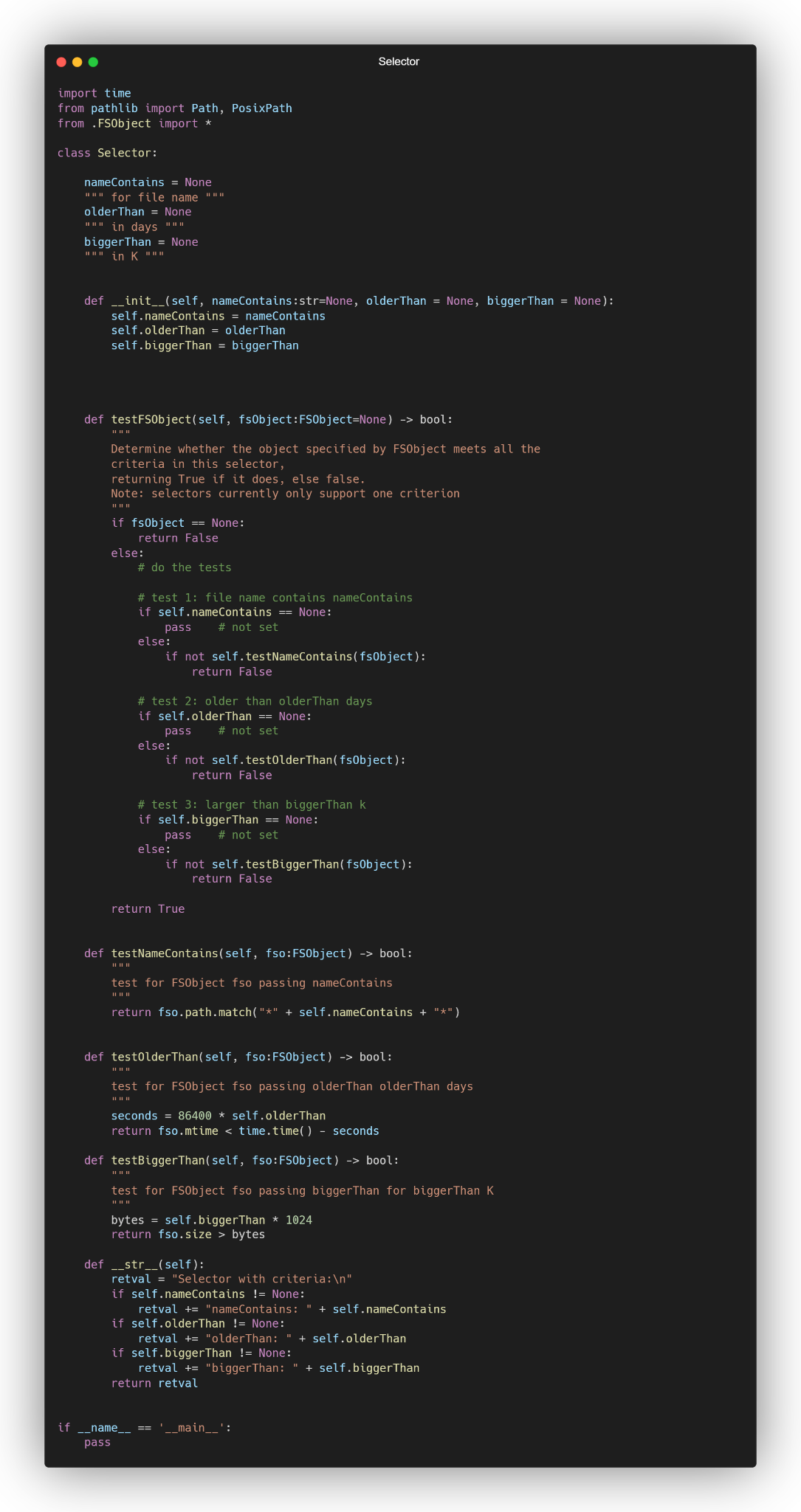
FSObject Original:



FileCollection Original:



Selector Original:



Fspow Original:

