The complete TheMatrix syntax

This document fully describes the syntax of the TheMatrix Language. The document is aimed to reader that already has some confidence with the TheMatrix language. The document is structured as follows. Section 1 (Basic Syntax) contains the syntax of the barebone elements of the language, such as modules, comments, types, etc..; Section 2 gives the syntax about all the available modules; Section 3 provides a description of all function elements, including predicates and expressions.

1. Basic Syntax

Note: White space is non-significant

Identifiers

An identifier is an alphanumeric string which may contain underscores, but no spaces. For example, “My_Module” is a correct identifier, while “My Module” is wrong.
Common identifiers are:
<ModuleId>, <SchemaName>, <ColumnId>, <FunctionId>

Comments

The content inside a comment block is ignored in the program execution. TheMatrix supports both single line and multi-line comments. Example:

```
// single line comment
/* multi
     line
 comment
*/
```

Types

There are six available <type> :

int
integer value, **literal: 10**

float
floating point, value **literal: 23.5**

string
string value, **literal: "Hello World"**

boolean
boolean value, **literal: true**, **false**

date
timestamps. **literal**: YYYY-MM-DD␣hh:mm:ss that is 4-digit year, '-', 2-digit month, '-' 2-digit day. The time of the day is optional. Examples: “2013-01-08”, “2013-01-08 15:07”, “2013-01-08 15:07:33”.

missing

is only available as the literal **MISSING** and it represents a value that is unknown, unavailable or undefined

Note: there is no regex literal, matches are against simple string patterns such as "**foo***".
Note**:** the domain of each type implicitly contains **MISSING,** i.e. any value can be compared to MISSING

Literal values
<LiteralValue> is either an int literal value, a float literal value, a string literal value, a boolean literal value, a date literal value, the literal **MISSING** or a **param.**

Params

A param is an Identifier preceded by a dollar-sign, for example $MY_VAR.

Literal values
<Value> a value is either a LiteralValue, a ColumnId or a Param

Lists of values
Lists of values are between square brackets, each element is separated by commas.
Example: [Val1, Val2, …]
A singleton list contains no commas.
Example: [Val]

Type annotations
Type annotations are in curly brackets, when an identifier must be qualified with its type.
Example: { Identifier : <type> }
Used in schema definitions (e.g. NewDataModule, ParameterModule)

Comparisons
Curly brackets denote comparisons as well: example { MyColumn < 10 }

1.2 Declare Schema

It is possible to declare a new schema. Declarations (as the following) must be put at the **beginning** of the script, and can be referenced within any subsequent module.

declareSchema <NewSchemaName> = [{ ColumnId1 : <type> }; { ColumnId2 : <type> }; …]

2. Modules

There are 18 available modules. For each one, it is given a brief semantic description and the full syntax.

FileInputModule

Loads a csv file from disk.

```
<ModuleId> (FileInputModule)
parameters
  inputFilename = nomefile.csv
  inputSchema   = <SchemaName> // the schema the file is expected to have
  orderBy = [ ColumnId1, ColumnId2, … ]
end
```

FileOutputModule
Writes a csv file to disk.

```
<ModuleId> (FileOutputModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  checksum = none | md5
  compression = none | zip | gzip
end
```

ParametersModule
Allows the definition of script-wide parameters.

```
<ModuleId> (ParametersModule)
parameters
  params = [ { $PARAM1 : <type> }; { $PARAM2 : <type> }, … ]
end
```

FilterModule
Allows to filter out tuples.

```
<ModuleId> (FilterModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  conditions = [ { ColumnId1 <ComparisonOp> <Value> }; … ]
  boolExpr = AND | OR
  filterType = keep | discard
end
```

Filter type
filterType is optional and defines whether to keep or discard the matching tuples. It can have two values:
keep: The filter will *keep* every tuple where the condition holds, and discard any other
discard: The filter will *discard* every tuple where the condition holds, and keep any other
The default value is keep.

Available Operators

<ComparisonOp> is one among **“**<”, “>”, “<=”, “>=”, “!=”, “=”, or the **matches** keyword. In the latter case, <Value> is always a string literal representing a pattern**.**
A pattern is a string containing the wildcards ***** or **?** with their usual meaning:
- * one or more character
- ? exactly one character

For instance: "po?r*" matches poor, poorer, poorest, pour, power, powerless, etc.

Example

For example, a filters that wants to discard (or keep) tuples in which COL1 is less than 10, COL2 is not missing and COL3 matches strings like patt*ern would have the following conditions:

*conditions = [{ COL1 < 10}; {COL2 != MISSING}; {COL3 matches "patt*ern"};]*

Note**:** The domain of each type implicitly contains MISSING, i.e. any value can be compared to MISSING

ExtendDataModel

Extends a Data Model with additional fields.

```
<ModuleId> (ExtendDataModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  attributes = [ { ColumnId1 : <type> }; { ColumnId2 : <type> }; … ]
end
```

ApplyFunction

Apply a custom function to a tuple

```
<ModuleId> (ApplyFunction)
inputs
  <SchemaName> = <ModuleId>
parameters
  function = FunctionId ( Value1, Value2 … )
  result   = ColumnResult
  // optional:
  conditions = [ { ColumnId1 <ComparisonOp> <Value> }; … ]
  boolExpr = AND | OR
end
```

A <value> can be either a variable, column, or a literal value (a constant).
Please refer to Section 3 for the lists of the available functions.

MergeModule

Merges two different flows of tuples with a common key.

```
<ModuleId> (MergeModule)
inputs
  <SchemaName> = <ModuleId>
  <SchemaName> = <ModuleId>
parameters
  primaryKey  = [ ColumnId1, ColumnId2, … ]
  fieldsName  = [ ColumnId1, ColumnId2, … ]
end
```

DropModule
Drops an entire column.

```
inputs
    <SchemaName> = <ModuleId>
parameters
     params = [ ColumnId1, ColumnId2, … ] // colonne da eliminare
end
```

Note: the DropModule produces a custom schema.

ProductModule
Applies a conditional join to two flows of tuple.

```
<ModuleId> (ProductModule)
inputs
  <SchemaName> = <ModuleId>
  <SchemaName> = <ModuleId>
parameters
  IDfield   = ColumnId
  functions = [
     <ProductFunction1>, <ProductFunction2>, …
  ]
  boolExpr = AND | OR
end
```

The ProductFunction is defined as:
FunctionId1 (Value1, Value2, …)
where each value can be either a variable, **qualified** column, or a literal value (a constant). A *Qualified Column* is <ModuleId> **.** <ColumnId>

Please, refer to Section 3 for the lists of the available functions.

Example

```
MyModule (ProductModule)
inputs
        IADoutpat = SortedOutpat_File
        IADdrug   = SortedDrug_File
parameters
```

```
        IDfield   = PatientID
        functions = [ equalsTo(
                  SortedOutpat_File.PROC_COD,
                      "89.41", "89.42", "89.43", "89.44", "89.44.1", "89.44.2", "92.05.1"
                   ),
                  match (
                    SortedDrug_File.ATC,
                    "C07*", "C08*"  // string literals
                  )
       ]
end
```

SortModule
Sorts the tuples according the fields given.

```
<ModuleId> (SortModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  fieldNames  = [ ColumnId1, ColumnId2, … ]
end
```

ScriptInputModule
Provides the ability of using nested scripts.

```
<ModuleId> (ScriptInputModule)
parameters
        scriptFilename = <ScriptName.txt>
        scriptParams = [ $VAR1, $VAR2, … ]
        inputName    = ModuleId // result of the module
    expectedSchema = <SchemaName> // schema that the input module is expected to
have
end
```

RenameDataset
Renames a dataset.

```
<ModuleId> (RenameDatasetModule)
inputs
  <SchemaName> = <ModuleId>
end
```

Note: <*ModuleId*> is the new name of the old dataset

RenameAttributesModule
Renames an attribute (a column in csv).

```
<ModuleId> (RenameAttributesModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  inputAttributes  = [ col1, col2, … ]
  outputAttributes = [ col1', col2', … ]
end
```

AggregateModule

```
<ModuleId> (AggregateModule)
inputs
  <SchemaName> = <ModuleId>
parameters
  isInputSorted = true | false
  groupBy =  [ Column_1, Column_2 ] // MEMO: supportato?
  functions = [ Function1 ( Value_1, Value_2, …), ... ]
 results =  [ { Column_Id : Type } ]
end
```

Available Functions for the aggregate module
Functions are in the form FUNC(X), where X is a column name.

MIN(COLUMN)	returns the smallest value in the given column
MAX(COLUMN)	returns the largest value in the given column
COUNT(COLUMN)	returns the number of rows
SUM(COLUMN)	computes the sum of the values in the given column
AVG(COLUMN)	computes the average
STDEV(COLUMN)	computes the standard deviation

Union
Returns a module that appends the tuples from the second to the tuples of the first module listed in the inputs section

```
<ModuleId> (UnionModule)
inputs
  <SchemaName> = <ModuleId>
  <SchemaName> = <ModuleId>
end
```

3. Available Function

Names of these functions are **case insensitive.**

3.1 Predicates

The following functions are available in *ProductModule* as predicates, but can also be used in the *ApplyFunction* as expressions returning boolean values.

equalsTo(Val1 : string, Val2 : string) : boolean
  params 2 string parameters
  returns true when Val1 = Val2, false otherwise

lessThan(Val1 : anyValue, Val2 : anyValue) : boolean
  params 2 parameters of the same type
  returns true when Val1 < Val2, false otherwise

greaterThan(Val1 : anyValue, Val2 : anyValue) : boolean
  params 2 parameters of the same type
  returns true when Val1 > Val2, false otherwise

notEqual(Val1 : anyValue, Val2 : anyValue) : boolean
  params 2 parameters of the same type
  returns true when **it's not** Val1 = Val2, false when **it is**

match(InputString : string, Pattern : string) : boolean
  params a column and a string constant (or a variable) representing a pattern
  returns true when they match, false otherwise

elapsedTimeInRange(Date1 : date, Date2 : date, leftEndPoint : int, rightEndPoint : int) : boolean
  params 2 or more date parameters
  returns true when the elapsed time is within the given range (the interval is closed)

3.2 Expression

The following functions can be used in the module **ApplyFunction** as expressions**.**

Date Manipulation

elapsedTime(Date1 : date, Date2 : date) : int
  params 2 or date values
  returns an integer value representing how much time has passed in days

Year(Val1 : date) : int
  params 1 date value
  returns an integer value representing the year

min(Val1 : date, Val2 : date, …) : date
  params 2 or more date values
  returns the oldest date in the list

String Manipulation

Concat(Val1 : string, Val2 : string , ...) : string
  params 2 or more string values
  returns the concatenation of the given values

Trim(Val1 : string) : string
  params 1 string value
  returns a new string with whitespace trimmed at the beginning and at the end

Replace(Val1 : string, Val2 : string, Val3 : string) : string
  params 3 string values
  returns a new Val1 in which the sequence of Val2 is replaced by Val3

Number Manipulation

Many of these functions come in two versions, int and float. The int version accepts only integer parameters. The float version accepts float and int parameters (the int will be casted to float) but it always returns a float value.

Sum(Val1 : int, Val2 : int, …) : int
  params 2 or more int values
  returns the sum of the values in the list

Sum(Val1 : float, Val2 : float, …) : float
  params 2 or more float values
  returns the sum of the values in the list

Substract(Val1 : int, Val2 : int, …) : int
  params 2 or more int values
  returns the subtraction from the first value of the other values

Substract(Val1 : float, Val2 : float, …) : float
  params 2 or more float values
  returns the subtraction from the first value of the other values

Inverse(Val1 : int, ...) : int
  params 1 or more int values
  returns the inverse of the sum of its arguments (at least one)

Inverse(Val1 : float, ...) : float
  params 1 or more float values
  returns the inverse of the sum of its arguments (at least one)

Prod(Val1 : int, ...) : int
  params 1 or more int values
  returns the product of all the arguments

Prod(Val1 : float, ...) : float
params 1 or more float values
returns the product of all the arguments

Division(Val1 : float, Val2 : float) : float
params 2 float values
returns the division of the two arguments

Reciprocal(Val1 : float) : float
params 1 float values
returns the reciprocal of the argument

Floor(Val1 : float) : int
params 1 float value
returns the largest int less than the argument

Ceil(Val1 : float) : int
params 1 float value
returns the smallest int greater than the argument

Round(Val1 : float) : int
params 1 float value
returns the closest int to the argument

Random

Note: currently there is no possibility to indicate the seed of the random generators

Irandom() : int
params none
returns a random int value

Frandom() : float
params none
returns a random float value

Identity

Id(Val1 : string) : string
params 1 string value
returns the given value

Id(Val1 : boolean) : boolean
params 1 boolean value
returns the given value

Id(Val1 : int) : int
params 1 int value
returns the given value

Id(Val1 : float) : float
  params 1 float value
  returns the given value

Id(Val1 : date) : date
  params 1 date value
  returns the given value

3.3 Product code and ward mapping

Atc(Val1 : string) : string
  params a product code
  returns the atc of the given product code

Duration(Val1 : string) : float
  params a product code
  returns the duration of the given product code

Typeofward(Val1 : string) : string
  params a ward prefix
  returns the type of ward from the ward prefix (either 2 characters, or 3 characters with a 0 prefix)

Typeoffullward(Val1 : string) : string
  params a full ward
  returns the type of ward from the full ward, getting the initial substring of exactly 2 chars

Typeofproc(Val1 : string) : string
  params a product code
  returns the type of outpat of the given product code