University of Crete
Department of Computer Science

The Galaktion Language for XML Data Management and Application on Map Visualization

Ioannis Savvakis
R.N: 4234

Supervising Professor: Yannis Tzitzikas
Full Professor, University of Crete

Committee Member/Evaluator: Dimitris Plexousakis
Full Professor, Department of Computer Science, University of Crete

Co-Supervisor: Pavlos Fafalios
Assistant Professor, School of Production Engineering and Management of the Technical University of Crete

Heraklion, November 2025

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Web Development and Programming Language Development are two major areas of Computer Science that are not directly intertwined with each other. However, in this Thesis, both areas have an important role, each on its own way.

Below, a Web Map Application and Galaktion, a simple Programming Language, will be presented.

The Web Application was developed with the aim of visualizing RICONTRANS project's religious/historical/cultural data on a map and Galaktion was developed to meet the needs of the Application and the specific requirements of processing the raw form of this data.

2 Galaktion v0.2.0 - Usage

Galaktion v0.2.0 is a simple general-purpose scripting imperative programming language but with focus on linked XML Data management, built to support special-purpose concepts.

It was designed to meet specific requirements and the needs of many applications like the Web Map Application described in Chapter 4.
Some of them are:

- user-friendly and easy-to-learn syntax
- XML data management
- ability to configure and store key-value pairs in JSON format
- and, mainly, the ability from one XML file to navigate to the contents of another, and from there to another's, and so on based on a relation between them, with a simple syntax.

Galaktion is named after saint Galaktion (November 5th of each year).

2.1 Background and Definitions

In this Chapter, terms like XML, XPath, JSON, CSV, XQuery and more are used. For the reader's convenience, their definitions are listed below.

Extensible Markup Language (XML) *is a markup language and file format for storing, transmitting, and reconstructing data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The World Wide Web Consortium's XML 1.0 Specification of 1998 and several other related specifications - all of them free open standards - define XML.* [1]

XML Path Language (XPath) Version 1.0 is an expression language designed to address parts of an XML document. It gets its name from its use of path notations for navigating through the hierarchical structure of the XML document. [2] [3]

XML Query (XQuery) *is a query language and functional programming language designed to query and transform collections of structured and unstructured data, primarily in the form of XML. XQuery 3.1 extends XQuery to support JSON as well as XML.* [4] [5]

JavaScript Object Notation (JSON) *is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.*

It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. [6]

Comma-Separated Values (CSV) *is a plain text data format for storing tabular data where the fields (values) of a record are separated by a comma and each record is a line (i.e. newline separated). CSV is commonly-used in software that generally deals with tabular data such as a database and a spreadsheet.* [7]

2.2 I/O and First Program

2.2.1 Data Input

Galaktion's only data source is XML files and is explained in section 2.8.

2.2.2 Data Output

Galaktion v0.2.0 outputs data in two way s:

1. By creating a new JSON or CSV file with user-defined contents, which is discussed later, in section 2.13.
2. By printing them to the system's standard output as explained below.

2.2.3 "print" Statement and First Program

Using the `print` statement, the user can output/print data to the standard output. It is possible to print multiple data by separating them with comma (,) as shown in Figure 2.1.

```
print "Hello world!"
print 'Welcome', " to Galaktion!"
```

```
Hello world!
Welcome to Galaktion!
```

Figure 2.1 | First Galaktion program (top) and its output (bottom)

2.3 Primary Types and Operations

Galaktion supports the following literal types.

These types are used by Galaktion to manage different types of data internally. They are not available to the user as language keywords/type specifiers.

Each literal can be seen as stand-alone, since it is evaluated to a complete expression.
This also means that it could be a valid Galaktion program by itself.
This behavior of the language results directly from its grammar, discussed in Chapter 3.

2.3.1 Literal "none"

Keyword "none" is a special literal that represents the absence of a regular value.
It is evaluated to a None type expression.

The supported operations for this type are:

- **Equality**, with operator "**=**"
- **Inequality**, with operator "**!=**"

✓ **Comparison operations** are evaluated to **boolean expressions**.

2.3.2 Numbers

A number is typed as it is.
If it is decimal, the decimal point is represented by a dot (.).
If a number has a "+" sign or is unsigned is considered non-negative, while having a "-" sign means it is negative.

`2` `3.14` `-2.7` `+0`
All the above are valid numeric literals.

The supported operations are:

- **Addition**, with operator "**+**"
- **Subtraction**, with operator "**-**"
- **Multiplication**, with operator "*****"
- **Division**, with operator "**div**"
- **Modulo**, with operator "**mod**"
- **Negation**, with operator "**-**" (unary)

- **Equality**, with operator “**=**”
- **Inequality**, with operator “**!=**”
- **Greater than or equal to**, with operator “**>=**”
- **Less than or equal to**, with operator "**<=**”

✓ **Arithmetic operations** are evaluated from left to right to **arithmetic expressions**.

✓ **Comparison operations** are evaluated to **boolean expressions**.

2.3.3 Strings

A string literal is any alphanumeric sequence enclosed in either double or single quotes (“...” or ‘...’).
Strings are immutable, indexable and sliceable.
They are written in a single line and cannot span across multiple lines. Any formatting to their contents is done by using escape characters.

The supported operations are:

- **Concatenation**, with operator “**+**”
    - ➢ Left operand must be a string and right one can be of any type.
- Galaktion will concatenate the left operand with the string representation of right operand.
- **Equality**, with operator “**=**”
    - ➢ Two strings are equal if they have the exact same contents.
- **Inequality**, with operator “**!=**”

✓ **Concatenation** operation is performed from left to right and evaluates to a **string expression** containing the new string.

Figure 2.2 demonstrates a little more complex use of strings than Figure 2.1.

Note: Due to lack of horizontal space, the string literals will appear folded across this document, but no new line character was used. As mentioned above, Galaktion strings are single-line.

```
print '\tMike\'s friend used to walk through' + "\n" +
'"Kapodistriou" street from ' + 1.5 + ' to 2 km.'
```

```
        Mike's friend used to walk through
"Kapodistriou" street from 1.5 to 2 km.
```

Figure 2.2 | Printing a bit more complex string expression - here 1.5 is a number while 2 is part of a string literal

2.3.4 Booleans

Boolean literals are `true` and `false`.
The command `print true, "/", false` will output

```
true/false
```

The supported operations are:

- **Logical AND**, with operator “**and**”
- **Logical OR**, with operator “**or**”
- **Logical NOT**, with operator “**not**”
- **Equality**, with operator “**=**”
- **Inequality**, with operator “**!=**”

✓ **Logical operations** are evaluated from left to right to **Boolean expressions**.
✓ **Comparison operations** are evaluated to **Boolean expressions**.

Logical operations are performed between Boolean expressions.
If an expression is of a non-Boolean type, first it is converted to Boolean and then the operation is performed.
Table 2.1 shows the different type conversions to Boolean.

Expression Type	Expression Value	Boolean Conversion
None	none	false
Number	0	false
	non-0	true
String	'' *or* ""	false
	non-empty string	true
Array	[]	false
	non-empty array	true

Table 2.1 | Different expression types to Boolean expression conversions

2.3.5 Arrays

A Galaktion array stores one or more expressions of any type separated with comma (,) enclosed within square brackets ([…]).
Arrays are mutable, indexable and sliceable.
An array item's contents are always variable.

The supported operations are:

- **Insertion**, with operator "**+**"
  - Left operand is an array and right operand is of any type and a copy of it is inserted at the end of the array.
- **Extension**, with operator "**++**"
  - Both operands are arrays. Copies of each value of the right array are inserted (with the same order) at the end of the left array, extending it.
- **Equality**, with operator "**=**"
  - Two arrays are equal if they have the exact same contents.
- **Inequality**, with operator "**!=**"

2.4 Identifiers and Type Specifiers

The user can define identifiers and assign values to them to use them across the program. Every identifier has attributes like its type, value, mutability (whether it is variable or constant) and name.
Identifiers in Galaktion v0.2.0 are always global regardless of where they were declared.

The declaration syntax is C-like as follows:

[*type specifier*] [*identifier's name*]
or
[*type specifier*] [*identifier's name*] [*assignment operator*] [*an expression*]

The second syntax is used to initialize the identifier, and it is essentially an assignment expression with the declaration embedded in it.

An example can be seen in Figure 2.3.

Identifier names begin with "$". Then must follow one or more English letters or numeric digits or underscores.

This pattern was chosen for being as close to XPath variable notation as possible, so that the user does not have to keep many different rules in mind.

After its declaration, an identifier is by default variable and if uninitialized its value is `none`.
To reduce the possible errors, Galaktion allows the redeclaration of an identifier.

Five type specifiers are available:

- `folder`
- `xmlfile`
- `jsonfile`
- `csvfile`
- `flexible`

2.4.1 Type Specifier “flexible”

A variable of type `flexible` can hold any type of value at any time.

2.4.2 Strict Type Specifiers

The rest of the type specifiers (`folder`, `xmlfile`, `jsonfile`, `csvfile`) are strict, which means that they cannot hold every type of value.

In version 0.2.0, strict types are string-based, thus the value of a strict type is essentially a string literal.
The only thing that changes is their meaning. This is done for user’s “safety” as strict types represent external data structures that their type physically cannot be changed (e.g. a folder cannot become a number).

As a result, a strict type can hold values only of its own type and type string.

2.4.3 Built-in variable “$GALAKTION_ROOT”

A built-in identifier named `$GALAKTION_ROOT` is automatically declared for each Galaktion v0.2.0 program.
`$GALAKTION_ROOT` is of type `folder` and it begins as a variable. It follows all the rules that govern Galaktion identifiers.
Its job is to hold at any time the Current Working Directory of user’s source code.
Any change to its value effectively changes the Current Working Directory.

Every path to a file or folder is relative to `$GALAKTION_ROOT`.

2.5 Assignment and Identifier’s Mutability

2.5.1 Value Assignment

Assignment operations follow the syntax below:

[*left operand*] [*assignment operator*] [*right operand*]

The right operand can be any expression.
The left operand can be a previously declared identifier, or an identifier’s declaration (as mentioned in section 2.4), or an array’s item.

A copy of right operand’s value (after some processing as described in the next section) is assigned to the left operand.
All the assignments are done **by value** – not by reference.

2.5.2 Analysis of Assignment Operators

An assignment operator consists of two parts:

- The first/left part defines the mutability of the left operand after the assignment.
- The second/right part is one of the language’s operators and indicates the processing (if any) that the right operand’s value will go through before the assignment is performed.

Table 2.2 shows a list of the available assignment operators.

Left Operand Remains Variable	Left Operand Becomes Constant
:=	.=
:+	.+
:-	.-
:*	.*
:div	.div
:mod	.mod
:++	.++

Table 2.2 | Assignment Operators - Depending on which one is used, after assignment is completed the left operand remains variable (left column) or becomes constant (right column).

First /Left part:

- **:** After assignment, left operand will still be variable.
- **.** After assignment, left operand will be constant.

Second/Right part:

- **=** Left operand's value after assignment will be same as right operand's value (no further processing).
- **+**, **-**, *****, **div**, **mod**, **++** Left operand's value after assignment will be the result of **[*left operand*] [*second/right part*] [*right operand*]** operation.

In Figure 2.3 below, the assignment operator's usage becomes clearer.

```
`Declare two constants with names $step and $space and
initialize them to 2 and one-space string respectively`
flexible $step .= 2
flexible $space .= ' '



`
In the following lines, a little more of the grammar's
flexibility can be observed.
`



`Declare a variable named $counter, initialize it to 0
and print the result of the assignment.`
print flexible $counter := 0



`Start incrementing $counter, each time by step:`

`
In each of the next 3 lines:
    1. The value of $counter + $step is assigned to $counter
       using the operator :+ ($counter remains variable)
    2. The result of $space multiplied with the new $counter
       is printed
    3. The new $counter is printed
`
print $space * ($counter :+ $step), $counter
print $space * ($counter :+ $step), $counter
print $space * ($counter :+ $step), $counter
```

```
`Start decrementing $counter:`

`
Any of the following would cause an error:
    $step := -2    or    $step .= -2
    $step :* -1    or    $step .* -1
and generally any assignment to $step or $space
will result in an error because they are declared
as constants.
`


`This is one of the possible solutions: `
flexible $negativeStep .= -$step


`
The following 3 lines decrement $counter and are
similiar to the 3 incrementing ones.

The main difference here is that the $counter is
not printed at the end separately.

The spacing string is firstly concatenated with
$counter's string representation and then the
result is printed all at once.
`
print $space * ($counter :+ $negativeStep) + $counter
print $space * ($counter :- $step)         + $counter
print $space * ($counter .+ $negativeStep) + $counter



`For the last assignment to $counter, the
operator .+ was used.

This means that from then on $counter is constant.

Therefore the following line causes an error.`
$counter := $counter
```

```
0
  2
    4
      6
    4
  2
```

```
0
Error: at line 77: Assignment to a constant
```

Figure 2.3 | Example program showing some scenarios discussed earlier like type specifiers, assignment operators, variables, constants and others (top) and its output (bottom)

2.6 Statements if and if...else

These statements follow the C language's syntax except that there is no need for enclosing the condition in parentheses, as shown below.

```
    if [an expression] [a statement]
    else [a statement]
```

The `else` part is optional.

2.7 Directives

Directives follow the same naming rules as identifiers with the only difference being that instead of `$` they start with `#`.

Their purpose is to provide extra information/directions for Galaktion about a certain command.

They are written next to each other separated with white characters usually at the end of the command.
To reduce the number of possible errors, when Galaktion encounters a wrong directive, it just ignores it and provides a warning message.

2.8 XPath Expressions and "from" Statement

As mentioned earlier, Galaktion's input source is XML files and its way of reading them is through XPath 1.0 expressions.

2.8.1 XPath 1.0 Expressions Usage

XPath 1.0 expressions are a central feature of the language.

Everything, since it is lexically valid, is considered by default to be an XPath expression unless Galaktion knows that it is something else (e.g. a $ followed by letters is an identifier).
Even the * is a full XPath expression until Galaktion sees user's intention to use it as operator.

In Galaktion v0.2.0, XPath expressions have their root at the specified XML document's root. So, their starting element is always the document's root element for both absolute and relative ones.

2.8.2 "from" statement

To specify the source XML file from which XPath expressions will retrieve data, the `from` statement is used with the following syntax:

> *`from`* [*an expression of type* `xmlfile`]
>
> or
>
> *`from`* [*an expression of type* `xmlfile`] [*code block*]

Unlike identifiers, source XML files are scoped.
Starting at global scope with depth 0, every blocked `from` statement inserts a new scope with +1 depth.

At any time across any scope only one source file is active, and this is the last one specified by "from" statement. All the previously specified are shadowed.

Once a scope is executed, the current source XML file becomes again the one that was right before the scope.

If an XPath expression is executed without having specified any source file, an error will occur.

2.9 From File to File: The "->" Operator

When working with structures of related XML files (e.g. XML databases) it is usually needed to access other XML files related to the one that is currently being processed.

The `->` operator gives the user access to an XML file while working with another.

2.9.1 How it works

The `->` operator connects different XPath expressions with each other visually forming a chain.
The first expression in this chain is level-1 XPath expression, the second is level-2 XPath expression, and so on.

It performs linking to the indicated by the left XPath expression's current node and evaluates the right XPath expression on it, and so on until the whole chain is evaluated.

A code block for the `->` must be defined for Galaktion to execute it whenever encounters an `->`.
The arrow's code block follows the syntax below:

```
-> {
      [an expression][directives],
      [an expression][directives]
}
```

Executing the above code block means for Galaktion to search for an XML file according to the two expressions given.

Both expressions are of string-based type.
One expression holds the path to the location where the file will be searched, and the other holds the name of the file to search for.
Which one holds what, is determined by each one's directives:

- **To hold the path to the file's location:** The `#filelocation` directive must be used. If user wants Galaktion to also search inside this location's subfolders, the `#subfolders` directive has to be used too.
- **To hold the file's name:** The `#filename` directive must be used.

Once the file is found, it is used as the source XML file for the current level (introduced by the `->` operator) in the XPath expression chain.
So, the XPath expression following the `->`, works on the file found by it.

The arrow's code block is executed one time for every item of the results of the XPath expression prior to the `->` that is an XML element. Furthermore, at each execution the respective XML element is used as the starting point of relative XPath expressions.

Examples demonstrating the arrow's use are available in the Galaktion's GitHub page.

2.9.2 Relative XPath expressions

As mentioned in section 2.68, both absolute and relative XPath expressions in Galaktion v0.2.0 start from the current source XML file's root element.

The only exception is the arrow's code block.
Inside this code block, any relative XPath expression has its starting point at the XML element of the results of the XPath expression prior to the `->` on which the code block is executed.
Therefore, an XPath expression like `@id` would make more sense inside arrow's code block.

2.10 "foreach" Loop Statement

`foreach` statement follows the syntax below:

```
    foreach [xmlfile declaration] in [folder expression] {
        [code]
    } [directives]
    Or
    foreach [of-any-type declaration] in [array expression] {
        [code]
    } [directives]
```

The top syntax is used to iterate through all the XML files in a folder. During each iteration the identifier of the given declaration holds the path to the current XML file.
Furthermore, the current XML file becomes the source file for the current iteration. After `foreach` statement the source file is again the same as before.

The bottom syntax is used to iterate through an array. During each iteration the identifier of the given declaration holds a reference to the current array item. Any change to it changes the actual item.
This is the only case of Galaktion where the user accesses data through reference. Also, this case does not affect the source files.

The examples shown in Figure 2.4 and Figure 2.5 demonstrate both cases.

```
`Declare an array with folder names`
flexible $folders .= [
    "Cars",
    folder $motorcycles .= "Motorcycles",
    'Bikes'
]

`Iterate through the array`
foreach flexible $i in $folders {
    print 'Opening "', $i, "\" folder..."

    `For each array item, print all its contained XML files`
    foreach xmlfile $fil in folder $fol := $i {
        print "\t", $fil
    }

    print `just a new line`
}
```

```
Opening "Cars" folder...
        Cars/Car1.xml
        Cars/Car2.xml

Opening "Motorcycles" folder...
        Motorcycles/Motorcycle1.xml
        Motorcycles/Motorcycle2.xml

Opening "Bikes" folder...
        Bikes/Bike1.xml
        Bikes/Bike2.xml
        Bikes/Bike3.xml
```

Figure 2.4 | Example program showing the use of "foreach" statement (top) and its output (bottom)

```
`Declare an array that holds the multipliers of the ' '`
flexible $array .= [0, 2, 4, 6, 4, 2, 0]

`For each item, print <item> spaces followed by the <item>`
foreach flexible $i in $array {
    print ' '*$i + $i
}
```

```
0
  2
    4
      6
    4
  2
0
```

Figure 2.5 | The program of Figure 2.3 implemented using "foreach" loop (top) and its output (bottom)

2.11 Indices and Slices

Strings and Arrays are indexable and sliceable.

Indices and Slices syntax are shown below:

Index:	*[* [*arithmetic expression*] *]*
Slice:	*[* [*arithmetic expression*] ~ [*arithmetic expression*] *]*

The indexing of the elements from first to last starts from 1 while from last to first starts from -1.
Indexing at 0 results in an empty array or empty string, depending on the indexed type.

Slices use the expression before ~ as the **inclusive start** index and the one after ~ as the also **inclusive stop** index.
Start and stop are regular indices as described above.

A visual explanation is shown in Table 2.3 below.

Array Declaration	`flexible $continents .= [`
	`    "Africa",`
	`    "Antarctica",`
	`    "Asia",`
	`    "Australia",`
	`    "Europe",`
	`    "N. America",`
	`    "S. America"`
	`]`						

Contents	"Africa"	"Antarctica"	"Asia"	"Australia"	"Europe"	"N. America"	"S. America"
All Possible Indices	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]
Example Slices	▨	[2~4]			▨	▨	▨
	▨	▨	[3~-1]				
	▨	▨	[3~]				
	[-7~-3]					▨	▨
	[~-3]					▨	▨
	[~5]					▨	▨
	[~]						

Table 2.3 | Visual examples of indices and slices

2.11.1 Galaktion Indexing and Slicing in XPath Expressions

Galaktion adds to XPath 1.0 predicates index-like functionality, allowing predicates containing Galaktion code to be part of an XPath expression and act like Galaktion indices and slices.
A predicate cannot contain XPath and Galaktion code at the same time. It is, however, possible for an XPath expression to have more than one predicate, some of which may contain XPath code and some Galaktion code.

For example, if the user wants to get the texts from all the elements that each is the last element of its parent element, inside the Galaktion's environment he would use the following XPath expression:

```
//*[-1]/text()
```

This expression is a valid XPath 1.0 expression, but the reason it works the way the user wants in this example is because the [-1] is considered to be Galaktion code (an index) and not a pure XPath predicate, thus it is interpreted as "*the last element*".

If the same expression was used outside of Galaktion environment as pure XPath 1.0 code, it would not cause any errors (valid XPath code as mentioned before), but it would also not work the way the user wants, since the [-1] would be translated into the XPath predicate [position() = -1], which of course is never true.

Therefore, the user would get an empty array with no results inside.

This is a syntactic sugar feature of the language that adds support for slicing and negative indexing to an XPath expression in a user-friendly way.

2.12 “extract” Statement

Galaktion keeps an internal memory where the user can store key-value pairs of their choice, and this is achieved by using `extract` statement.

Important: *extract statement stores the key-value pairs in a file-centric way according to the current XML source file. That is, all the pairs are grouped together according to which the source file was at the time of their creation.*

`extract` statement follows the syntax below:

`extract` [*an expression*] *`as`* [*an expression*] [*directives (optionally)*]

The first expression is the value, and the second one is the key.
The key expression must be of a string-based type or else an error will occur.
The value expression is of any type.
If directives are needed, they can be written at the end.

2.13 “generate” Statement

The user can generate a CSV or a JSON file with the contents of Galaktion’s internal memory using `generate` statement as follows:

`generate` [*file type*] *`as`* [*an expression*] [*`#flush`* *(optionally)*]

The file type can be only `jsonfile` or `csvfile`.
The expression holds the path of the file that is going to be generated with its name and must be string-based.

After the generating, the internal memory’s contents remain unharmed.
The user can type the `#flush` directive at the end of the `generate` statement and after generating is complete, internal memory will be completely empty.

2.14 Example Code

An example code working on RICONTRANS (see 4.1) data, and specifically on file `RICONTRANS_DATA_2023-10-24/Objects/3/Objects2159.xml`, is shown in Figure 2.6 below.

Its goal is, since the file is saved, to create two files, a JSON and a CSV, that contain the following information:

- The name of the file
- The materials used for the represented object
- The name of the XML file that represents the location of this object
- The name of the location
- The coordinates of the location
- The IDs of five different objects that have passed through this location throughout the centuries.

```
` Setting the cwd where is convinient for this case `
$GALAKTION_ROOT .= 'path / to /RICONTRANS_DATA_2023-10-24'


`Definition of arrow's code block`
-> {
    @sps_type + @sps_id + '.xml'    #filename ,
    @sps_type                       #filelocation #subfolders
}


` Declare $myFile that will represent the XML file
  at the specified location and make it constant. `
xmlfile $myFile .= 'Objects/3/Objects2159.xml'


from $myFile {
    ` If the file is not saved... `
    if //admin/saved/text() != 'yes' {
        print $myFile, " - not saved" ` ...print a message... `
        exit ` ...and stop execution. `
    }

    ` Else... `


    ` store the value of $myFile `
    ` as 'Source File' `
    extract $myFile as 'Source File'
```

```
` Create the constant $materials that holds `
` the texts of all PrincipalMaterial nodes  `
` and store it as 'Materials'. `
flexible $materials .= //PrincipalMaterial/text()
extract $materials as 'Materials'



` The lines below retrieve the attributes `
` sps_type and sps_id  of node `
` CurrentLocation/Location from wherever inside `
` $myFile this is. `
` As attributes are always strings, these three `
` additions will conclude in a single string which `
` which will be stored as 'Location specs'. `
extract   //CurrentLocation/Location/@sps_type
        + //CurrentLocation/Location/@sps_id
        + '.xml'
as 'Location specs'



` find the node CurrentLocation/Location inside the $file `
` and using -> go to the file that it indicates `
` and inside that file, find the node Name, get its text `
` and store it as 'Location Name' `
extract //CurrentLocation/Location->Name/text()
as 'Location Name'



`find the node CurrentLocation/Location inside the $file
 and using the -> go to the file that it indicates
 and inside that file, find the node Coordinates/Value,
 get its text, convert it to lat and long and store it as
 'Location Coordinates' `
extract //CurrentLocation/Location->Coordinates/Value/text()
as 'Location Coordinates' #latlong



` Declare the constant $id containing the text of admin/id `
flexible $id .= //admin/id/text()
```

```
    ` L1: find the nodes ref_by under refs_by that have an
          attribute sps_type with value "ObjectTransfers" and
          from each one of them go to the file that each
          indicates. From there, `
    ` L2: find the nodes Object under MovedObject, go to the
          files that each indicates, and from there `
    ` L3: find the nodes Location under CurrentLocation, go to
          the files that each indicates. From there, `
    ` L4: find the nodes ref_by that have an attribute sps_type
          with value "Objects" and one sps_id with different
          value from the $id constant, keep only the first five,
          and go to the files that they indicate. From there, `
    ` L5: find the nodes id under admin and get their texts and
          store them as
         'Other objects passed through the involved Locations'.`
    extract //refs_by/ref_by[@sps_type="ObjectTransfers"]->
            //MovedObject/Object->
            //CurrentLocation/Location->
            //ref_by[@sps_type='Objects' and @sps_id != $id][~5]
                                 `Keep only the first 5 elements` ->
            //admin/id/text()
    as 'Other objects passed through the involved Locations'
}


` Print a message with the last 15 character of $myFile. `
print 'Done processing ', $myFile[-15~]


` Generate a JSON file output.json in the specified location `
` containing the data that have been extracted until now. `
generate jsonfile as 'output.json'


` Generate a CSV file output.csv in the specified location `
` containing the data that have been extracted until now. `
` After the process is complete, empty the internal memory
(#flush). `
generate csvfile as 'output.csv' #flush
```

Figure 2.6 | Example Code

Indicatively, the contents of the `output.json` are presented in the next figure.

```
[
    {
        "Source File": "Objects/3/Objects2159.xml",
        "Materials": [
            "gold thread",
            "semi-precious stone/stones"
        ],
        "Location specs": "Location1269.xml",
        "Location Name": "Cetinje Monastery",
        "Location Coordinates": {
            "lat": 42.38778349850573,
            "long": 18.92178833478511
        },
        "Other objects passed through the involved Locations": [
            "2130",
            "2131",
            "2132",
            "2134",
            "2135"
        ]
    }
]
```
Figure 2.7 | output.json

Another example code is discussed in Chapter 4.
More examples can be found in Galaktion's GitHub repository.

3 Galaktion v0.2.0 - Implementation

Galaktion v0.2.0 is entirely implemented using Python and Python Lex-Yacc (PLY) package.

A high-level abstraction of Galaktion's architecture consists of three main functional units:

1) The User Interface Unit (see 3.1) that handles the input and output (I/O) from and to the console/end user.
2) The Lexical Analyzing Unit (see 3.2) which scans the user's source code and produces tokens.
3) The Syntax Analyzing and Code Generation Unit (see 3.3) that consumes the tokens and executes user's code.

In the following sections, the reader will have the opportunity to look deeper in Galaktion's implementation.

3.1 Important Structures: Galaktion_core.py

Galaktion_core.py contains important data structures that aim to make the development of Galaktion easier and the code more human-readable. They are described in the next sections.

3.1.1 Class UserInterface

It is the class that creates objects responsible for the interaction between user and Galaktion.
It has methods (the most important of which are mentioned below) and the following properties with their initializations:

```
1) self.errors = []
2) self.warnings = []
3) self.userSourceFile = from command line arguments
4) self.terminateOnError = True
5) self.printErrorImmediately = True
6) self.printWarningImmediately = True
```

1) holds all the errors as `Galaktion_core.Error` objects.
2) holds all the warnings as `Galaktion_core.Warning` objects.
3) holds the path to the user's source code.

4), 5), 6) are flags for future use.

Inside `__init__` method an argument parser from package `argparse` is used for parsing the user's command line arguments.

Method `emitWarning` appends `self.warnings with a new Galaktion_core.Warning` object. If `self.printWarningImmediately is True` pops it and prints it immediately.
When called, `printWarnings` prints the contents (if any) of `self.warnings`.

The same principle applies to emitError and printErrors.
Additionally, if `self.terminateOnError` is `True` `emitError` terminates Galaktion's execution.

3.1.2 Classes Warning and Error

They are used to represent a Galaktion warning and a Galaktion error respectively. They each have the following properties:

1) `self.message`
2) `self.line`

3.1.3 Class Stack

It is used to model a stack.

3.1.4 Variable `memory`

It is Galaktion's internal memory.
It is a dictionary where `extract` statement stores the key-value pairs.

3.1.5 Class Checks

It has static methods that are used to perform various checks.

3.1.6 Class Directives

It holds all the available directives as properties.

3.1.7 Class Index

It models the concept of index.

It has the following properties:

1) `self.index_galaktion`
2) `self.index_python`
3) `self.index_xpath`

They each hold the Galaktion, Python and XPath representation respectively for the same index.

1) is essentially what user typed in source code.

2) and 3) are the translation of 1) to Python and XPath.

3.1.8 Class Slice

It models the concept of slice. This concept has a start index and a stop index. Class `Slice`, following the same principle as class `Index`, represents the start and stop indices using the properties below:

1) `self.start_galaktion`
2) `self.start_python`
3) `self.stop_galaktion`
4) `self.stop_python`
5) `self.start_stop_xpath`

3.1.9 Classes SymbolType and Symbol

The `Symbol` class models the user-defined identifiers and the `SymbolType` one represents their different types.

`SymbolType` inherits from `Enum` class, and it has the values shown in the list below:

1) `folder = 0`
2) `xmlfile = 1`
3) `jsonfile = 2`
4) `csvfile = 3`
5) `flexible = 4`

`Symbol` has the following properties:

1) `self.type`
2) `self.const`
3) `self.name`
4) `self.value`

1) is of type `Galaktion_core.SymbolType`.
2) is a flag indicating whether the symbol is constant or variable.
3) holds the symbol's name
4) holds the symbol's value

3.1.10 Class SymbolTable

It implements Galaktion's symbol table where all the identifiers are stored along with their properties. It has the following properties and methods (some of them):

```
1) hiddenSymbolPrefix
2) self._symbolTable
3) lookup
4) get
5) update
```

1) is a string that is used to auto-generate hidden symbol names.
2) is a dictionary with keys the symbol names and values the `Symbol` objects.
3) searches a symbol by name – returns `True` if it exists in symbol table, otherwise `False`.
4) returns the `Symbol` object if exists, else `None`.
5) inserts if does not exist or updates if exists a symbol's object.

3.1.11 Classes ExpressionType and Expression

The `Expression` class models the Galaktion syntactic expressions and the `ExpressionType` one represents their different types.

`ExpressionType` inherits from `Enum` class, and it has the values shown in the list below:

```
a) folder = SymbolType.folder.value
b) xmlfile = SymbolType.xmlfile.value
c) jsonfile = SymbolType.jsonfile.value
d) csvfile = SymbolType.csvfile.value
e) STRING = 4
f) NONE = 5
g) ARITHMETIC = 6
h) BOOLEAN = 7
i) ARRAY = 8
```

`Expression` has the following properties:

```
1) self.type
2) self.const
3) self.symbol
```

```
4) self.value
5) self.arrayItem
6) self.indexed
```

1) is of type `Galaktion_core.ExpressionType`.
2) is a flag indicating whether the expression is constant or variable.
3) if the expression involves an identifier, holds its `Symbol` object else `None`.
4) holds the expression's value.
5) indicates whether the expression is an array's item or not.
6) indicates whether the expression is result of indexing or not.

3.2 Lexical Analyzer: Galaktion_lexer.py

The Lexical Analyzer (Lexer) converts the source code into tokens.
The possible tokens are defined using regular expression (regex) rules. Most rules are trivial so, for the sake of brevity, they will not be discussed. Only the most significant ones are going to be mentioned.

3.2.1 Keeping Track of the Last Token

For the needs of Galaktion v0.2.0, the Lexer must keep track of the last token found while it is trying to match the next token. A way of achieving this is by creating a global variable in Galaktion_lexer.py and defining all the regex rules as functions so as for each one to update the global variable with its matched token. However, this is not a good practice, and it was not followed.

Finally, the solution followed was to modify the `lex.py` file of `ply` package to keep track of the last token.
At the `__init__` method of `Lexer` class an attribute `last_token` of type `ply.lex.LexToken` was added along with its initialization to hold the last token at any time.
This attribute is then updated by the `token` method of the `Lexer` class each time it matches a new token. At six points in total throughout `token` method, right before a new token is returned, an assignment like

`self.last_token = ` [*the new token*]

was added.
This way, from `Galaktion.lexer.py`, the lastly matched token can be read through `t.lexer.last_token` inside each function-defined token rule.

3.2.2 Lexer's States

In addition to the default state `INITIAL` four more states were used:

1) `recordingCodeBlock`
2) `recordingParBlock`
3) `recordingBraceBlock`
4) `scanBraceBlock`

1) A code block is a curly brace pair (`{`…`}`) with all its contents, regardless of what they are. When in this state, the Lexer records blindly the whole block and matches it to a single `BLOCK_WITH_CODE` token.

2) If the last token was `PATH`, similar to 1) the whole block is matched as a `PAR_PATH` token. This is used to match the parentheses with the arguments of an XPath expression's function. The name of the function is matched by the `PATH` token's rule.

3), 4) Regardless of the last token, the Lexer enters this state to record a brace block (`[`…`]`). After it is recorded, the Lexer checks its validity as an XPath predicate using the `lxml` library.
If it is valid, it matches it to a `PREDICATE` token.
If it is not valid that means that it contains Galaktion code (or something faulty) so it enters state 4) to tokenize it. After the tokenization, the Lexer returns to the `INITIAL` state.

3.2.3 Rule "t_recordingBraceBlock_BRACE_R"

This is the rule that matches the "`]`" character while in the `recordingBraceBlock` state. The logic described in the **3), 4)** of the previous section is implemented here.
If the last token was neither PATH nor PREDICATE, then sure this brace block contains Galaktion code that it must be tokenized, so the Lexer just enters the `scanBraceBlock` state without further actions. The transition is done with the following lines of code:

```
t.lexer.lexpos -= len(t.value)
t.lexer.openBraceBlocks = 0
t.lexer.begin('scanBraceBlock')
```

If the last token was `PATH` or `PREDICATE` it is possible for the current block to be either predicate or Galaktion code, so the following two checks must be performed:

Second check: A dummy XPath expression followed by the brace block as its predicate is evaluated on a dummy XML tree using the `xpath` function of `lxml` library. If an exception is caught, it means that the predicate is invalid. The chances of the block containing Galaktion code are very high now and the Lexer proceeds to tokenize it by executing the code shown above.

First check: As discussed in section 2.11, although a negative-index-like predicate as `[-1]` does not behave like an actual Galaktion negative index, it is a valid XPath predicate. This means that it would pass the second check but then it would not work as the user expected. For this reason, this check precedes the second check.
It is a very simple to-float conversion of the block's contents. If the block contains only a number, the conversion will be successful and the Lexer will reach the point where it enters the `scanBraceBlock` state to effectively tokenize it. Otherwise, an exception will be caught, which will force the Lexer to procced to the second check.

3.2.4 Rule "t_INITIAL_scanBraceBlock_PATH"

It is used to match XPath expressions. But because relative ones are supported, it also matches words like `if`, `else`, `generate`, `*` etc. considering them relative XPath expressions.
To solve this, all Galaktion's reserved tokens that are going to be matched by this regex are grouped in the `reservedTokens` dictionary, with keys being the actual tokens and values being their type. Then inside this rule this line

```
t.type = reservedTokens.get(t.value, 'PATH')
```

is added. This way, the matched token will always have the correct type and will be `PATH` only when it is not reserved.

The syntax analyzer (parser) in the next stage will treat consecutive `PATH`, `PAR_PATH` or `PREDICATE` tokens as a single big XPath expression.
For example, `my /path [last ()]/ text()` will produce the following tokens:

1. `PATH`: "my"
2. `PATH`: "/path"
3. `PREDICATE`: "[last ()]"
4. `PATH`: "/"
5. `PATH`: "text"
6. `PAR_PATH`: "()"

The parser will concatenate them to the whole `my/path[last ()]/text()` XPath expression, and this is correct.

But if the input was a little different, like

```
my /path [last ()]/ text()                    my/second/path
```

the tokens would be:

1. `PATH`: “my”
2. `PATH`: “/path”
3. `PREDICATE`: “[last ()]”
4. `PATH`: “/”
5. `PATH`: “text”
6. `PAR_PATH`: “()”
7. `PATH`: “my/second/path”

The parser, as before, would concatenate them to a whole XPath expression but this time it would not be correct since the user intended to write two standalone XPath expressions, not one.

To overcome situations like this, whenever it is obvious that the `PATH` token that was just matched was intended to be separate from the last token, the Lexer produces a `DUMMY_STMT` token (`;`) before the new `PATH` token to force the parser to treat them as separate XPath expressions. This is achieved by the `if` statements in this rule.

Therefore, the tokens that this input produces are:

1. `PATH`: “my”
2. `PATH`: “/path”
3. `PREDICATE`: “[last ()]”
4. `PATH`: “/”
5. `PATH`: “text”
6. `PAR_PATH`: “()”
7. `DUMMY_STMT`: “;”
8. `PATH`: “my/second/path”

By now, in terms of XPath expression recognition, it is apparent that, in order not a full XPath parser to be built from scratch, the Lexer does a lot of the work providing the parser only a few building blocks (`PATH`, `PAR_PATH`, `PREDICATE`) to simply assemble the XPath expression and use `lxml` library for parsing and evaluating it (discussed in the next section).

3.2.5 Function “newLexer”

It accepts one argument `startingLine` of type `int` and returns a new Lexer object with its `lineno` property initialized at `startingLine`.

It is used by the parser to execute the code blocks.

3.3 Syntax Analyzer: Galaktion_parser.py

The syntax analyzer (parser) consumes the tokens produced by the Lexer, effectively parsing the input based on Galaktion's grammar. It is implemented using `yacc.py` of `ply` package, version 3.11.

In `yacc.py`, the language's grammar rules can be written inside the docstrings of functions prefixed with `p_`. Each of them accepts a single argument `p` that holds the values of each grammar symbol of the recognized rule. Each `p[i]` is mapped to the grammar symbols with the order they appear inside the rule [8].

For faster initial development, Galaktion v0.2.0's parser uses the syntax directed translation technique to translate the token streams into python source code. This is one of the major things to change in the next versions.

The `Galaktion_parser.py` imports data structures from `Galaktion_core.py` (see 3.1). The `ExpressionType` (see 3.1.11) is imported as `ExprT` and the `SymbolType` (see 3.1.9) is imported as `SymT`.
`Galaktion_core` itself is also imported as `G`.

For the sake of brevity, only the most important parts of the parser will be discussed in the following sections.

3.3.1 Variables

The following global variables are initialized:

```
1) UI = G.UserInterface()
2) symbolTable = G.SymbolTable()
3) userSrcFiles = G.Stack()
4) linkCode = None
5) linkCodeStartingLine = 0
6) linkSubfolders = None
7) currentSrcElement = None
8) fileLocation = None
9) fileName = None
10)      activeLoopCounter = 0
11)      afterErrorExpression = Expression(ExprT.NONE, True)
```

1) instantiates the `UserInterface` class.
2) instantiates the `SymbolTable` class.
3) instantiates the `Stack` class. It is used to hold with the right order the XML source files defined by `from` statement.
4) holds the arrow's code block as a string.

5) holds the line number at which the arrow's code block starts inside user's source code. When the 4) is to be parsed, the 5) will be passed to the `newLexer` function (see 3.2) as argument.
6) is a flag that becomes `True` if the user used the `#subfolders` directive in the arrow's code block, `False` otherwise.
7) is used during XML file linking to hold at each execution the element that will be the root for all the relative XPath expressions inside arrow's code block.
8) holds at each arrow's execution the specified by the user location of the file to be found.
9) holds at each arrow's execution the specified by the user name of the file to be found.
10) counts the active loops. Outside a loop it should be zero.
11) is essentially a dummy `Galaktion_core.Expression` instance which is used as the result of an expression where an error occurred.

3.3.2 Grammar Rule "fromstmt" – Defining XML Source Files

```
def p_fromstmt(p):
    '''
    fromstmt : FROM expr
             | FROM expr BLOCK_WITH_CODE
    '''
```

If expr is not of `xmlfile` type it emits an error.

If a code block exists it adds a scope to `userSrcFiles`, inserts `p[2]` (the value of `expr`) to the new scope, executes the block, and deletes the new scope using the following code:

```
userSrcFiles.push(p[2].value)
parser.parse(p[3][1:-1], lexer=newLexer(p.lineno(3)))
userSrcFiles.pop()
```

Otherwise, it just updates the current source XML file in the current scope, using the code below:

```
userSrcFiles.pop()
userSrcFiles.push(p[2].value)
```

3.3.3 Grammar Rule "linkexec" – Performing Linking

```
def p_linkexec(p):
    '''
    linkexec : expr drctvs COMMA expr drctvs
    '''
```

This is the rule that is recognized at each execution of `linkCode`'s (see 3.3.1) contents.

It spots which of the expr symbols carries the file name and which carries the file location and updates the global variables `fileName`, `fileLocation`, and `linkSubfolders` accordingly.

If an `expr` holds non-string-based value, the corresponding error is emitted. This is done because both file location and name are by nature string values.

3.3.4 Grammar Rule "foreachstmt" – Emulating the Loop

```
def p_foreachstmt(p):
    '''
    foreachstmt : FOR_EACH decl IN expr BLOCK_WITH_CODE drctvs
    '''
```

It increments and decrements the `activeLoopCounter` variable (see 3.3.1) accordingly to be able to identify whether a `brake` or `continue` statement is not inside a loop. If so, it emits the corresponding error.

If the value held by `expr` is of type `ExprT.ARRAY` it emulates the loop's behavior using the `exec_foreach_array` function while if the value is of type `ExprT.folder` it uses the `exec_foreach_folder` one.
If the value is of another type, it emits error.

3.3.5 Grammar Rule "generatestmt"

```
def p_generatestmt(p):
    '''
    generatestmt : GENERATE decl AS STRING_LITERAL drctvs
    '''
```

If the type of `p[2]` is `ExprT.jsonfile` it uses the `json.dumps` function to convert the contents of `G.memory` (see 3.1.4) to JSON string and then writes the string into the output file specified by `p[3]`.
If the type of `p[2]` is `ExprT.csvfile`, it uses the `json_normalize` function of the `pandas` library to handle nested data in the internal memory and then using the `to_csv` function of same library it creates the CSV file that `p[4]` indicates.

3.3.6 Rules “xpath”, “linkedxpaths” and “linkedxpathsexpr” – Retrieving XML Data

```
def p_linkedxpathsexpr(p):
    '''
    linkedxpathsexpr : linkedxpaths
    '''

def p_linkedxpaths(p):
    '''
    linkedxpaths : xpath
                 | linkedxpaths ARROW xpath
    '''

def p_xpath(p):
    '''
    xpath : MUL
          | PATH
          | PATH PAR_PATH
          | xpath predicates
          | xpath index
          | xpath slice
          | xpath PATH
          | xpath PATH PAR_PATH
    '''
```

The `xpath` rule is the one that reassembles as a simple string the previously disassembled by the Lexer XPath expression which the user typed. Most of its grammar symbols are strings therefore all it takes for the expression to be formed again is to concatenate them all together. So, at the end of the rule’s semantic actions `p[0]` will be a string.

The `linkedxpaths` is recognized after either a single xpath recognition or multiple xpath recognitions with ARROW recognized between them. Here is where most of the work is done.

In case of the single xpath: The value of currentSrcElement is checked. If it is `None` means that the recognized xpath was outside the arrow’s code block where, as mentioned earlier, relative XPath expressions are unexploited. As a result, it parses the top element of the userSrcFiles stack and evaluates the p[1] on its root XML node.
On the other hand, if the value is not `None`, it means that the p[1] was recognized inside the arrow’s code block during its execution Therefore the p[1] is evaluated

directly on currentSrcElement and this is how relative XPath expressions are exploited while the `->` performs the linking from one XML file to another.

The above are implemented with the use of `lxml` library with the lines shown below:

```
if currentSrcElement == None:
   val = etree.parse(userSrcFiles.top()).xpath(p[1], **xpathVars)
else:  # when executing link code
   val = currentSrcElement.xpath(p[1], **xpathVars)
```

The xpathVars variable, passed as argument to xpath function, is a dictionary produced by getDynamicXPathVariables function a few lines earlier. It contains the names of the variables inside the XPath expression along with their values, read from the symbol table. It is automatically unpacked using the operator `**`, providing the `xpath` function with the named arguments it expects.

On each case, the results of the XPath expression are stored inside the `val` variable. Because, at user level, any XPath expression is normally evaluated into a Galaktion array expression (unless it has only one item), if `val` is not of type `list` already, it is converted to list and it is assigned to `p[0]`.

In case of multiple xpath: It follows a two-step process. The first step is to perform the linking procedure on each item in p[1] to find each item's linked file and the second step is to evaluate p[3] on each of the linked files found, assigning the results to p[0].

For the linking to be performed, the `getPath2File` function is called for every p[1][i] that is of type `etree._Element`. On each call, `getPath2File` internally executes the arrow's code block, effectively performing the linking.

Then, on every value returned by `getPath2File` the `p[3]` is evaluated, as shown below:

```
 val = etree.parse(r).getroot().xpath(p[3], **xpathVars)
```

Again here, `val` is converted to list, if it is not already.
`r` is the current result of `getPath2File`.
`xpathVars` again holds the named arguments for `xpath` function, as described earlier.

3.3.7 Initialization and Run

When the parser is executed, it firstly initializes the `$GALAKTION_ROOT` variable (see 2.4.3) to hold the path to the Current Working Directory. This is done by inserting into the symbol table a new variable `Symbol` object of type `SymT.folder` named `$GALAKTION_ROOT` with the value of `os.getcwd` function. The initialization code follows:

```
symbolTable.update(
    Symbol(
        type=SymT.folder,
        const=False,
        name='$GALAKTION_ROOT',
        value=os.getcwd()
    )
)
```

Then, the user's Galaktion source code contained in the input file specified by the command line arguments is read into the `parserInput` variable, using the code below:

```
try:
    with open(UI.userSourceFile, 'r') as inp:
        parserInput = inp.read()
except FileNotFoundError:
    UI.print('Error: Input file not found')
    sys.exit()
```

A parser object is then created, and the parsing of `parserInput` is started by the following code:

```
parser = yacc.yacc() # Create the parser object.

try:
    parser.parse(parserInput, lexer=newLexer(1))
except KeyboardInterrupt:
    UI.print('Aborting...')
    sys.exit()
```

Finally, after parsing is complete, if there are any warnings or errors, they are printed according to the following two lines of code:

```
UI.printWarnings()
UI.printErrors()
```

4 Web Map Application: Visualizing “RICONTRANS” Data

The goal of this Application is to visualize the data of the RICONTRANS project on a map with a modern and easy-to-use Graphical User Interface (GUI).

4.1 The “RICONTRANS” Project

RICONTRANS (Visual Culture, Piety and Propaganda: Transfer and Reception of Russian Religious Art in the Balkans and the Eastern Mediterranean (16th – early 20th century)) is a research project led by Dr. Yuliana Boycheva, art historian, funded by the European Research Council (ERC) under the Horizon 2020 program.

The project studies the transfer and reception of Russian religious art in the Balkans and the Eastern Mediterranean between the 16th and 20th centuries, examining aesthetic, ideological, political and social factors.

The research team collects and manages data in collaboration with the Centre for Cultural Informatics (CCI) of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), integrating the data into the CIDOC Conceptual Reference Model.

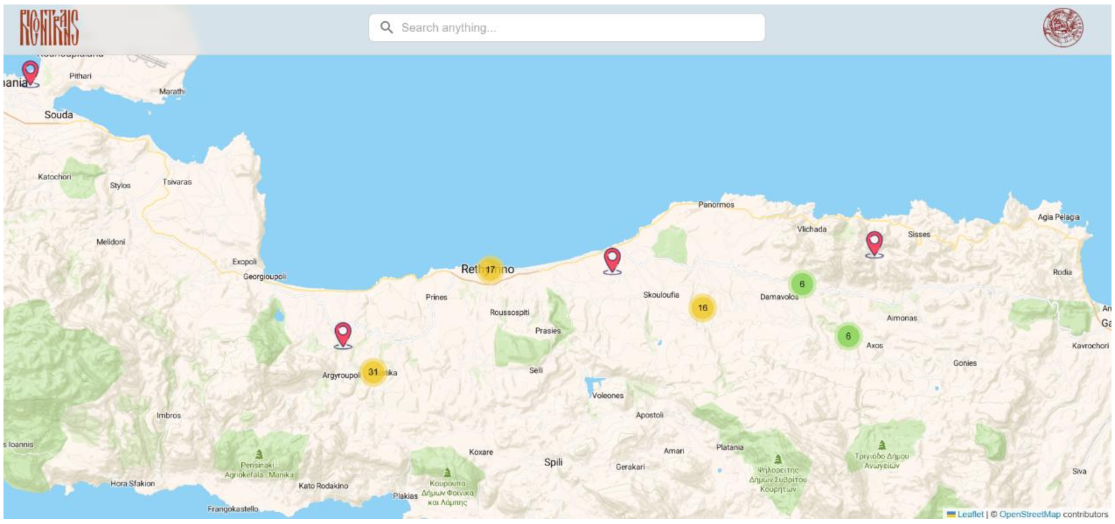
4.2 Implementation

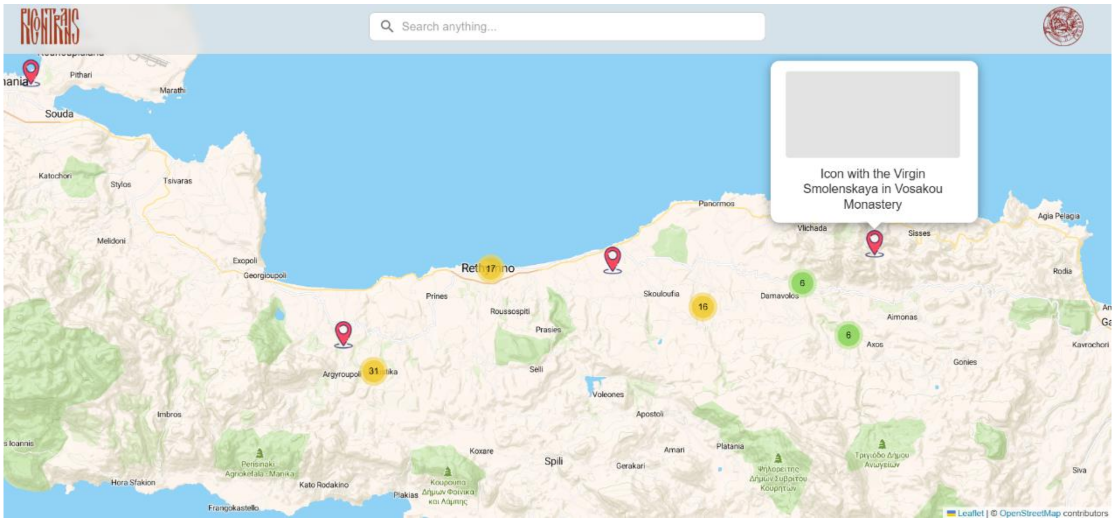
For the web map app development, the following tools and technologies were used:

- Node.js
- React
- Vite
- Leaflet.js
- React Leaflet
- Material UI

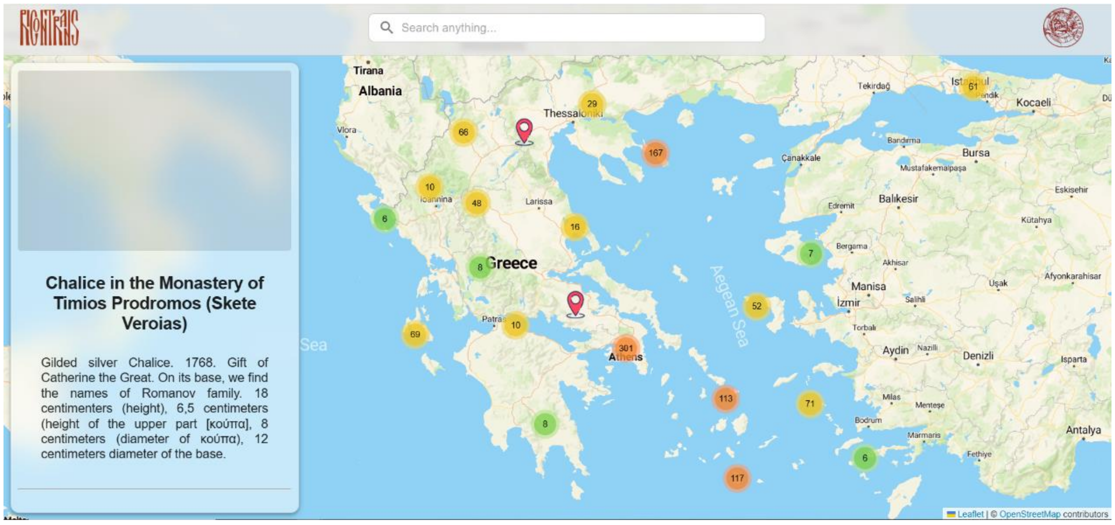
The full code is available on GitHub.
Some pictures of the Application running visualizing the RICONTRANS data extracted using Galaktion as JSON format into `Objects.json` are shown below.

Here the reader can see the Galaktion-Map App cooperation: The one prepares the data and the other visualizes them.



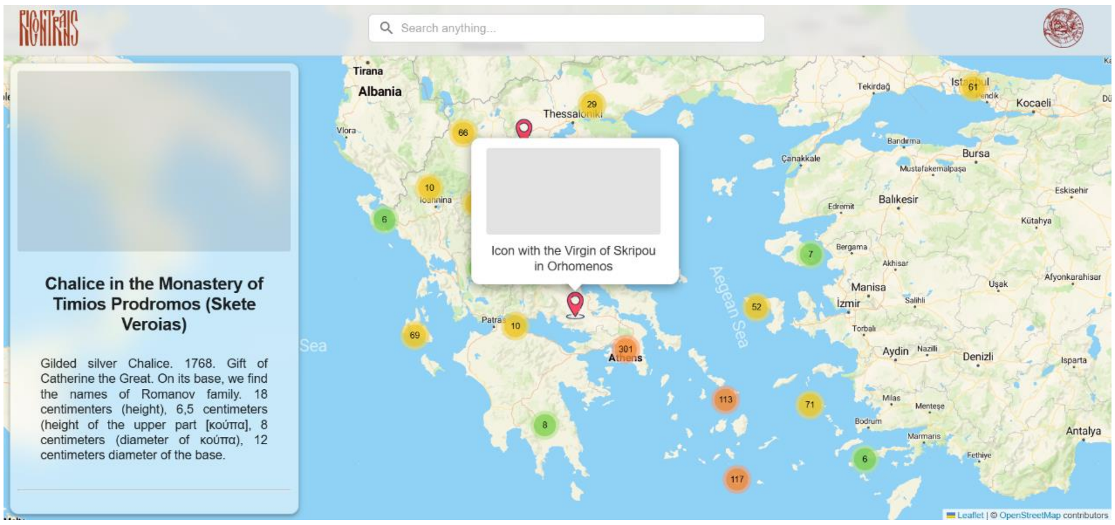


Icon with the Virgin Smolenskaya in Vosakou Monastery



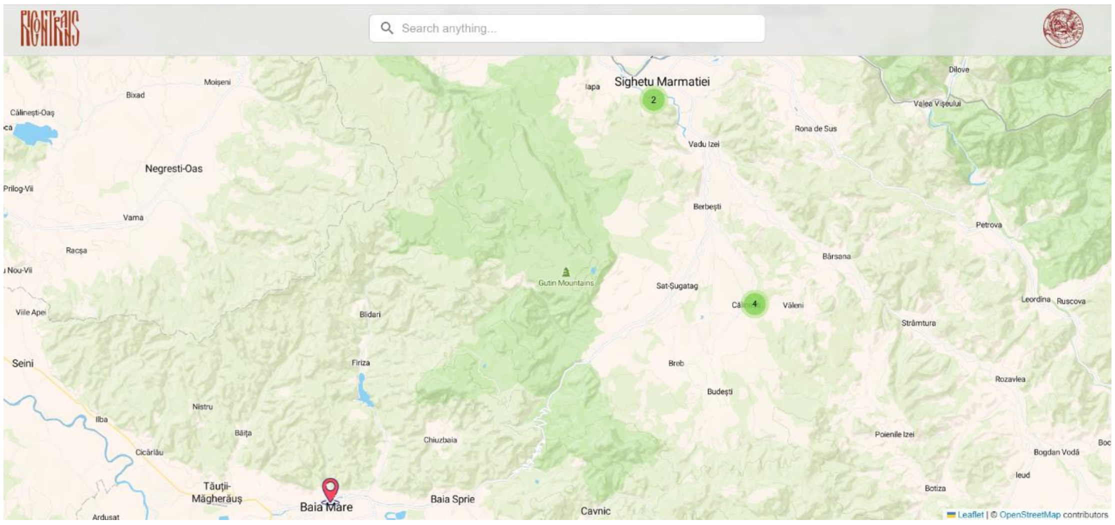
Chalice in the Monastery of Timios Prodromos (Skete Veroias)

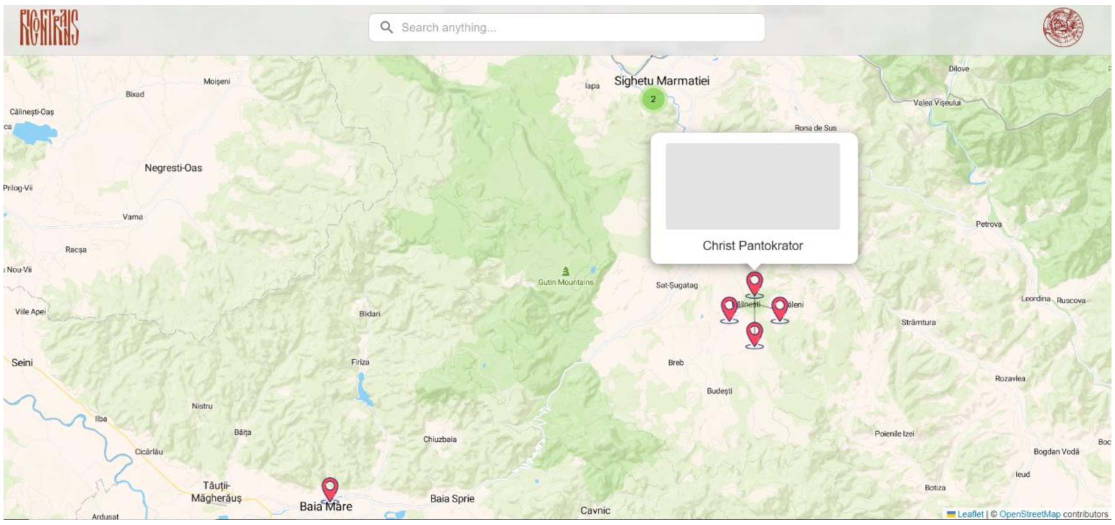
Gilded silver Chalice. 1768. Gift of Catherine the Great. On its base, we find the names of Romanov family. 18 centimeters (height), 6,5 centimeters (height of the upper part [κούπα], 8 centimeters (diameter of κούπα), 12 centimeters diameter of the base.

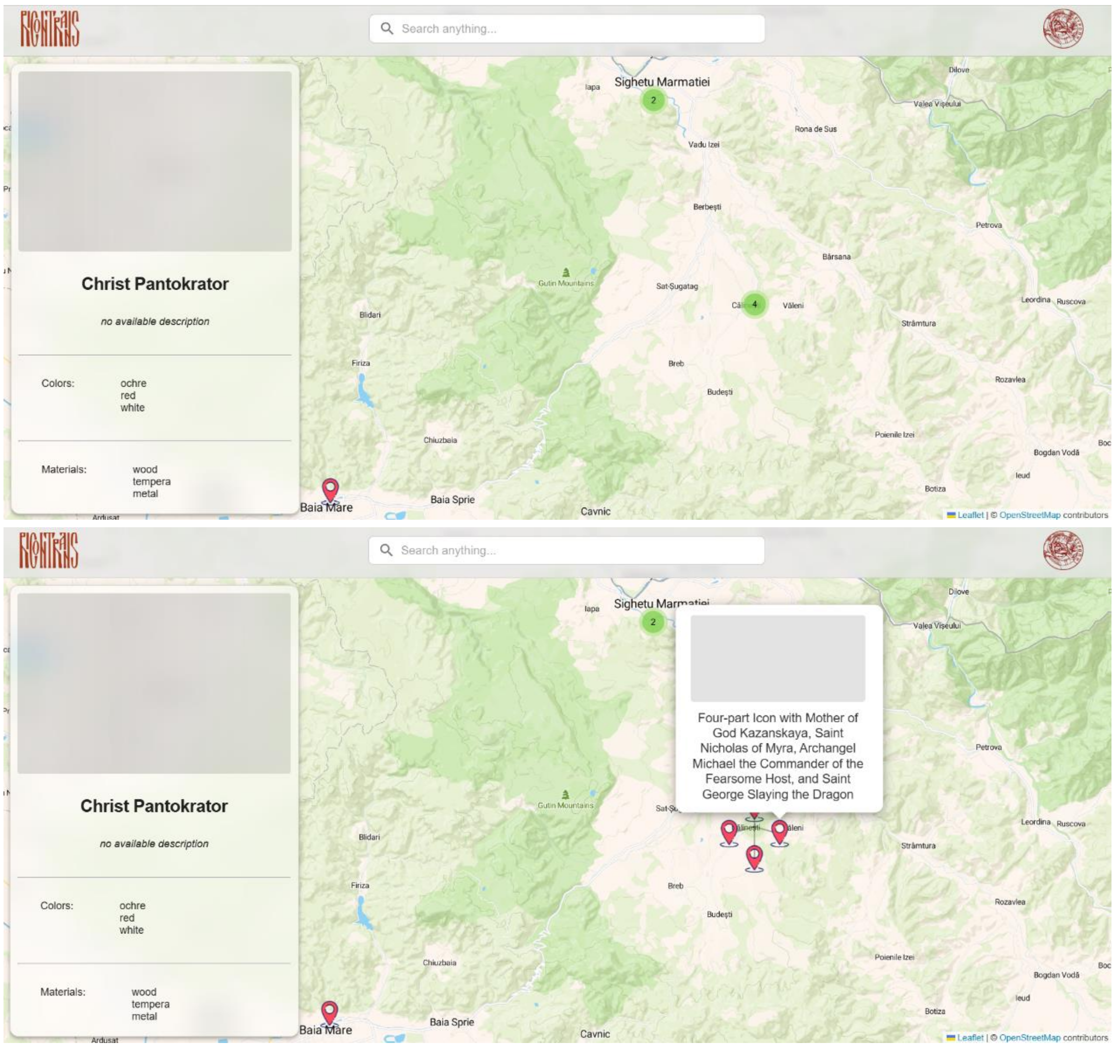


Chalice in the Monastery of Timios Prodromos (Skete Veroias)

Gilded silver Chalice. 1768. Gift of Catherine the Great. On its base, we find the names of Romanov family. 18 centimeters (height), 6,5 centimeters (height of the upper part [κούπα], 8 centimeters (diameter of κούπα), 12 centimeters diameter of the base.






Christ Pantokrator

no available description

Colors: ochre red white

Materials: wood tempera metal

Four-part Icon with Mother of God Kazanskaya, Saint Nicholas of Myra, Archangel Michael the Commander of the Fearsome Host, and Saint George Slaying the Dragon

Figure 4.1 | Snapshots of the Web Map App's execution that demonstrate a bit of its functionality

4.3 Galaktion's Contribution

Galaktion was used to extract specific data from the XML files of RICONTRANS project and create the `Objects.json` file which the Application uses to draw the markers on the objects' positions on the map.
The code used is shown below:

```
$GALAKTION_ROOT .= 'path / to / RICONTRANS_DATA_2023-10-24'

-> {
    @sps_type                         #filelocation #subfolders ,
    @sps_type + @sps_id + '.xml'      #filename
}


folder $objectsFolder .= 'Objects'
```

```
foreach xmlfile $file in $objectsFolder {
    ` if it is not saved, we ignore it `
    if //admin/saved/text() = 'yes' {
        `store the value of $file in this iteration as srcFile`
        extract $file as 'srcFile'


        `store the id of the current XML file as key`
        extract //admin/id/text() as 'key'


        `store the object's name and description as follows`
        extract //IdentityOfObject/ObjectNameOfficial[1]/text()
             as 'Name'
        extract //IdentityOfObject/Description/text()
             as 'Description'


        `store the object's materials and colors as follows`
        extract //PrincipalMaterial/text() as 'Materials'
        extract //BasicColour/text()       as 'Colors'

        `find the node CurrentLocation/Location inside the $file
        and using -> go to the file that it indicates
        and inside that file, find the node Name, get its text
        and store it as 'Current Location Name' `
        extract //CurrentLocation/Location->Name/text()
             as 'Current Location Name'

        `find the node CurrentLocation/Location inside the $file
        and using the -> go to the file that it indicates
        and inside that file, find the node Coordinates/Value,
        get its text, convert it to lat and long and store it as
        'Current Location Coordinates' `
        extract //CurrentLocation/Location->Coordinates/Value/text()
             as 'Current Location Coordinates' #latlong
    }
} #subfolders ` iterate through the subfolders of $objectsFolder
too `


` generate a JSON file Objects.json in the specified location `
` with all the data that have been extracted until now `
` after the process is complete, empty the internal memory
(#flush) `
generate jsonfile
      as 'path / to / webapp / src/Objects.json' #flush
```

Figure 4.2 | The Galaktion code used to create the Objects.json file the Map App uses

5 Conclusion and Future Work

When it comes to XML Data management, the most common choice is the powerful XQuery.
However, Galaktion too, exposing another perspective of the field, offers several benefits and simplifies situations that would be difficult or even impossible to overcome using XQuery.

Some of Galaktion's advantages are listed below:

- Its imperative and general programming nature is closer to how the human brain works.
- It can be expanded to be capable of doing many complicated and useful things, while XQuery's querying nature does not leave a lot of room for future expansions.
- Galaktion makes the management of linked XML data very easy and intuitive with its straightforward arrow syntax (see section 2.9).
- It offers directives like `#latlong` which convert coordinates expressed in XML directly to a lat and long value, with a very user-friendly way.
- It has user-friendly syntax, which makes it suitable for users that are not necessarily involved in computer programming.

Adding to that, Galaktion v1.0.0 is expected to enhance the above nice-to-have features.
The current plans for Galaktion's expansion by version 1.0.0 include:

- Making it a compiled language.
- Making an Abstract Syntax Tree and using an Intermediate Code Generator on it.
- Replace string-based types with actual path expressions as separate literals.
- Maybe abolishing the `flexible` type specifier.
- Adding native support for XML syntax directly into source code and making a new data type that will represent the XML tree structure.
- Adding native support for JSON and CSV syntax directly into source code and making the new data types needed to hold their structures.
- Exploiting of relative XPath expressions.
- Improving XPath support.
- Adding support for user-defined functions.
- Adding support for user-defined directives
- Adding support for type casting and easy inline value-of-specific-type creation.

- Adding support for different arrow definitions to allow different types of linking.
- Adding support for the slices below:
    - `(…~…)` : excusive start, exclusive stop
    - `(…~…]` : exclusive start, inclusive stop
    - `[…~…)` : inclusive start, exclusive stop
- Adding more command line arguments.
- Adding support for external Galaktion code usage.
- Adding support for XML Schema importing.
- Potential changes to the syntax for making it more user friendly.

and more.

6 References

[1] Wikipedia, «XML - Wikipedia,» Wikipedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/XML. [Πρόσβαση 9 November 2025].

[2] World Wide Web Consortium (W3C), «XML Path Language (XPath),» World Wide Web Consortium (W3C), [Ηλεκτρονικό]. Available: https://www.w3.org/TR/xpath-10/. [Πρόσβαση 10 November 2025].

[3] Wikipedia, «XPath - Wikipedia,» Wikipedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/XPath. [Πρόσβαση 10 November 2025].

[4] Wikipedia, «XQuery - Wikipedia,» Wikipedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/XQuery. [Πρόσβαση 10 November 2025].

[5] World Wide Web Consortium (W3C), «XQuery 3.1: An XML Query Language,» World Wide Web Consortium (W3C), [Ηλεκτρονικό]. Available: https://www.w3.org/TR/xquery-31/. [Πρόσβαση 10 November 2025].

[6] json.org, «JSON,» json.org, [Ηλεκτρονικό]. Available: https://www.json.org/json-en.html. [Πρόσβαση 10 November 2025].

[7] Wikipedia, «Comma-separated values - Wikipedia,» Wikipedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Comma-separated_values. [Πρόσβαση 10 November 2025].

[8] D. M. Beazley, "PLY (Python Lex-Yacc)," [Online]. Available: https://www.dabeaz.com/ply/. [Accessed 5 November 2025].